An FPGA-Based Hardware Development System with Multi-Gigabyte Memory Capacity And High Bandwidth

by

Joshua Fender

A thesis submitted in conformity with the requirements
For the degree of Master of Applied Science
Graduate Department of Electrical and Computer Engineering
University of Toronto

An FPGA-Based Hardware Development System with Multi-Gigabyte Memory Capacity And High Bandwidth

Joshua Fender
Masters of Applied Science
Graduate Department of Electrical and Computer Engineering
University of Toronto
2005

Abstract

FPGA-based hardware development systems are extremely useful for exploring exciting applications in vision, graphics, and many other computationally intensive problems. Experience with previous systems has shown that memory capacity, inter-FPGA bandwidth, host-to-FPGA bandwidth, and memory bandwidth are all critical to the successful implementation of high performance systems. This thesis presents the design, and implementation, of a new FPGA-based development system that was created with the goal of providing as much performance in these four areas as feasible. The design was built with 8GB of memory and its bandwidth performance was measured. The system has 17.6GB/s total aggregate memory bandwidth, and 154MB/s (read) and 266MB/s (write) host-to-FPGA bandwidth. The result is a working development system that is capable of implementing the applications of the future.

ACKNOWLEDGMENTS

I would like to thank my supervisor Jonathan Rose for the guidance, motivation, and even the funding, he provided over the last several years. Also deserving thanks are Dave Galloway and Marcus van Ierssel for providing their experiences with previous Transmogrifiers to help guide the design of the TM-4.

I would also like to acknowledge the support, in knowledge, FPGAs, and project funding, that Altera provided, as well as the funding providing by Micronet. I would also like to recognize Altera's FAE Cheug Ng for both attending the specification review and for providing helpful information about using Altera's FPGAs.

Everyone who has passed through the LP392 lab over the last few years including: Anish, Peter Y, Peter J, Tom, Leslie, Navid, Ahmad, Ian, Aaron, Jason, Andy, Reza, Rubil, Imad, Denis, and Paul, also deserve acknowledgement. I would also like to acknowledge as honorary LP392ers: Andrew, Frank, and Blair, not so much for contributing to the Transmogrifier but for providing the distractions that turned a simple Masters into a twenty nine month affair.

TABLE OF CONTENTS

TABLE OF CONTENTS

TABLE OF CONTENTS

TABLE OF CONTENTS

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ACRONYMS

API ..... Application Program Interface
CMOS..... Complementary Metal Oxide Semiconductor
CPU ..... Central Processing Unit
DDR ..... Dual Data Rate
DRC..... Design Rule Check
FIFO ..... First In First Out
FPGA..... Field Programmable Gate Array
FPP ..... Fast Passive Parallel
IBIS ..... I/O Buffer Information Specification
IC..... Application Specific Integrated Circuit
IO ..... Input/Output
IP ..... Intellectual Property
ISA ..... Industry Standard Architecture
LUT ..... Lookup Table
LVDS ..... Low Voltage Differential Signalling
PCI..... Peripheral Component Interconnect
PCI-X ..... Peripheral Component Interconnect eXtended
PLL..... Phase Locked Loop
RAM..... Random Access Memory
SDRAM..... Synchronous Dynamic Random Access Memory
SRAM..... Static Random Access Memory
TM..... Transmogrifier
TTL ..... Transistor-Transistor Logic
USB ..... Universal Serial Bus
VHDL..... VHSIC Hardware Description Language
VHSIC..... Very High Speed Integrated Circuits
VME ..... Versa Module Eurocard

1 INTRODUCTION

1.1 FPGA-BASED RAPID PROTOTYPING

An FPGA-based rapid prototyping system is a set of hardware and software components that enable hardware engineers to design and implement high speed digital systems both quickly and cheaply. Typically, the hardware components consist of a number of programmable FPGAs, some memory, some peripherals, and a link to a host computer. The software components usually consist of a design tool flow, such as synthesis, placement and routing tools, and an IP library. Through the use of a properly designed hardware platform, an engineer can design and test many different digital systems without having to design a physical hardware platform for each. The only limitations on what is possible are those that arise from the hardware platform itself.

It is the goal of this research to design a next generation FPGA-based prototyping system that removes a number limitations found in existing prototyping systems. In particular this thesis will focus on improving four key areas: memory depth, memory bandwidth, inter-FPGA bandwidth, and host computer bandwidth.

1.2 FPGA-BASED RAPID PROTOTYPING LIMITATIONS

The primary limitation of early single FPGA-based prototyping systems was the small size of circuits that they could implement. FPGA logic capacity lags behind semi-custom ASIC technology by an order of magnitude, or more. This meant that a single FPGA-based prototyping system could only handle circuits one-tenth the size of what could be implemented using an ASIC. Incorporating multiple FPGAs on a single hardware platform has been employed to address this issue.

These multi-FPGA systems provided ample amounts of usable logic but it came at a cost. While each of the different FPGAs can provide high-speed and high-bandwidth

intra-FPGA connections, the inter-FPGA board level connections have much less bandwidth and much greater latency. Once systems grew to span multiple circuit boards the inter-board bandwidth only exaggerated this problem. This means that an engineer must either design their circuits to run at the lower speed, dictated by the inter-FPGA connections, or must segment their designs in such a way to account for the heterogeneous nature of the development system. Either way this inter-FPGA bandwidth is another limitation provided by multi-FPGA development systems.

Another factor that can limit the usability of such prototyping systems is peripherals and interfaces provided on the hardware platform. A development platform is typically designed towards a specific market with specific interface needs. Some examples of such markets, and the interfaces they require, are listed below:

- Computer Vision
  - Computer vision applications require one, or possibly more, video input peripherals.
- Computer Networking
  - Networking devices require peripherals that interface with the physical network media.
- Embedded Application
  - Since embedded systems often employ standard bus interfaces, such as PCI, or I2C, development systems must include the same.

It is the inclusion, or absence, of these interfaces that limits the types of systems that a prototyping system can implement. In addition to these specific interfaces, almost all development boards contain some amount of on-board memory. Once again the amount of memory and its speed is also a limitation on what a prototyping system can implement.

The combination of all of these different factors: the amount of usable digital logic, the inter-FPGA bandwidth, the available peripherals, the memory bandwidth and the memory depth, dictate the limitations on the types of systems that can be implemented using modern FPGA-based prototyping systems. The goal of this research

is to develop a state-of-the-art multi-FPGA development system that maximizes usability of the system by enhancing many of these key limitations.

1.3 KEY OBJECTIVES

The prototyping system described in this thesis, the Transmogrifier-4 (or TM-4 for short), is the fourth generation development system designed at the University of Toronto. The previous three systems [1,2,3], the details of which can be found in chapter 2, focused on providing sufficient amounts of programmable digital logic, combined with useful peripherals and interfaces. The most recent version of these systems, the TM-3, provided both video-in and video-out, links to a PC-class host computer, and several megabytes of on-board SRAM.

The TM-3 system has been successfully used to implement a number of different applications including stereo vision [4], ray tracing [5], and a protein identification system [6]. While each project was successful, they also provided new insight into what could be improved in the Transmogrifier design.

Figure 1: Stereo Vision Input [4]

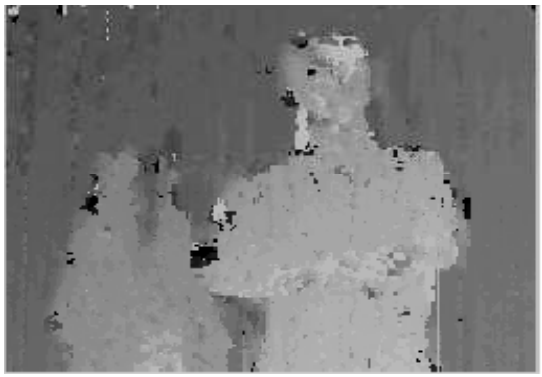


Figure 2: Stereo Vision Output [4]

The stereo vision application [4] was a computation-heavy design that was segmented between several different FPGAs. The purpose of the application was to take video from two cameras and determine the distance to each object in the video. Figure 1 shows a sample input from one of the stereo cameras. Figure 2 shows the resulting depth map as generated by the stereo vision hardware. The lighter colours indicate pixels that are closer to the camera and the darker colours indicate pixels that are further away. It

was found that the amount of available bandwidth between FPGAs was such that the inter-FPGA buses had to be carefully designed to transfer sufficient amounts of data. Without that constraint, this effort could have been better spent on designing the core stereo vision algorithms and implementations instead.

The ray-tracing project [5] used the TM-3 to render 3D images from data describing a virtual scene. This data was generated by a host PC and sent to the TM-3 where it would render the images. A sample image is shown in Figure 3. It was found that the amount of bandwidth available on this host-to-Transmogrifier link was a serious bottleneck to performance. Although the TM-3 could render a 3D scene in well under a second, it would take the host link several seconds just to transfer the scene data and completed image out of memory.

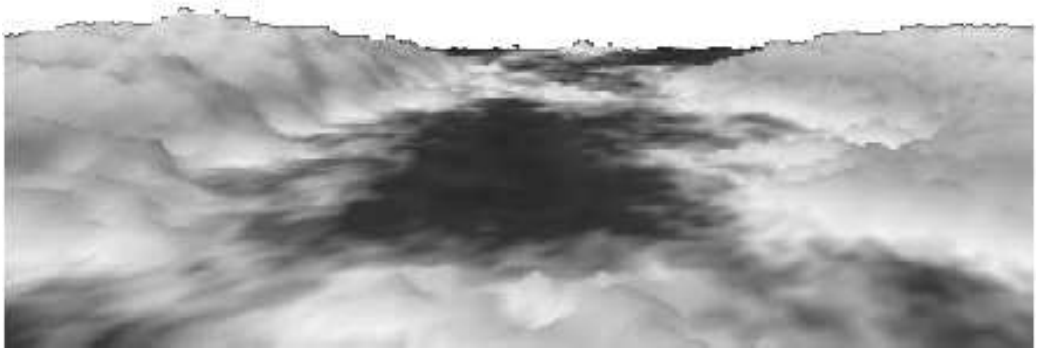


Figure 3: Scene From The TM-4 Ray Tracer

The final application, the protein identification system [6], implemented a bioinformatics algorithm that identified proteins through the use of the human genome. The desired system would take data from a mass spectrometer sample and then perform a linear search through the gigabytes of human genome data to identify the protein sample. A prototype was implemented on the TM-3 that showed that searching the genome this way is feasible but would require a development system with more memory.

These three systems illustrate the limitations inherent in any prototyping systems, those of inter-FPGA bandwidth, host computer bandwidth, and memory bandwidth and depth. It is the goal of the TM-4 project to design a development platform that

maximizes the inter-FPGA, host computer, and memory bandwidth as well as the available memory depth.

1.4 ORGANIZATION

The remaining chapters of this thesis are organized as follows. Chapter 2 will provide some background information on past and current FPGA-based development systems along with a brief primer on some of the key technologies being used in the design of the TM-4. Chapter 3 will describe the design methodology, and the circuit design itself of the TM-4. Chapter 4 will describe the design of the printed circuit board for the TM-4. Chapter 5 will measure the performance of each of the four key goals, and Chapter 6 will conclude.

2 BACKGROUND

2.1 INTRODUCTION

The first half of this chapter will examine the history of FPGA-based development systems to provide a context for the research presented here. First, a representative set of systems will be described and their functionality divided into a taxonomy of the entire space of FPGA-based development systems. Next, the history of the previous Transmogrifier systems will be described and finally a description of recent, commercially-available, development system will be given.

The second half of the chapter will provide a brief description of some of the technologies that are incorporated into the presented work. These include a discussion of source-synchronous clocking, the Altera Stratix FPGA, dual data rate SDRAM, and low voltage differential signalling, or LVDS, high-speed serial communication.

2.2 A TAXONOMY OF FPGA DEVELOPMENT SYSTEMS

Over the history of FPGAs there have been many of different development systems that are based on FPGAs. These range from simple single-FPGA systems to huge multi-board, multi-FPGA systems with the software tool flows to match. Of these different development systems there are a number of key characteristics that differentiate them from each other. This section will present these characteristics and use them to categorize a sample of development systems.

The first categorization is the number of FPGAs comprising the development system. The simplest development system is composed of only one FPGA. These systems are typically designed to allow very simple development to be done on a given generation of FPGA. As such, single FPGA-based development systems will not be examined in further detail. The following subsections will provide classifications that are

applicable to multi-FPGA-based systems. Although some of these classifications apply equally to both single and multi-FPGA systems, only multi-FPGA examples will be provided.

2.2.1 PURPOSE: PROTOTYPING/VERIFICATION

There are two distinct classes of development systems, those that allow the user to prototype an algorithm or a system and those that only allow verification. The primary difference between the two different classes is that a designer typically uses a prototyping system to design a system that will ultimately end up targeting FPGAs, whereas verification systems are used to validate a design that will target an ASIC. This distinction is important as each type of system has very different needs.

The designs that can be prototyped on a given system are dictated by the architecture of the system itself. For example, if a designer wishes to prototype a video-processing system, the prototyping system must have the proper video interfaces and must be able to operate at the desired speed. If the final design will run at 200Mhz then the prototyping system must also be able to handle this speed. This is directly contrary to a verification system where speed is not a driving factor.

It is typically impossible to prototype an ASIC design running at full speed on an FPGA-based prototyping system. This is because an FPGA's speed lags that of a custom ASIC chip manufactured in the same fabrication process. However, FPGA development systems can still be used in the design of ASICs. There are a number of commercial systems, such as Aptix's system explorer [7], Emulation and Verification Engineering's ZeBu-XL [8], Mentor Graphics' VStationPRO [9], Cadence Palladium II [10] and AMO GmbH's Venux-X Emulator [11], that enable such high-speed designs to be emulated using FPGAs running at a much lower speed. These systems are designed to abstract away the underlying FPGA structure to allow easy verification. This is quite different from a prototyping system in which the structure of the system is very important.

To allow for an easy transition between a prototyping system and a production system, both systems typically have a similar architecture. If the designer is targeting a production system with two FPGAs and a certain amount of memory, the prototyping

system should also have at least two FPGAs and sufficient memory. A designer could then directly target their design to the architecture without the need for the abstraction layer provided in an emulation system. By removing this abstraction layer the prototype system is able to operate at a much higher speed than an emulation system.

2.2.2 MULTI-FPGA INTERCONNECT TOPOLOGY

A design targeting a multi-FPGA-based development system must be split across the various FPGAs. The way this is done is dependent on the interconnection topology between the different FPGAs that the system provides. The topology can either be tuned for a specific class of application or designed to be flexible enough to implement most applications. Figure 4 shows several common topologies used by various development systems.

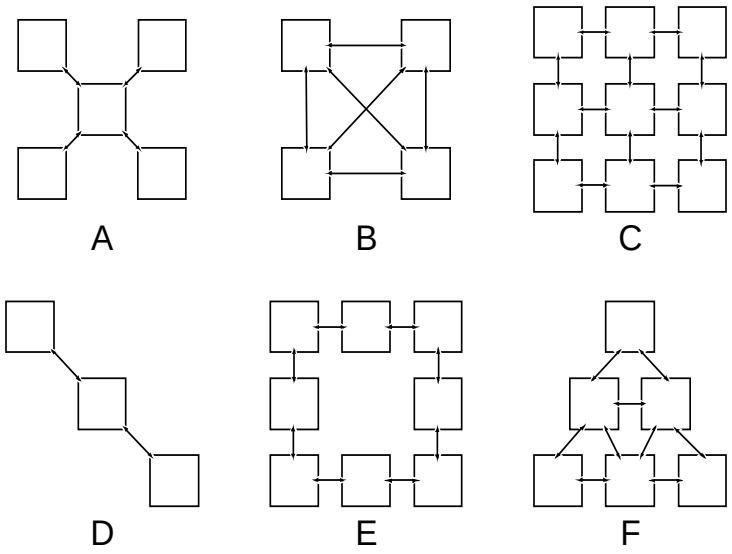


Figure 4: Interconnect Topologies

Topology A consists of a crossbar style interconnection scheme. Each FPGA is connected to a central crossbar. This crossbar allows any FPGA to communicate with any other FPGA. This crossbar might be a specialized crossbar chip, or another FPGA that is used for routing. The crossbar topology is quite common in academic development systems as it allows the topology to be configured to emulate any other style. Crossbars can be found in many development systems including the following [1,

2, 14, 16, 19, 22, 27]. Figure 5 shows how this topology was used in the Splash 2 [27] development system. The Splash 2 architecture consists of 16 FPGAs all connected to a central crossbar with the crossbar's programmable connection controlled by a 17th FPGA.

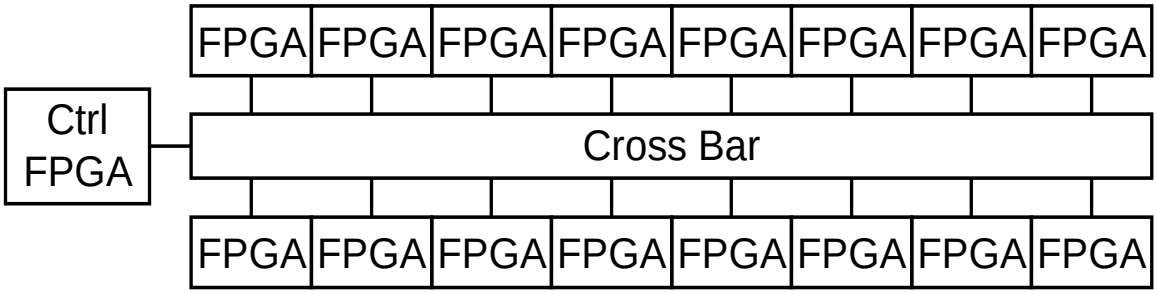


Figure 5: Splash 2 Crossbar Interconnect Architecture

A slight variation of the crossbar style interconnect is to fully connect all the FPGAs to each other, as show in Topology B, instead of using a programmable connection scheme. This scene has the advantage of removing the crossbar latency from interconnects but requires many more wires to achieve the same connectivity. Examples of systems that use this topology are [3, 25]. Since both topologies A and B have each FPGA connected to every other FPGA they are the most flexible topologies. However, this flexibility comes at a cost. As the number of FPGAs increases in a system the number of connections required grows with O(n^2) and with fixed interconnects, the number of connections between any 2 FPGAs is reduced..

Topology C, a 2D or 3D interconnection mesh, is a slightly less flexible connection scheme then the crossbar scheme. This mesh allows a given FPGA to communicate directly with its neighbours. Communication between more distant FPGAs must be relayed through an intermediate node. This topology is good in systems with a large number of FPGAs, due to the fact that the inter-FPGA connection requirements scale linearly as the FPGA count is increased. This comes at the cost of having less total interconnection resources and increased connection delay between distant FPGAs, but as long as applications are designed to localize inter-FPGA communication this topology works well. Examples of systems that use 2D meshes are [20, 22] and systems that use a 3D mesh are [21].

There is a common class of application for which a crossbar or mesh topology is excessive. Many algorithms operate in a pipelined manor. That is there is a data source for which a series of independent algorithms are applied. Topology D, which consists of a linear interconnection scheme between FPGA, can efficiently implement pipelined applications. Data is fed into the system from one end of the chain for each FPGA to process in turn. The result is then output from the last FPGA. This topology can be extended slightly by connecting the first and last nodes of the chain, as shown in Topology E, the ring. A linear topology is used by the Anyboard system [17] and a ring topology is used by the ARMen system [18].

Another class of application, which is common, are those with a series of algorithms that increase in complexity. Topology F, a tree structure, can implement some of these classes of applications. Data is fed into the system at the root node and is processed by nodes downward along the tree. The deeper the data flows down the tree, the more processing power is available. Figure 6 shows the 3 FPGA tree structure used by the Prism II [24] system.

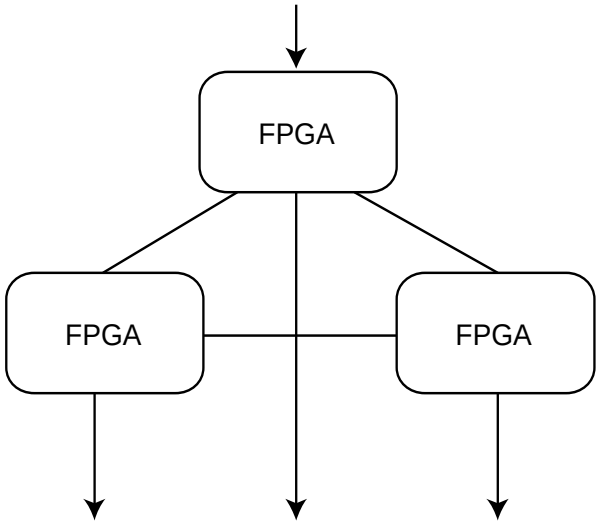


Figure 6: Prism II Tree Interconnect Topology

Systems with very large numbers of FPGA tend to use a combination of the above topologies. Typically a small group of FPGAs are connected tightly using either a crossbar or mesh approach. This allows single algorithms to span across several FPGAs.

These groups of FPGAs are then connected with other groups through either a linear, tree or larger mesh structure.

In addition to these flexible interconnection schemes, there are also a number of development systems that consist of application specific interconnections. For example, the functional memory computer [12] is designed to explore a very specific type of computational paradigm. It consists of a number of FPGAs that are connected using a common memory-mapped bus. Another development system, Ganglion [13], was designed to explore neural network style processing. As such, the interconnections are optimized for this specific requirement.

2.2.3 FPGA INTERCONNECT IMPLEMENTATION

As described in the previous section, there are a number of different possible interconnection topologies. In addition to this, the way these topologies are implemented can also vary between development systems. The types of interconnections typically falls into one of four different categories: fixed, programmable, switched and hybrid.

A fixed interconnect structure is one where the connections between different FPGAs is hardwired into the systems circuit board. A programmable interconnect structure has the various FPGA signals feeding into one, or more, switching chips. These chips provide the ability to connect the various FPGA signals programmably. Typically these chips are either a crossbar or an additional FPGA itself.

The third classification, switched interconnect, is a connection method that allows the interconnections to be changed while the development system is operational. Typically, this is done using packet switching, as commonly used in the networking world. An example of a switched interconnection network can be found in the form of either an FPGA controlled crossbar, as found in Splash 2 [27], or as a packet switched network, as found in the Bee 2 system [14].

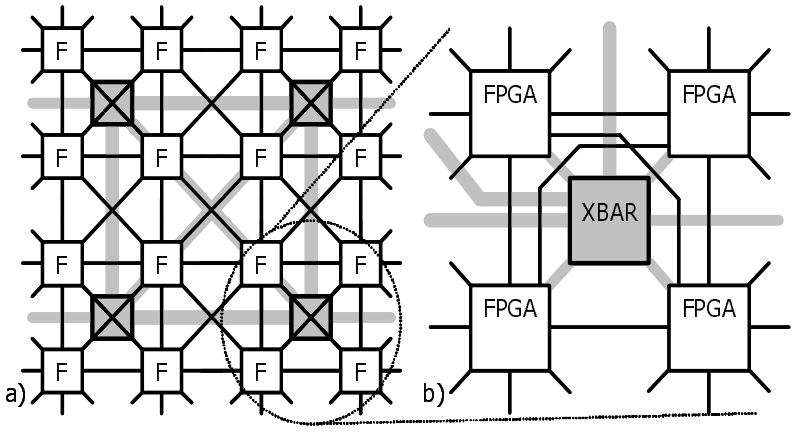


Figure 7: Bee's Interconnection Topology Hierarchy

The final classification, hybrid, combines several of the previous interconnection structures. For example, the Bee system [14], consists of an interconnect hierarchy, shown in Figure 7. The lowest level of the hierarchy consists of a fixed nearest neighbour grid of interconnected FPGAs. This fixed grid provides local connections. The next level in the hierarchy provides global connections through the use of programmable crossbar switches.

2.2.4 EXTERNAL INTERFACE/HOST

A development system does not exist in complete isolation. For a design to perform a useful task, it must have some interface to the outside world. This interface might consist of a specific type of interface, such as video I/O or a network port, or a more general interface, such as a CPU or general-purpose expansion IO. The general-purpose interfaces allow more flexibility, since they could also provide a video stream, but the bandwidth provided is not optimized towards a specific application, as a dedicated video I/O port would be. This means that the available bandwidth on a general interface will ultimately limit the rate, and type of data can that can be provided to the development system.

The bandwidth of general-purpose expansion I/O is limited by the number of FPGA pins dedicated to expansion as well as the maximum data rate each pin can operate at. A host CPU interface's bandwidth is limited by the method used to connect the development system to the host computer. The simplest, and slowest, method is to

provide a parallel or serial port interface as [2] does. This method can provide at most a few megabytes per second of bandwidth due to the low operating performance of both parallel and series ports. A method with much higher bandwidth is to connect the development board directly to a CPU's expansion bus, such as SBUS, SCSI, PCI [15], or VME. This type of connection can provide anywhere from a few dozen megabytes per second to several hundred megabytes per second.

Systems that use an SBUS host computer interconnect include [16, 18, 20, 25, 27], a SCSI interface [28], a PCI interface [3, 12] and a VME interface [13,21].

2.2.5 MEMORY

Many important applications require a large amount of memory. Most modern FPGAs include moderate amounts of on-chip ram, up to 10Mb in state-of-the-art FPGAs, but this amount is usually not sufficient. Most development systems augment this memory by adding additional external memory.

The speed, type, and configuration of the memory can have a substantial effect on the performance of circuits using it. For example, the simplest memory for a circuit to interface with is SRAM. This type of memory does not require refreshes and typically has very low latency. The primary drawback is that the memory capacity is low. Using DRAM can increase this memory capacity and bandwidth, but this comes at a cost. A DRAM interface is much more complex than an SRAM one due to the fact that DRAM must be refreshed. In addition to this the memory latency is also increased. Where an SRAM's latency could be a low as 2 cycles, a DRAM's latency is often closer to 10, or more, cycles.

Which type of memory is best depends upon the type of application using it. If an application is very latency-dependent, and only requires a small amount of memory, then SRAM is ideal. If an application requires high bandwidth or a large amount of memory then DRAM is the best choice.

In the past development systems have usually used SRAM, as it is easier to interface with. The few systems that do incorporate DRAM, such as Enable++[22] and

RPM [26], do so to provide more memory then SRAM can provide. However, both of these systems also include a small amount of SRAM as well.

2.2.6 SUMMARY

A development system can be defined by a number of different characteristics, as discussed in the previous section. These include the system's purpose, the interconnection topology, the interconnection implementation, the external interfaces, and the amount and type of system ram. Table 1 summarizes the last four characteristics for a number of historical and modern prototyping systems.

	FPGA Count And Type	**Memory**	**Host Interface**	**Interconnect**
ACME [16]	14 Xilinx XC4010	28K SRAM	SBUS	Programmable
Anyboard [17]	5 Xilinx 3042	384K SRAM	ISA	Fixed
ARMen [18]	8 Xilinx 3090	512K SRAM	SBUS	Ring
Bee [14]	20 Virtex 2000E	16MB SRAM	Ethernet	Mesh Hierarchy
Borg [19]	4 Xilinx 30XX	2K SRAM	ISA	Programmable
Chameleon [20]	3 Xilinx 4010	1.25M	SBUS	Fixed Mesh
DFFC [21]	512 Custom FPOA	8M	VME	3D Grid
Enable++ [22]	~50 Xilinx 40XX	12M SRAM 384M DRAM	Custom	Programmable
FMC [12]	11 Xilinx 40XX	1M SRAM	PCI	Bus
Ganglion [13]	24 Xilinx 3090	None	VME	Fixed Custom
Marc 1	25 Xilinx 4005	6M SRAM	SBUS	Programmable
Morrph-ISA	6 Xilinx 40XX			Fixed Mesh
Perle-0	25 Xilinx 3020	500K SRAM	VME	Fixed Mesh
Prism [23]	4 Xilinx 3090	None	16bit Bus	None
Prism II [24]	3 Xilinx 4010	1.5M SRAM	64bit Bus	Tree
Race [25]	4 Xilinx XC4013	512K SRAM	SBUS	Fixed Mesh

	FPGA Count And Type	Memory	Host Interface	Interconnect
RPM [26]	63 Xilinx XC4013	90M SRAM 864M DRAM	SCSI	Fixed
Splash 2 [27]	16 Xilinx 4010	8M SRAM	SBUS	Crossbar
Spyder	5 Xilinx 4003	128K SRAM	SBUS/VME	Fixed
Teramac [28]	1728 Custom FPGA	512M SRAM	SCSI	Crossbar
TM-1 [1]	4 Xilinx 4010	144K SRAM	SUN	Programmable
TM-2 [2]	32 Altera 10K100	128M SRAM	Parallel	Crossbar
TM-3 [3]	4 Virtex 2000E	6M SRAM	PCI	Fixed Mesh

Table 1: FPGA Development System Characteristics

2.3 THE TRANSMOGRIFIER PROJECT

While the global goal of the Transmogrifier Project has always been to create an easy-to-use development platform for researching algorithms and implementations, the goal of each specific Transmogrifier, or TM for short, has varied.

The goal of the TM-1 [1] was to provide an initial research platform to serve as a starting point for future development. Next came the TM-2 [2] with the goal of designing a system that could handle very large circuits. This was achieved through the use of a large number of FPGAs in a scaleable design. By the time the TM-3 [3] was designed, FPGA technology had improved sufficiently so that a single FPGA could now do the job of the entire TM-2. This led to a new goal, a goal of performance, instead of just size. The TM-3 was thus designed to enable circuits to operate at a high clock rate, up to 100MHz.

The following three subsections will provide an overview of the first three Transmogrifiers.

2.3.1 TRANSMOGRIFIER-1

The Transmogrifier-1 was designed at the University of Toronto in 1991. The TM-1 consisted of four Xilinx 4010 FPGAs connected using Aptix programmable

crossbars with an Aptix programmable board. These four FPGAs provided 3200 4 input lookup tables, or LUTs, for use as programmable digital logic. Each of the four FPGAs was connected to a 32Kx9 SRAM.

2.3.2 TRANSMOGRIFIER-2

The Transmogrifier-2 was designed at the University of Toronto in 1996. The TM-2 incorporated a scalable multi-board design for which systems where built consisting of two, four and sixteen boards. Each board on the TM-2 consists of two Altera 10K100 FPGAs, 8MB of SRAM, and a programmable crossbar. In the largest configuration there are 32 FPGAs providing a total of 160,000 LUTs, and 128MB of ram. The TM-2 also introduced a parallel port interface to a Sun host computer. This interface allowed programming of the FPGAs as well as communication with the circuit under test. Communication was facilitated through the use of parameterizable bus hardware interfaces, combined with a software package on the Sun computer. The TM-2 was used successfully for a number of applications including face detection [29], and procedural texture mapping [30].

2.3.3 TRANSMOGRIFIER-3

The Transmogrifier-3 abandons the multi-board approach due to the effort involved in effectively building and using multi-board prototyping systems. The TM-3 system consists of four Xilinx Virtex 2000E FPGAs each with 2MB of SRAM. This provides a total of 150,000 LUTs and 8MBs of SRAM. The interface with the host computer has also been improved by using a direct SBUS or PCI link instead of the parallel port connection provided by the TM-2. This link is once again used to program and communicate with the development FPGAs.

The TM-3 was used for a number of different applications including stereovision [4], ray-tracing [5] and a protein identification proof of concept system [6].

2.4 COMMERCIAL DEVELOPMENT SYSTEMS

	Purpose	FPGA Count	Memory	Host Interface
HAPS [31]	Prototype	4 Virtex II 8000 per board (Stackable)	1GB DRAM / FPGA 0.5GB SRAM / FPGA	PCI-X 133Mhz
Venux-X [9]	Emulation	6 Virtex II 6000	< 1 G SRAM	Custom
ZeBu-XL [8]	Emulation	64 Virtex II 8000	< 1 G DRAM	Custom
PROCStar II [32]	Prototype	4 Stratix S80	2 GB DDR II	PCI 66 Mhz
DN6000k10 [33]	Emulation	9 Virtex II Pro 100	1.5 GB DDR	USB
System Explorer [7]	Emulation	8 Virtex 2000E		
Wildstar II PRO [34]	Prototype	3 Virtex II Pro	528MB DDR	VME
Pro 3100 [35]	Prototype	4 Virtex II 8000	512MB DDR	PCI 66 Mhz

Table 2: Available Commercial Development Systems

There are a number of commercially available development systems that have a comparable level of functionality to the Transmogrifier-4 (the system described in this dissertation). To understand what sets the TM-4 apart from these systems, this section will summarize the capabilities of competitive commercial systems.

Table 1 summarizes the available commercial development systems with similar amounts of FPGA development resources. The first system, the HAPS, is a modular system that allows scalability. The base configuration consists of 4 Virtex II 8000 FPGAs and up to 4 GB of DDR SDRAM. However the memory subsystem does not use RAM modules and instead requires 6 individual expansion boards with discrete RAM chips. This solution is more expensive and slower then using standard RAM modules but fits with the systems goal of being modular.

The next two systems, Venux-X and ZeBU-XL, are both very large ASIC emulation systems. Although the amount of available logic will be greater than that of the TM-4 the systems are only designed to run in the several Mhz range. The Gidel PROCStar II board consists of 4 Stratix S80s and a 66MHz PCI interface, the same as the TM-4. What the PROCStar lacks is memory bandwidth. There are only two channels of DDR II available.

The next two systems, the Dini Group DN6000K10 and the Aptix System Explorer are also only emulation systems. They are also designed to only run in the few megahertz speed range. The remaining development system, the Pro 3100, is also similar to the design of the TM-4. It consists of 4 Virtex II 8000 chips and a PCI 66MHz host computer connection. However once again lacks both memory bandwidth and memory depth.

2.5 BACKGROUND TECHNOLOGIES

Many of the different decisions involved in the design of the TM-4 were the result of various technical requirements or limitations. The following subsections will provide some background technical information relating to FPGA technology, external memory and high-speed inter-chip signalling in order to provide suitable context for the design presented in Chapter 3.

2.5.1 ALTERA STRATIX FPGA

The largest chip in the Altera Stratix line of FPGAs, the EP1S80, consists of several different types of programmable logic blocks. Its core programmable functionality is provided through 79,040 four-input programmable lookup tables, or LUTs, and is supplemented with other blocks such as multipliers, on-chip memories, phase locked loops, and specialized I/O hardware. Of particular relevance to the TM-4 project is the structure of the Stratix FPGA.

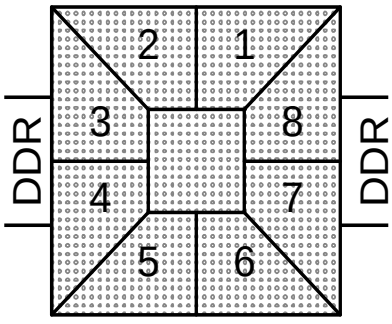


Figure 8: Stratix I/O Banks

The Stratix FPGA pins are grouped into eight different banks, as shown in Figure 8. The I/O pins in each bank are connected to a common I/O voltage and reference. This restriction limits what types of I/O standards can be operated in any given bank. For example, it is not possible to have both a 2.5v and 3.3v output I/O standard operating in the same bank since the bank must have a common I/O voltage. A similar restriction exists for reference voltage as well. In addition to voltage restrictions, the Stratix FPGA's I/O banks are also limited in which I/O standards they can implement.

Only I/O banks 1, 2, 5, and 6 can support the LVDS differential standard. These banks provide dedicated serialization/deserialization hardware that help enable LVDS communication at up to 840Mbps. The remaining banks 3, 4, 7, and 8 only support single ended output standards. However, these banks do contain dedicated hardware that allow for easy DDR SDRAM interfaces. These banks incorporate a delay locked loop for use in properly aligning the clock and data received from a DDR SDRAM memory.

As will be discussed in Chapter 3 these bank restrictions are very significant when it comes to routing the TM-4's circuit board.

2.5.2 SOURCE-SYNCHRONOUS CLOCKING

Transmitting data at a high rate is a very difficult problem. As data rates increase the effect of skew and IC fabric and circuit board process variation become very signification. Skew directly reduces system-timing margins, and process variations increase the uncertainty of inter-chip delays. These two factors make it uneconomical to use a simple global clocking method to transmit data. To compensate for this problem a clocking method known as source-synchronous clocking is often used.

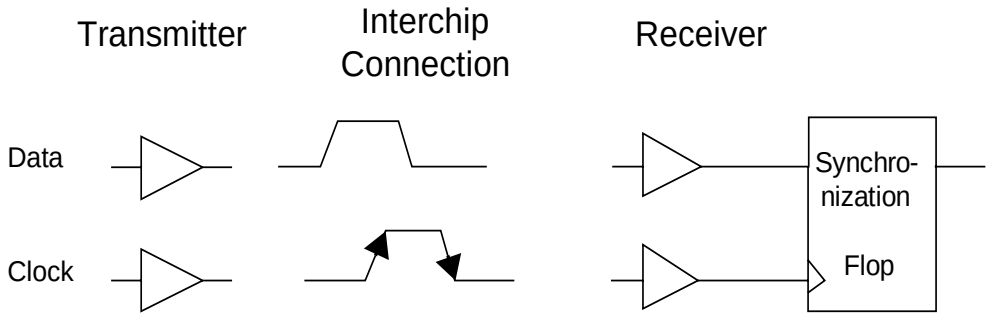


Figure 9: Source-synchronous Clocking

Source-synchronous clocking is designed to eliminate most of the effects of process variation to enable higher-speed signalling. Figure 9 illustrates how this works. A transmitter drives both a data and clock signal. The receiver can then uses this clock signal to synchronize the provided data. Provided that the data signal meets the setup and hold time requirements relative to the clock signal the flip-flop will successfully capture the data. It is easy to see that if sufficient skew is introduced between the clock and data signals that synchronization will fail.

To limit the amount of skew introduced, the two signals traces are designed in such away as to minimize the sources of skew. For example, the transmitter would typically place the data and clock pins physically very close together on the die and use the same branch of the clock distribution tree. The goal in this design is to remove process-variation-induced clock skew. Similarly, the physical circuit board inter-chip connections would be routed similar to each other in order to reduce skew. Finally, the receiver chip would try to match the delays between the internal clock distribution network and the input time of the data signal. If all these sources of skew are sufficiently controlled, then the data can be successfully captured in the receiving device. However, the data must still be transferred to the receiver's clock domain.

There are two different ways to handle this problem depending on the situation. In the most difficult situation, where the phase of the data clock is completely unrelated to the internal clock, it is necessary to treat the output of the synchronization register as asynchronous. The means that all metastability issues [36, 37, 38] must be addressed. In the easier case, where there is some relationship between the data and internal clocks, the resynchronization can be handled by buffering the received data while still in the data

clock domain. Periodically this data can then be transferred into the local clock domain at a much lower rate.

2.5.3 DDR SDRAM

DDR SDRAM, or dual data-rate synchronous dynamic random access memory, is a dynamic RAM standard that incorporates double data rate signalling, with source-synchronous clocking, to achieve high data rates. Double data rate means that data is transferred on both the rising and falling edges of the clock. The DDR SDRAM standard supports speeds of up to 200Mhz. To meet these speeds, while still being inexpensive to implement, the standard uses source-synchronous data strobes for transmitting data. A number of data bits, usually 4 or 8, have a strobe signal that is routed along with them. This strobe is then used as a source-synchronous clock as described in the previous section. The only difference between DDR SDRAM, and the example provided in the previous section, is that the strobes and data signals driven by the memory have coincident edges. To meet setup and hold requirements the strobe must be delayed by the receiver. This can be accomplished on the Stratix FPGA through the use of a specialized delay-locked loop.

Like many DRAMs before it, DDR SDRAM memory is organized into a hierarchy. Figure 10 shows a subset of the hardware contained in a DDR SDRAM chip. Memory is stored in four different memory arrays, or banks. Each array consists of a fixed number of rows and columns. In order to read or write a memory element, it is necessary to first have the internal DDR SDRAM controller “open” the row. This means that the controller will read an entire row of data from the 2D memory array into a buffer. This buffer can then be read or written into by selecting which column to access. Upon completion the row must be “closed”. This means that the buffer data is written back to the 2D array.

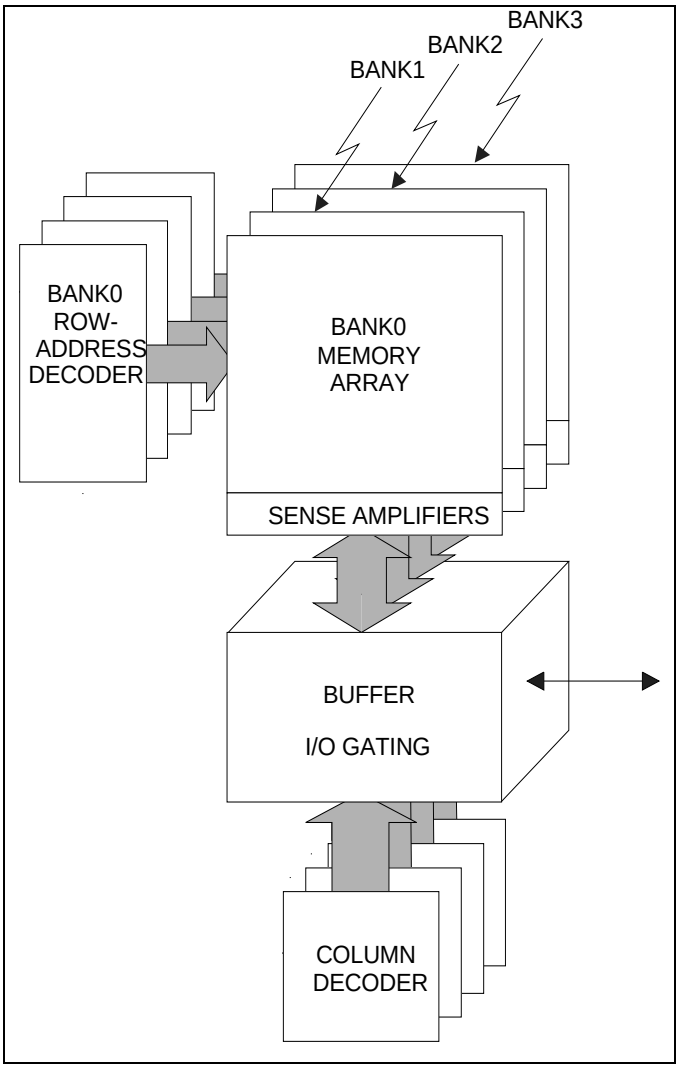


Figure 10: DDR SDRAM Memory Organization

This process of opening and closing rows takes a significant number of cycles. Although the exact time varies between memory modules, typically each process takes around 3 cycles. In order to mask some of this time, DDR SDRAM allows one bank to be opening or closing a row while another bank is performing a read or a write. Through the use of clever access patterns, DDR SDRAM can sustain burst transfers very near the theoretical maximum of two words per clock cycle.

2.5.4 LVDS HIGH-SPEED SERIAL COMMUNICATION

Electrical signalling standards have a strong effect on the speed performance of a communication link. One such standard is known as *low voltage differential signalling*,

or LVDS for short. This standard belongs to a family of standards known as differential standards, which use two wires to transmit data. One wire carries a data signal and the other carries the data signal's complement. This is in contrast to more conventional signalling that uses only one wire called a single-ended standard. It should be noted that both differential and single-ended standards have an implied return current, or ground, path.

LVDS signalling has two primary advantages over conventional single-ended standards, and one major drawback. The first advantage is that LVDS involves very little voltage swing. Whereas TTL or CMOS might have 2.5V or 3.3V voltage swings between high and low, LVDS signal swing is between 240mV and 450mV. This decreased voltage swing means that there will be less inter-trace coupling between different inter-chip LVDS signals, thereby helping reduce signal integrity issue. The second advantage arises from the fact that LVDS is a differential standard.

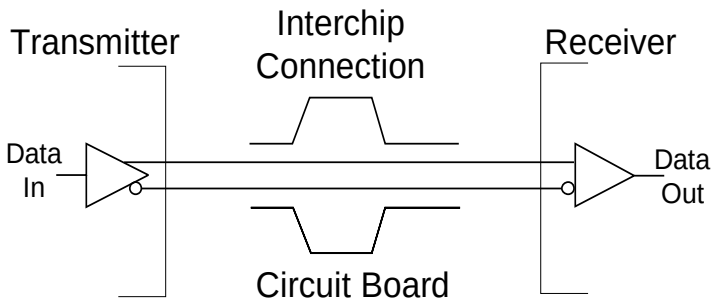


Figure 11: LVDS Communication Channel

A differential topology consists of three different components: the transmitter, the interconnection, and the receiver. Figure 11 illustrates these components. Each of these components benefit from the differential nature of LVDS.

The benefit to the transmitter is in the form of reducing the inductance caused voltage drops. When a conventional single ended transmitter switches between output states there is a corresponding supply current step. This step will cause a voltage drop due to the inductance between the driver and the power source. At low speeds this problem can be reduced by capacitors on the circuit board. At higher-speeds it is the responsibility of on-chip capacitance to prevent excessive voltage drops, but at

sufficiently high speeds even on-chip capacitors cannot help with this problem. Differential signalling reduces the effect of inductive induced voltage drop by trying to eliminate voltage steps. When one wire of the differential signal is switching in a given direction, the other wire is switching in the opposite direction. Ideally, these two voltage swings will cancel each other out and eliminate the effect of the power supply network's inductance completely.

The interconnection signal integrity is helped by the differential nature of LVDS as well. Ideally the two wires are switching in opposite directions at the exact same time. This means that the return current of one wire will flow through the other wire, as opposed to a ground plane or neighbouring trace. This is an ideal situation as it helps reduce the signal integrity effects of holes and slots in the ground plane. Unfortunately, in practice the LVDS wires are not perfectly matched and the transmitters' drivers will have some skew. This will lead to the non-ideal situation where some current will still flow through the ground plane. However the situation is still better than a single ended wire.

The final piece of the communication channel is the receiver. The receiver benefits from differential signalling in two ways. The first is that differential signalling makes a small voltage swing seem twice as large. That is, even though one wire is switching only 400mV, the other wire is switching 400mV in the other direction. When put through a differential amplifier the voltage swing is effectively doubled to 800mV. Another advantage is that common mode noise between the two signals will cancel each other out. Ideally both wires in a differential pair will run very close together on a circuit board, the idea being that any noise will couple into both wires. This noise will then be filtered out at the receivers' differential amplifier. Once again reality is not as friendly as the ideal case in that differential pairs tend to be only loosely coupled. This means that noise will not be equally coupled into each wire but once again this situation is still better than a single ended wire.

The major drawback of LVDS is that it requires twice the number of wires of a single ended standard. To achieve the same bandwidth it is necessary that an LVDS pair run at twice the data rate as a single ended standard. In the case of a long cable connection this is almost always possible; the common noise rejection can often easily

out perform single ended standards by a factor of 2 or more. However, in the case of short onboard connections the case is not as strong. LVDS still wins out performance wise but the margin is much smaller.

2.6 SUMMARY

This chapter described a set of both commercial and academic FPGA-based development systems, in terms of taxonomy of the entire space of development systems. A description of the previous generations of Transmogrifier systems was then presented followed by a technical introduction to some of the key technologies employed in the TM-4.

The next chapter will describe the design of the TM-4 by examining the design methodology employed in creating the TM-4.

3 DESIGN

3.1 INTRODUCTION

This chapter presents the design of the Transmogrifier-4 by examining the design methodology used in creating the TM-4 and then describing important parts of each step of the design. These steps include the identification of the requirements for the TM-4, the hardware design of the circuitry and the verification of the circuitry. An additional step, the design of the circuit board, is examined in detail in the next chapter.

The requirement identification section, Section 3.2, identifies the high level requirements of the TM-4 by examining past experiences and predicting future needs. These requirements are then used to create a complete hardware design for the TM-4, described in Section 3.3. Prior to actually fabricating the TM-4 hardware a number of different verification test were performed. These tests are described in Section 3.4.

3.2 REQUIREMENT IDENTIFICATION

The first step in designing the TM-4 was the identification of a set of key desired characteristics for it. These requirements were identified from three different sources. The first source was the previous Transmogrifier, the TM-3 [3]. Many of the same requirements that drove its design were still applicable to the TM-4. Next, experience gained through the use of the TM-3, in creating applications was used to provide new requirements and finally a number of forward-looking requirements were identified by anticipating the future application space of the TM-4.

The following three subsections will examine each of these different sources of design requirements and one additional subsection will summarize the final design requirements.

3.2.1 PAST TRANSMOGRIFIER REQUIREMENTS

Many of the decisions made in the design of the previous Transmogrifier, the TM-3, are still relevant today. In particular, experience using the TM-3 has shown that the selection of the number of FPGAs, the FPGA interconnection topology, and the choice of the external interface was well suited for the TM-3 and suggests that similar selections would also suit the TM-4. The remainder of this section will present the selections made in the design of the TM-3 and examine the reasoning behind them.

The choice of the number of FPGAs that a development system contains is a trade-off between complexity and capacity. An arbitrary amount of capacity can be added to a system by increasing the number of FPGAs that the system contains. However, this comes at a cost of complexity, as users of the system must now partition designs over a large number of FPGAs. The design of the TM-3 selected a trade-off point of having four FPGAs. This was found to provide a large amount of logic capacity while still being usable by designers. The selection of only four FPGAs also simplified the interconnection topology selection.

Since the number of FPGAs was limited to only four, it became feasible to fully interconnect the FPGAs to each other, similar to Topology B in Section 2.2.2. This topology allowed for the lowest latency interconnections, as there were no programmable elements to introduce delay.

The final relevant choice made in the design of the TM-3 was the selection of an external interface. The purpose of an external interface on a prototyping system is to provide a mechanism for communicating with the circuit operating within the system itself. The mechanism employed on the TM-3 was a simple bus protocol between the four FPGAs and a fifth "housekeeping" chip. This "housekeeping" chip provided a bridge between the four FPGAs and a host computer via a PCI bus. Experience has shown that this simple bus, combined with parameterizable soft IP modules, provides a very easy-to-use communication channel for designers to use.

Since past experience has shown the basic organization structure of the TM-3 to be reasonably good the same structure was employed in the TM-4. The TM-3 provided the basic requirements of having four of the largest FPGAs, each that is fully

interconnected to each other, and each that is connected to a host computer via a "housekeeping" bridge.

3.2.2 PAST APPLICATION DRIVEN REQUIREMENTS

The previous generation of Transmogrifiers, the TM-3, was used to implement a variety of applications including: ray tracing [5], protein identification [6], stereo vision [4], and a molecular dynamic simulator [39]. The process of designing these applications brought to light a number of different shortcomings in the architecture of the TM-3. In particular it was found that the TM-3 lacked host computer bandwidth, memory bandwidth, inter-chip bandwidth and memory depth. To better understand how these shortcomings were found, three of the applications implemented on the TM-3, ray tracing, protein identification, and stereo vision will be examined in more detail.

3.2.2.1 RAY TRACING ON THE TM-3

Ray tracing is a method of rendering 2D images of a virtual 3D scene. The algorithm renders a 2D projection of a 3D scene by approximating the way that light rays propagate around the scene and end up hitting the viewer's eye. The light ray propagation model involves "tracing" the path that light rays travel back from the viewpoint, through the projection point and in to the scene. Figure 12 shows an example of how this works.

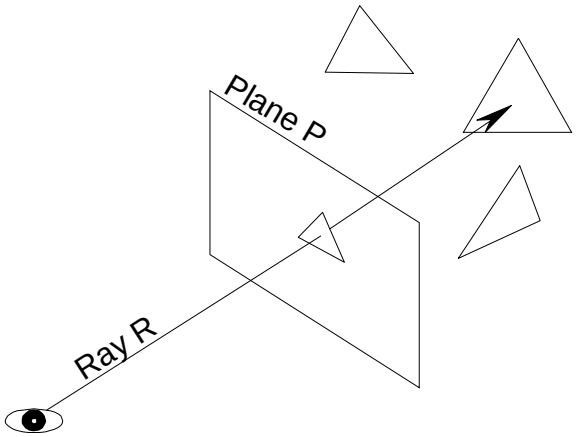


Figure 12: A Simple Ray Tracing Example

The three triangles in the upper right represent the virtual 3D scene. The eye in the lower left represents where the 2D projection should be viewed from. The plane, in the middle, is what the 3D scene should be projected on. The ray tracing algorithm first generates a ray from the eye point through each pixel in the projection plane. These rays are then used to determine what objects are visible and which object should be projected onto a given pixel in the plane P. Intersecting the rays with the 3D scene, a computationally intensive procedure makes this determination. It is this process that was accelerated using the TM-3.

The ray tracing implementation [5] was limited several ways by the architecture of the TM-3. First, the available memory on the TM-3 was limited to only 6 megabytes. This allowed only relatively small 3D scenes to be rendered on the TM-3. The second limitation was memory bandwidth. The TM-3's memory subsystem was built to run at 50Mhz. At this clock speed the memory could not provide 3D data as fast as the hardware could process it. This turned out to be the performance-limiting factor in the design of the hardware ray tracer. The final limitation of the TM-3 was the amount of host computer bandwidth. Both the dataset, which represents the 3D scene, and the resulting rendered image needed to be transferred between the TM-3 and its host computer. However, the available bandwidth, less the 2MB/s, meant that the TM-3 could process data much faster than it could communicate its results.

Experience from the ray tracing application suggested that the TM-4 should have more memory, more memory bandwidth, and more host computer bandwidth. To address the question of exactly how much more of each, a more in-depth look at how a 3D scene can be stored is necessary.

In its simplest form a 3D scene consists of a set of triangles that are arbitrarily positioned in three-dimensional space. Each vertex of the triangle is represented by three numbers defining its position in space. If we assume that 32-bit numbers are used then each triangle will require 36 bytes to store. For a scene of 1,000,000 triangles the storage requirements will be 36MB. If a more complicated 3D scene is used, one that includes texture maps, and other such data, the memory requirement could easily exceed 100MB or more.

The issue of how much memory bandwidth is difficult to address. Since ray tracing can be parallized extensively, it is possible to consume any available memory bandwidth, with the only limitation being logic area. The remaining issue of host bandwidth is also difficult to address as this link is utilized to load scene data. During this time the system is idle and the host link is the bottleneck. Ideally, the bandwidth on this link should be as high as possible to limit the performance impact.

The experience from this application, combined with the simple calculations above, suggest that the TM-4 must have at least 100MB of memory and as much memory bandwidth and host computer bandwidth as feasible.

3.2.2.2 PROTEIN IDENTIFICATION ON THE TM-3

An active area of research in proteomics involves the identification of biological proteins contained in a physical sample. The current approach attempts to identify the molecular make up of the proteins using a device known as a mass spectrometer. This device can take a protein, break it up into small pieces and identify the molecular make up of these small pieces. It is then necessary to assemble these "fragments" into a completed protein. One approach to assembling the fragments involves searching the human genome [6].

The human genome contains a description of every possible protein, and as such can be used to reassemble the protein fragments previously obtained. The algorithm to accomplish this involves searching the entire human genome dataset, several gigabytes of data, and matching the fragments to certain proteins. Successive fragment searches each reduce the set of possible protein matches until only one protein is left. This protein should be the same as the protein in the physical sample.

A prototype created on the TM-3 [6] showed that the algorithm could be easily parallized to consume all available memory bandwidth. In addition it was found that a large amount of memory was also required to store genomic data. It takes around 1 gigabyte of data to store the 3.3 billion base pairs that make up the human genome. In addition there is evidence that it might be necessary to search several different genome

datasets at the same time. This would push the amount of RAM required to between 2-4GB.

The experience from this application suggests that the TM-4 should have between 2-4GB of RAM and as much bandwidth as feasible.

3.2.2.3 STEREO VISION ON THE TM-3

One of the fundamental problems facing computer vision is the problem of extracting depth information from an image or images of a scene. Stereo vision is one approach to this problem that works by mimicking the way human vision works.

The stereo vision approach uses two cameras that are aligned side-by-side. Each of the cameras sees a slightly different version of the scene and these differences can be used to extract depth information. By utilizing simple geometric relationships between corresponding objects in each image, depth can be calculated. The hard part of the problem is identifying the matching points between each image. One solution to the matching problem is to perform a large number of correlations between the pixels in each image. This approach works well but is very computationally intensive.

The TM-3 was used to accelerate the stereo vision computation to the point where it could operate in real time [4]. However, the logic area requirements necessary to meet real time performance were very high. The implementation of the stereo vision algorithm needed to be spread across all four FPGAs of the TM-3. It was found that the lack of communication bandwidth between each FPGA made partitioning the design difficult but in the end a functional stereo vision system was created.

The experience from this application suggests that the TM-4 should have as much inter-FPGA bandwidth as possible in order to simplify the problem of partitioning designs.

3.2.3 ANTICIPATED APPLICATION SPACE REQUIREMENTS

The TM-4 was designed with several different future application spaces in mind, each space with its own requirements. These spaces include stereo vision [4], genome [1]

based algorithms, reconfigurable computing, and others. In order to meet the demands of these applications several additional design requirements were necessary.

Stereo vision applications require two video sources as input and some method of output. This resulted in the requirement that the TM-4 have two analog video-in channels, two digital IEEE-1394 channels (to provide alternative video input and output channels through a standard digital interface), and a VGA video-out channel. Reconfigurable computing applications might require the prototyping system to be rapidly reprogrammable. This added the requirement of having the development FPGAs configured using the fastest method possible.

3.2.4 THE TM-4 DESIGN REQUIREMENTS

The following list summarizes the different design requirements of the TM-4. It includes all the requirements identified in the previous three sections as well as two additional requirements. The requirement that the TM-4 be designed to minimize the risk of a design error causing a complete system failure, and the requirement that the TM-4 fit into a standard PC case.

The first requirement is important since the TM-4 is a piece of physical hardware it cannot be easily changed after it has been fabricated. If a critical mistake is made in the design process the entire system will be useless. To reduce the risk of a complete failure of the TM-4, any system that is critical to functionality must have a simple redundant backup. For example the power supply, the FPGA programming subsystem, and the PCI interface all require a backup contingeny plan.

The second requirement, that the TM-4 fit into a standard PC case, is important as this means that the TM-4 can be easily implemented as a self-contained system while using off-the-shelf parts. The self-contained system will allow for easy portability and easy compatibility with the host computer.

When all of the requirements are tabulated, the following list of requirements results:

1. Logic capacity
    a. Four of the largest available FPGAs

2. Interconnect Topology
    a. Fixed point-to-point
    b. As much bandwidth as feasible
3. Memory
    a. 4GB or more
    b. As much bandwidth as feasible
4. External Interfaces
    a. 2 analog video in channels
    b. 2 digital video channels (IEEE-1394)
    c. VGA video out DAC
5. Host Computer Interface
    a. As much bandwidth as possible
    b. Simple to use for designers
6. Miscellaneous
    a. Must have a mechanism for remote access to the development platform.
    b. Should be reconfigurable as fast as possible
    c. Designed to minimize the risk of a design error induced system failure
    d. Circuit board should conform to extended ATX form factor specification

It should be noted that there are other practical limitations on these requirements. This means that there needs to be an engineering trade-off between the various requirements and other external factors. These factors include things such as: cost, power requirement, and any space limitations of the circuit board.

3.3 THE TM-4 DESIGN

The following sections present the design of the TM-4, as motivated by the previously identified design requirements. A system-level block diagram will be introduced for the TM-4 and each block will then be examined in more detail. The complete schematics for the TM-4 can be found in Appendix 7.

3.3.1 DESIGN OVERVIEW

The design of the TM-4 consists of two major subsystems: the development subsystem, and the interface subsystem. The development subsystem contains the portion of the TM-4 that is directly usable by designers in implementing their designs. The interface subsystem is not directly usable by designers but instead provides support functionality. This functionality includes both controlling the TM-4 and providing a communication channel with the development system.

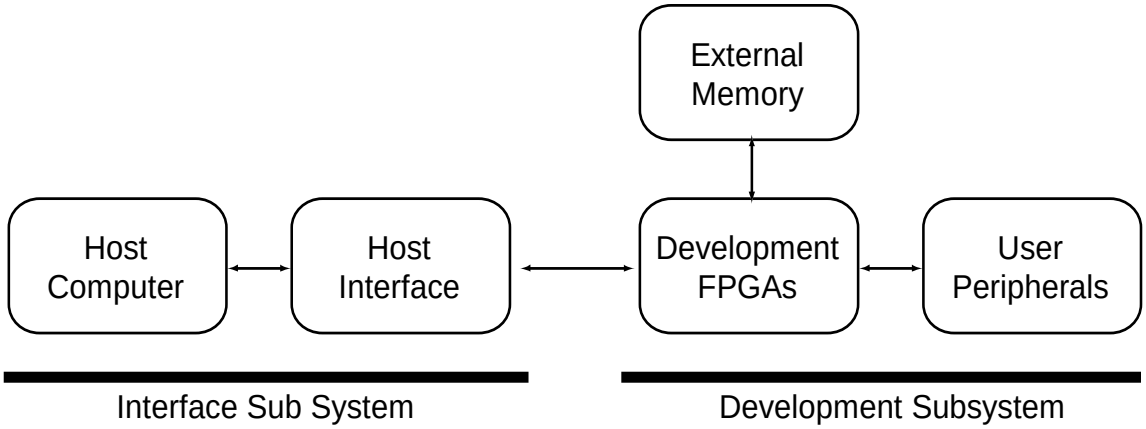


Figure 13: Top Level System Diagram

Figure 13 shows the division between development and interface in slightly more detail. The development subsystem is composed of programmable logic, external memory, and user peripherals that are all available for designers to use. The interface subsystem consists of a host computer and a host interface.

Each of these five components, the programmable logic, the external memory, the user peripherals, the host interface and the TM-4 controller will all be examined in the following subsections.

3.3.2 PROGRAMMABLE LOGIC

The programmable logic subsystem of the TM-4 is comprised of two different items: the FPGAs and the interconnection between them. The first two design requirements, 1 and 2 identified in Section 3.2.4, specified that the TM-4 should contain

four of the largest FPGAs available, be fully interconnected using point-to-point connections and provide as much bandwidth as possible. These requirements leave the question of which FPGAs to select and exactly how to connect them unanswered.

Each of the questions will be examined in the following two subsections.

3.3.2.1 FPGA SELECTION

The selection of which FPGA to use for the TM-4 was a choice between two different companies flag ship products: Altera's Stratix FPGA, and Xilinx's Virtex II Pro FPGA. Each FPGA had their own benefits and disadvantages. For example, the Virtex II Pro had more multipliers and more flexibility in the configurability of its I/O pins, whereas the Stratix had hardware support for certain types of RAM and termination schemes. In the end, neither FPGA was found to be clearly better then the other and the decision came down to the more practical consideration of availability. The largest Altera Stratix chip was available, whereas it was not clear if the Xilinx Virtex II Pro FPGA would be released in time, and as such, the Stratix was chosen for the TM-4.

Each of the four Altera Stratix S80 chips selected for the TM-4 provide 79,040 four-input lookup tables, 7.4Mb of on-chip SRAM, 176 embedded 9x9 multipliers and 1203 I/O pins. When combined the total usable development area of the TM-4 is 316,160 four-input LUTS, 29.6Mb of on-chip SRAM, and 704 embedded 9x9 multipliers.

3.3.2.2 FPGA INTERCONNECT STRUCTURE

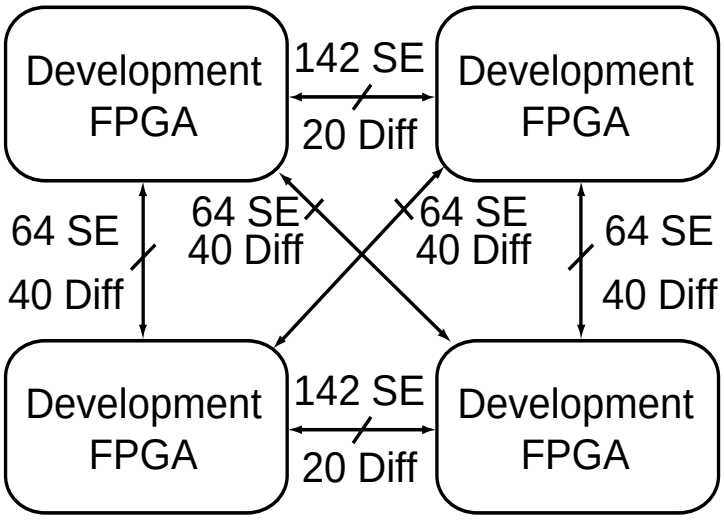


Figure 14: Development FPGA Interconnect Structure

The design requirements specified that the four development FPGA be fully interconnected, with each other, and have as much inter-FPGA bandwidth as possible. In order to meet these goals it was necessary to determine the number and type of connections between each of the FPGAs.

The Stratix FPGA supports two types of signalling standards, differential and single ended. The differential standards use two wires and can provide a theoretical data rate of 840Mbps, whereas the single ended standards use only one wire but only provide a data rate of 350Mbps. Of the two options, differential signalling provides the greatest throughput per pin.

The final decision of how many inter-FPGA signals to use and what standards they should be were based on a combination of Stratix hardware limitations and physical circuit board issues. Since the Stratix architecture supports only a limited number of differential signals it was necessary to use of both differential and signal ended signals. Figure 14 illustrates how the four development FPGAs are connected to each other and how many signals of each standard are used (where “SE” refers to single ended and “diff” refers to differential).

The differential signals are implemented using pairs of LVDS signal lines, and the SE, or single ended, signals are implemented using 2.5.v and 3.3v CMOS standards.

The total available bandwidth between pairs of FPGAs varies between 56Gb/s and 66.5Gb/s. This variation is the result of certain asymmetries in the Stratix architecture and the relative positions of the four FPGAs.

3.3.3 EXTERNAL MEMORY SELECTION

The design requirements of the TM-4 specified that the TM-4 should have at least 4GB of memory and have as much memory bandwidth as possible, item 3 in Section 3.2.4. This requirement raises the questions of what type of memory technology to use, how it should be connected to the development FPGAs, and exactly how much.

The selection of what memory technology to use was driven primarily by practical considerations. The amount of memory required, 4GB or more, meant that it was impractical to use SRAM, because of the number of components it would require. This meant that DRAM needed to be used, as it provides much greater memory capacity for the same number of components then SRAM. Once again the memory requirement suggested that it would be impractical to use discrete memory chips and that memory modules needed to be used instead. There were two types of memory module technology available at the time the TM-4 was being designed: DDR SDRAM and RAMBUS. Both technologies provided similar memory densities and bandwidths but DDR SDRAM could be easier incorporated in the TM-4, due to the Stratix's hardware support for this type of RAM.

Once DDR SDRAM was decided upon there was still the question of how many memory modules to use and how to connect the memory to the development FPGAs. The answer to this question needed to balance performance with cost. The total amount of memory bandwidth is proportional to the number of independent memory modules provided. However, each module comes at the cost of power, space, and expense. Since the Stratix FPGA has hardware support for up to two DDR SDRAM modules, the question became one of either using 1 or 2 independent RAM banks per FPGA. Since memory bandwidth was one of the driving goals of the TM-4 it was decided to use two DDR SDRAM modules per FPGA, for a total of 8 modules in total.

The use of standard memory modules allows for the TM-4 be populated with various amounts of memory. Each module can be populated with between 512MB and 2GB of ram running at upto 166MHz, the maximum specified speed for the Stratix FPGA. The standard configuration will contain 8 1GB modules and provide a total peak bandwidth of 17.8GB/s. In addition the TM-4 is designed to support future RAM modules up to 4GB in size.

3.3.4 USER PERIPHERALS

The design requirements for the TM-4 identified video applications as one possible use for the TM-4, and as such, specified several different peripheral requirements, item 4 in Section 3.2.4. The requirements indicated that the TM-4 should contain two NTSC analog video-in channels, one VGA video-out channel and two independent IEEE-1394 buses.

Since the first two peripherals, analog video-in and video-out, were both present on the TM-3, the same proven design was brought forward to the TM-4. The NTSC video-in channels were implemented using two Phillips SAA7111 decoder chips, and the VGA video-out channel was provided by an Analog Devices ADV7123 triple 10bit video DAC.

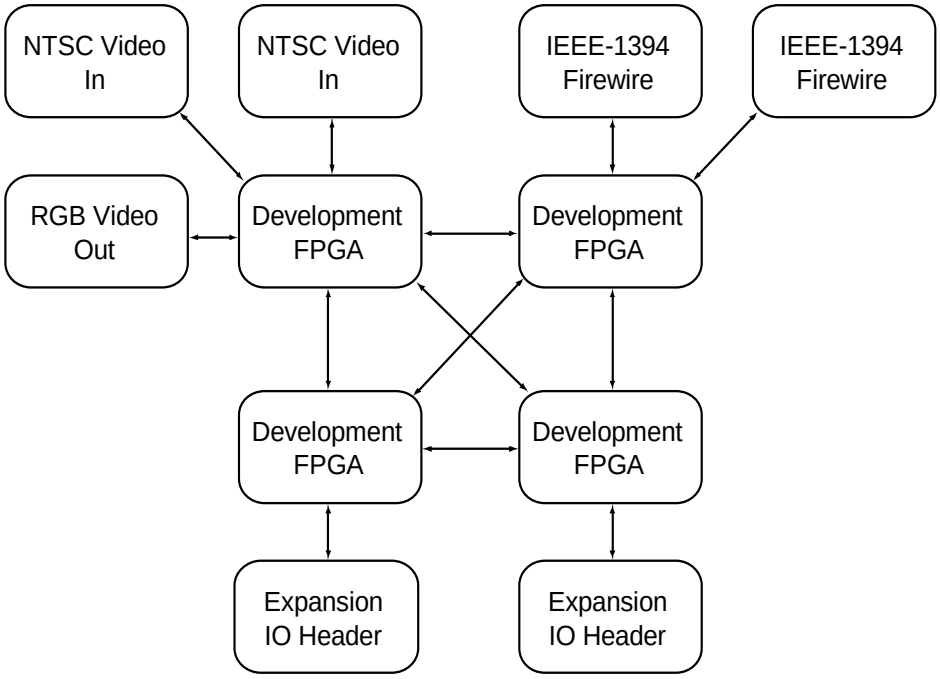


Figure 15: User Peripheral Connections

The IEEE-1394 bus is a new interface, which was not present on the TM-3. Its implementation is also more complicated due to the complicated communication protocols that it uses. The TM-4 was designed to implement as much functionality in hardware as possible, while still remaining flexible. This included using a 2 chip IEEE-1394 solution. These chips provide both the physical and link layers of the IEEE-1394 networking protocol. Users of the TM-4 must implement the remaining layers using the development FPGA. This division, between hardware components and logic within the development FPGA, was selected to allow the user of the TM-4 sufficient flexibility to control the bus how they see fit. This meant allowing the user to fully control all networking layers above the link layer.

Figure 15 shows how the different peripherals are connected to the four FPGAs. The top left FPGA handles all the analog video peripherals, include 2 video-in channels and one VGA out channel, the top right FPGA handles the two independent IEEE-1394 buses. The two remaining FPGA do not have any specialized peripherals but do have I/O headers available for future expansion.

3.3.5 THE HOST TO FPGA COMMUNICATION CHANNEL

The design of the communication link between the host computer and the development FPGAs was driven by two design requirements, the requirement for as much bandwidth as possible, and the requirement that it be easy for designers to use the channel. The latter requirement was already solved in the design of the TM-3 by providing a set of IP blocks, and software [40] running on the host computer, that abstract away the complexities of communicating with a host computer. This left the first requirement, maximizing bandwidth, as the focus of the TM-4's communication channel design.

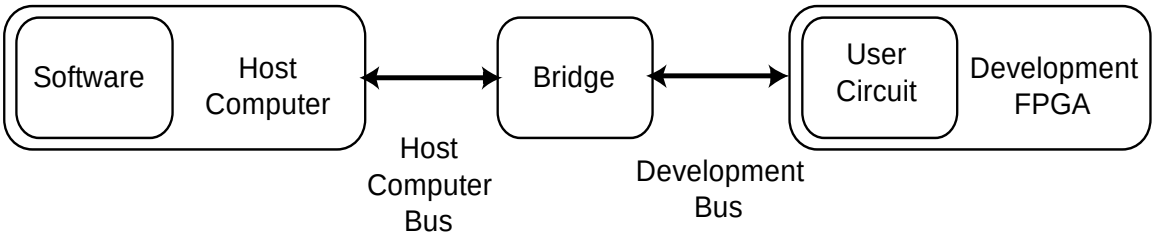


Figure 16: Simplified Host to FPGA Communication Channel

The communication channel between the development FPGAs and the host computer consists of a number of different components. Figure 16 shows a simplified view of the communication channel. For communication from the host computer to the development FPGAs there are several steps. First a piece of software must request that data be transferred to the development FPGAs. This data must then be transmitted from the host computer to a bridge within the TM-4 itself. This bridge must then pass the data onto the development FPGAs and ultimately to the circuit running within it. Transfers in the other direction must take the same steps, only in reverse.

The communication channel consists of four major components: the physical hardware links between the host computer, the bridge chip, and the development FPGAs, the IP core which implements the bridge, the IP cores running on the development FPGAs and the software running on the host computer. Each of these components will be examined in the following sections.

3.3.5.1 PHYSICAL HARDWARE COMMUNICATION LINKS

The host communication channel contains two physical hardware links, the link between the host computer and the bridge chip, and the link between the bridge chip and the development FPGAs. Each of these links was designed to meet the design requirement of having as much host computer bandwidth as feasible.

The first link, between the host computer and the bridge chip, needed to use a standard interface that was available in commodity computers. The links considered were links such as Firewire [41], USB [42], gigabit Ethernet [43], and PCI [15]. The selected link, was the link that provided the greatest bandwidth, PCI. In particular 66Mhz 64bit PCI was selected. This link provides a theoretical peak bandwidth of 528MB/s.

The second link, between the bridge chip and the development FPGAs, need not have been a standard interface and was custom designed. The link selected was a bus consisting of 32 data bits that could run at a data rate up to a 100Mhz. The result was a communication link that could sustain transfers of nearly 400MB/s. The reasoning behind the bus width and speed were that the bus needed to be easily combinable into 64bit PCI words, by combining two 32-bit words, and that the bus should still run synchronously, by keeping the clock below 100MHz. The resulting 400MB/s bandwidth was not expected to be a bottleneck to system performance due to the fact that the PCI bus's overhead prevents it from reaching its theoretical peak bandwidth.

3.3.5.2 HOST TO DEVELOPMENT BRIDGE

The connection between the host computer's PCI bus and the development FPGAs communication bus is bridged through the use of an Interface FPGA. This FPGA contains a custom design logic core that performs the translation between the two buses. Figure 17 shows a block level diagram of the tasks that are performed. The PCI interface is implemented using an Altera PCI IP core [44]. This core interfaces with two FIFO buffers, a read and a write buffer. These FIFO buffers allow for clock domain translation, between the PCI buses 66Mhz clock and the development FPGAs programmable clock, and buffer data for more efficient burst transfers. The two buffers

are also connected to the development FPGA interface circuit. This simple circuit decodes requests sent by the host computer and communicates with whichever of the four development FPGAs is required.

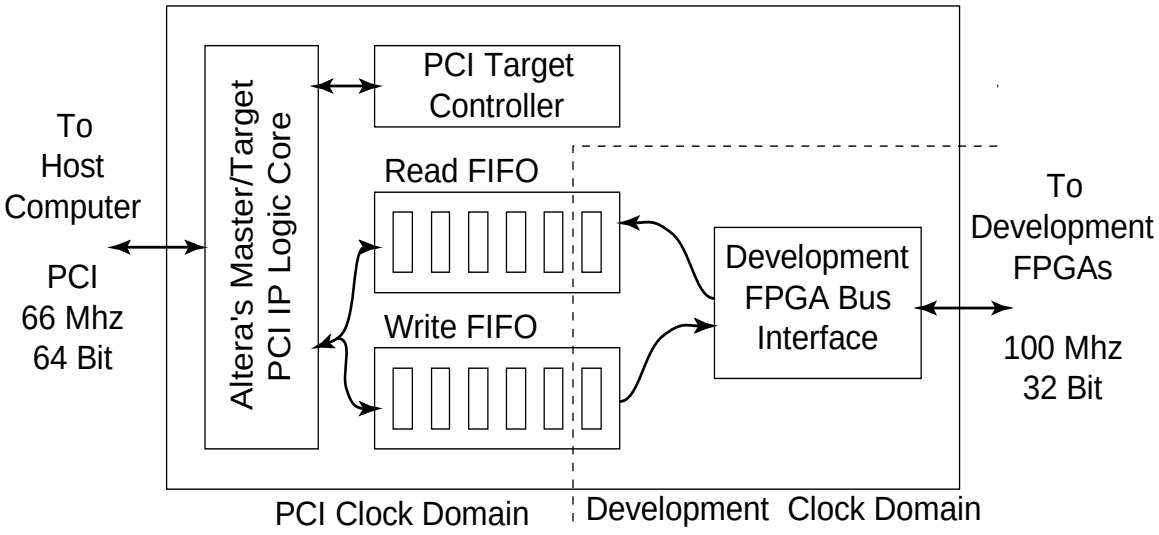


Figure 17: Host Communication Channel Bridge Functions

The complete VHDL code for the Interface FPGA can be found in Appendix 0.

3.3.5.3 PARAMETERIZABLE BUS INTERFACE LOGIC CORES

The physical communication link between the FPGA and the interface bridge incorporates a custom design bus protocol. In order to hide the complexity of interfacing with this bus, a set of parameterizable logic cores were created. These modules encapsulate all the functionality required to interface with the bus while presenting a simple handshaking based interface to the user. Instead of dealing with multi-cycle bus transactions the user only needs to interface with a simple three-wire handshake interface of one of the parameterizable cores.

3.3.5.4 HOST SOFTWARE

The last component of the host communication channel is the software that runs on the host computer. This software provides for a method for communication with the

TM-4 hardware. The functionality is divided into two components. There is a kernel-mode Linux device driver, which handles the details of setting up and sending raw data to the TM-4, and there is a library API that provides an easy way to communicate with the parameterizable bus interface logic cores running on the development FPGA.

The kernel mode driver is implemented as a character device that accepts read and write requests. These requests are directly translated in PCI DMA transfers between the host computer and the FIFO buffers in the TM-4's interface chip. It is the responsibility of the library API to send the correctly formatted data to the device driver.

The library API takes a description of development bus logic cores that reside in a design and provide a simple interface for communication with them. For example, transferring data from the host computer to the simple handshaking logic core, is as easy as issuing a single write call to the API. Likewise calls exist for reads and for error detection.

3.3.5.5 THE COMPLETE HOST TO FPGA BUS

Figure 18 provides an overview of the components that comprise the host computer to development FPGA communication bus. On the left of the figure is the development FPGAs. These FPGAs contain the user circuits and a library interface component. The library component implements the necessary protocols to communicate across the 32-bit development bus to the bridge within the Interface FPGA. This bridge buffers the data using FIFOs, performs the necessary clock domain translations, and transfers data between the FIFOs and the host computer over a 66MHz 64bit PCI bus. The host computer's kernel driver interfaces with the PCI bus and provides the data to a user's C program. This program incorporates an interface API that provides an easy abstract interface for communication with the library interface component on the development FPGA.

The only steps of this process that a user of the TM-4 must understand are the interface to the parameterizable bus interface logic cores, and the interface API. All the rest is hidden away.

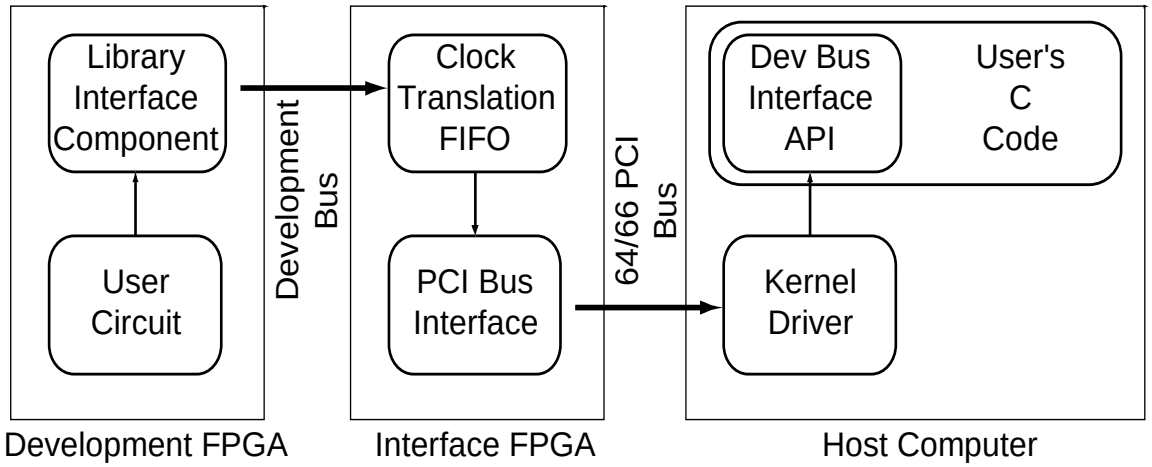


Figure 18: Development Communication Bus

3.3.6 TM-4 CONTROLLER

In order for the TM-4 to function there are a number of different basics functions that must be performed. These include things such as programming the development FPGAs, configuring the clocks, monitoring the FPGA temperatures, and several others. This functionality is provided by the same interface FPGA that provides the bridge functionality of the host to development FPGA communication bus.

The following sections will describe how these important functions are performed by the interface FPGA and will be followed by a complete system diagram of how the interface FPGA is connected to each of these functions.

3.3.6.1 DEVELOPMENT FPGA CONFIGURATION

One of the secondary goals of the TM-4 is to have it configurable as fast as possible. To achieve this the TM-4 was designed to utilize the fastest configuration mode available in the Stratix FPGA, the fast passive parallel, or FPP mode. In this mode each FPGA can be provided 8 configuration bits at a maximum rate of 100Mhz. In addition to this, each of the four FPGAs can be configured in parallel to further reduce the configuration time.

A secondary method to configure the development FPGAs is through a JTAG chain. This chain can provide one bit of configuration, to one FPGA, per clock cycle and acts as a backup in case a design error prevents the use of FPP mode. Using JTAG, configuration of the TM-4 requires tens of seconds.

3.3.6.2 TEMPERATURE MONITORS

A useful function for the TM-4 to have is the ability to detect a potentially dangerous overheat situation, caused by a flaw in a users design, and to safely stop operation. This can be accomplished through the use of each of the five FPGAs in the TM-4 internal temperature diode. These probes are connected to the interface FPGA using two temperatures probe chips from Maxim. The chips consist of a 4 channel MAX1668 temperature monitor chip, and a MAX1618 single channel monitor chip. The interface FPGA monitors both of these chips and automatically shutdown any chip whose temperature exceeds a safe threshold.

3.3.6.3 THE CLOCKING SUBSYSTEM

Incorporating all the different components necessary to meet the design requirements introduced the need for a large number of different clocks. Each of the DDR SDRAM banks, and each of the peripherals all require their own clocks. In addition the PCI bus required a clock, as did the buses between the four development FPGAs and the bridge within the interface FPGA. In addition to just providing these clocks, there was also a need to provide a way to perform globally synchronous transactions between various components of the TM-4.

The clocking architecture for the TM-4 was selected to be as easy-to-use as possible, while not adversely affecting the other core goals. This meant that globally synchronous clocking would be used whenever possible, and that peripheral clocking would be handled completely by the user of the TM-4.

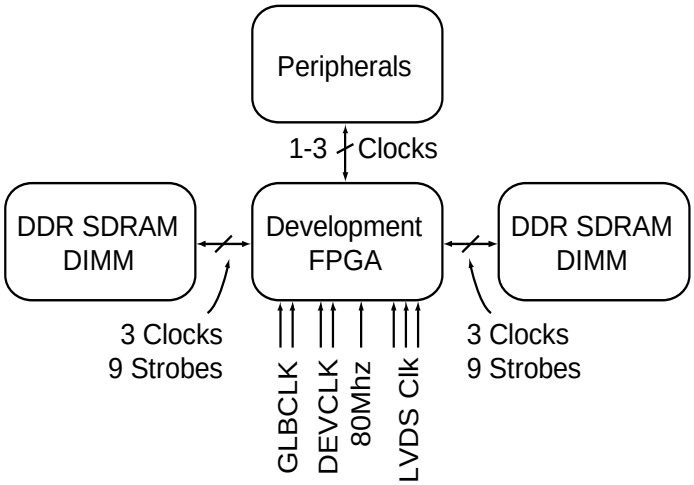


Figure 19: Development FPGA Clocking Structure

Figure 19 shows the clocking architecture that was selected and how it is connected to a development FPGA. There are two independent board wide synchronous clocks called DEVCLK and GLBCLK. Since the Stratix FPGA has multiple PLLs, each that require an external clock input, each global clock is connected several times to the same FPGA. Both these global clocks are driven by a programmable PLL in the interface FPGA. These PLLs can generate any frequency between 1 and 100MHz.

The development FPGAs are also each connected to an 80MHz low-jitter reference clock. The purpose of this clock is to provide a low jitter reference clock for use in high-speed serial LVDS communication. This reference can be used to generate precision source-synchronous clocks for use in the inter-FPGA LVDS communication channels. Each development FPGA has three such clocks, one from each of the remaining FPGAs, driving on chip PLLs.

Each of the development FPGAs is also connected to two independent DDR SDRAM modules. The module requires three clocks each, to be generated by the development FPGA. These clocks act as a source-synchronous clock for transmitting data and commands to the modules. The development FPGAs are also connected to various peripherals that require clocking. These clocks are typically fairly slow and are generated by the development FPGAs internal PLLs.

The interface FPGA clocking structure is simpler than that of the development FPGAs. Since its primary function is to act as a bridge between the PCI bus, with a fixed

66Mhz clock, and the development FPGA's programmable clock domain, DEVCLK, it only needs to be connected to those two clocks.

3.3.6.4 INTERFACE FPGA BLOCK DIAGRAM

The interface FPGA is responsible for communicating with the development FPGAs, configuring the development FPGAs, generating the programmable clocks, accessing the JTAG chain of the development FPGAs, and performing clock domain translation, all under the control of the host board computer. Figure 20 shows a block diagram of the logic contained in the house keeping chip and Appendix 0 contains the VHDL code that implements it.

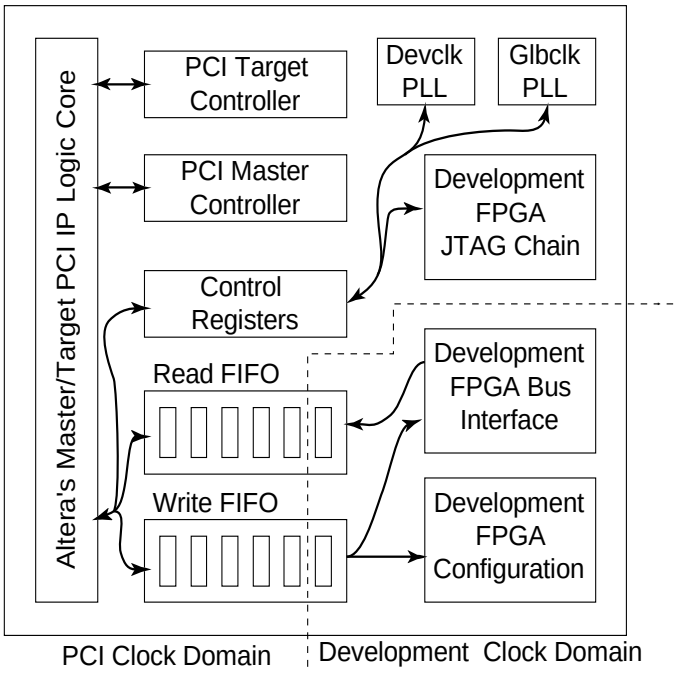


Figure 20: Housekeeping Chip Logic Core

The entire FPGA logic core is built around an Altera master/target 66Mhz/64bit PCI IP core. This core provides a simplified interface to the PCI bus. Access to the housekeeping chip is provided through two PCI memory regions. The first region allows access to a set of control registers. These registers control various aspects of the TM-4 such as the programmable clock frequencies and the temperature monitors.

The second memory region provides access to two FIFOs, the read FIFO and the write FIFO used to bridge the PCI bus to the development bus. The FIFOs are implemented using a dual ported ram with independent clocks on each port.

The two FIFOs provide data communication between either the development FPGA bus interface or the development FPGA configuration logic. The development bus interface is a small state machine that transmits write data and read requests to the corresponding bus interfaces on the development FPGAs and returns any read results to the read FIFO. The VHDL code for the development FPGA side bus interfaces can be found in Appendix A.

The development FPGA configuration unit contains the logic necessary to program the development FPGAs using the FPP mode. The unit is designed to run at 100Mhz, the maximum rate at which Stratix FPGAs can be configured.

3.3.7 THE POWER SUBSYSTEM

The design of the TM-4 calls for a large number of different components to be integrated together. Each of these components have there own power requirements. In total there is a need for 1.25v, 1.5v, 2.5v, 3.3v, 5v and 12v power supplies. The 1.25v supply is used for transmission line termination in the DDR SDRAM subsystem. The 1.5v supply provides the core power to the FPGAs, the 2.5v and 3.3v supplies are used for inter-chip signalling and the 12v supply is used for powering the Firewire bus.

The design selected for the TM-4 consisted of two layers of power conversion. The first layer is a standard ATX power supply. This supply is capable of providing 3.3v, 5v and 12v power. The second layer of power conversion consists of on-board DC-DC converter to generate the 1.5v and 2.5v supplies and 8 linear converters for the 1.25v termination voltage, one for each DDR SDRAM module. This style of power distribution system was selected as it met the needs of the TM-4 while still using low cost standard parts.

3.4 HARDWARE DESIGN VALIDATION

The TM-4's design is very large and very complicated; there are 17,279 pins and 4,238 different nets. It was quite likely that there would be some number of mistakes

made in the design process and a great deal of time was spent attempting to validate portions of the design. The process used to validate the hardware design was fairly simple. It consisted of first identifying possible sources of errors and then applying various checks to try and verify that these sources did not produce actual errors.

The identified potential sources of design error were as follows:

1. Electrical functionality errors
    a. Mistake in the schematic
    b. Designer misunderstanding of devices functionality
2. Incorrect component database information
    a. Component value (i.e. resistance capacitance)
    b. Schematic pin to physical package pin mapping
    c. Physical component shape for circuit board mounting
3. Schematic or component revision mismatch
4. Exceeding passive components voltage or power ratings
5. Signal integrity problems
    a. Termination topology

The validation process of the TM-4 design flow consisted of two types of validation methods. The first set of validation methods tries to find mistakes arising from errors in entering the schematic data into the design software, this method will be called data entry validation. This type of validation method attempts to find errors generated by sources 1a, 2, and 3 in the list above. The second set of validation methods tried to find errors in the actual design itself by verifing that the circuits are functionally correct. This method will be called functional validation. Functional validation attempts to find errors generated by sources 1b and 5. Each of these two methods will be described in the following two subsections and example of how they were applied will be presented.

3.4.1 DATA ENTRY VALIDATION

In a design as large as the TM-4, the probability of having a piece of information incorrectly entered into a software design tool is quite large. These types of errors might

be as simple as a typographical error in a part number or an unintentional short in a schematic diagram. In both cases the design entered into the software tools is not as the designer intended. The method employed to try and catch these types of errors was to check the entered designs against a set of validation rules in a process known as design rule checking, or a DRC test. In total three different types of DRC tests were performed.

The first types of DRC tests were those that the design entry tool, Mentor's Board Station Software, contained. These tests were designed to detect common schematic errors. For example, one test would verify that every pin in the design is connected to another pin. If a given pin in the design were meant to be floating, it would need to be explicitly defined as such. Another type of DRC test was designed to find different types of shorts between nets, for example between power nets, or explicitly named nets. Over all, Board Station's DRC tests were very good at finding errors without actually having any direct knowledge of what the designer had intended. In order to find cases where the designer had entered erroneous data, but in a correct format, custom DRC tests had to be designed.

The second type of DRC tests, custom DRC tests, incorporated more information than just the entered design. For example there was a custom DRC test designed, using the scripting language within Board Station, that verified that the pin mappings for large components was correct. The component at greatest risk for a mapping error was the large 1508 pin FPGAs. To ensure that this component was correctly mapped, a script was written to compare the mapping described in the net list, to a pin out file provided by the device's manufacturer.

The remaining possible schematic entry errors, such as value mismatches and package mismatches, could not be validated using an automated script. Instead these possible errors were check by hand using validation checklists. These check lists contained a number of possible errors that each component in the designed needed to be verified against.

The data validation phase successfully detected a number of different errors.

3.4.2 FUNCTIONAL VALIDATION

The process of functional validation was much more difficult to perform than the simple schematic entry checks. The typical way of verifying electrical functionality is through simulation. In order to simulate a circuit each component in the system must have a functional model. Unfortunately, this was not the case for many devices in the TM-4.

The primary problem with simulating the TM-4 is that, by itself, the TM-4 does not do very much. Most of the complex subsystems, such as DDR SDRAM and IEEE-1394, are connected to the four main development FPGAs. In order to simulate the functionality of either the DDR SDRAM or the IEEE-1394 bus, it would be necessary to simulate the functionality of a circuit running in the FPGA. This is not feasible, as any test circuit would also need to be verified to be functional, only further complicating the issue.

The approach that was taken to verify the TM-4 electrically was two-pronged. First, subsystems for which models existed were simulated and second, subsystems without models were carefully scrutinized by hand. For both cases the physical interconnect topology was simulated to verify their signal integrity.

The only subsystem that was simulated was the power conversion system. This system is particularly prone to error as it is primarily analog in nature. It was possible that the system might be unstable, or otherwise unusable. In order to validate functionality a spice model was created for the DC-DC controller and this model was tested under expected operating conditions. Appendix F contains the spice models used in simulation.

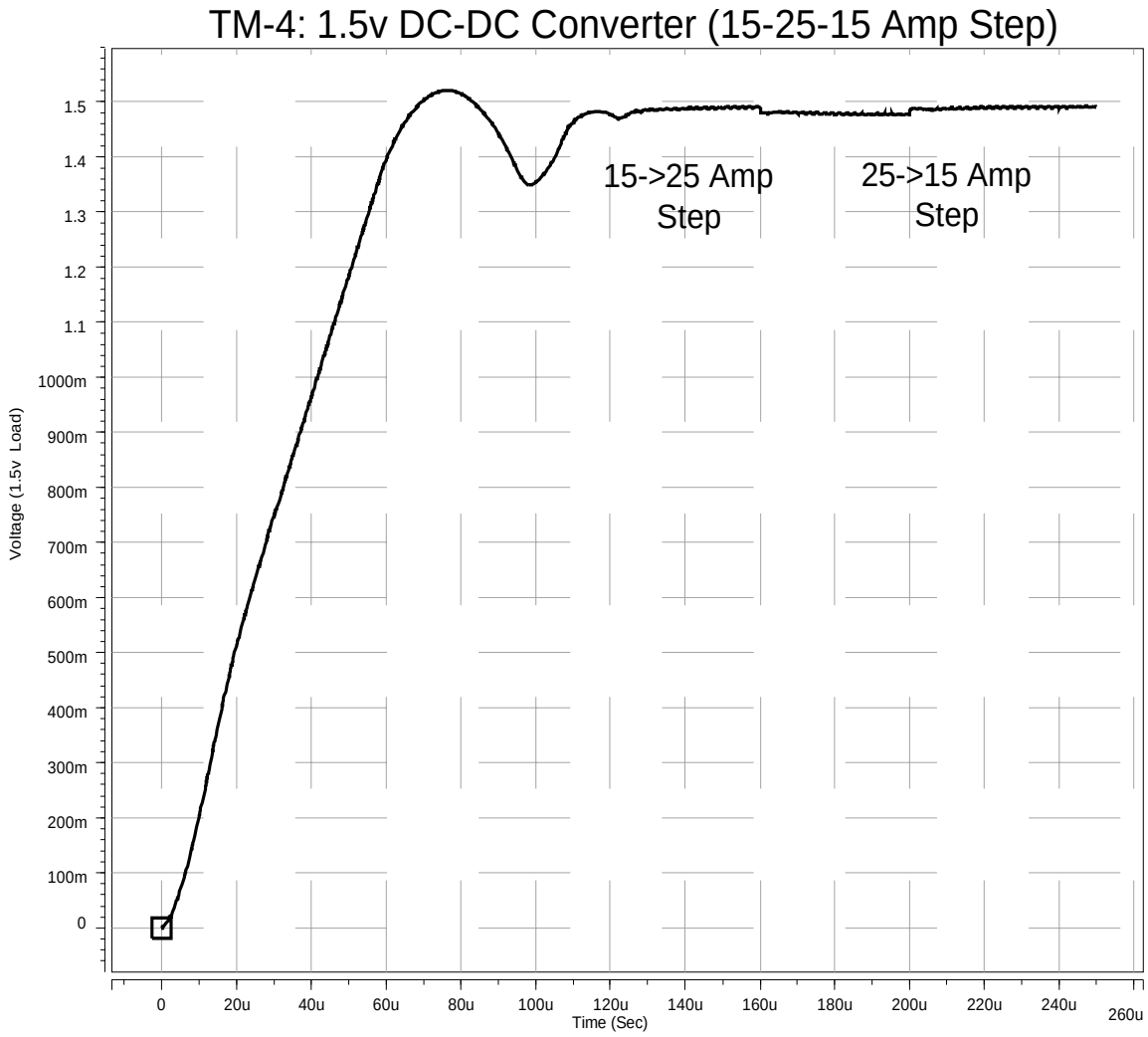


Figure 21: 1.5v DC-DC Converter Simulation

Figure 21 shows the result of one simulation test of the 1.5v DC-DC converter subsystem of the TM-4. The graph shows the output voltage of the DC-DC convertor while simulating several events. The first event is power-up under a load current of 15Amps. Under these conditions the DC-DC converter reaches a stable voltage after 130uSec. The next event, shown on the graph, is a load current step from 15 to 25 amps. This step was modelled by changing the load resistance from 0.1ohm to 0.06ohm. The result was a slight voltage drop with no ringing or other problems. Similarly, the step from 25 back down to 15 amps shows an equally good result.

3.5 SUMMARY

This chapter presented the first three steps of the design processed used to create the TM-4. These steps consisted of requirement identification, schematic design of the circuitry and verification of the schematic. The requirements were first presented as directed by past experience and anticipated future needs. These requirements were then used to motivate the presented design. This chapter concluded with a brief discussion about the verification techniques used to validate the schematic.

The next chapter will look at the final step in the design process of the TM-4, the design of the printed circuit board.

4 CIRCUIT BOARD DESIGN

4.1 INTRODUCTION

Once the schematic design and all the validation tests had been completed, the next step in the design process of the TM-4 was to design the printed circuit board (PCB). The PCB design process involved the following steps. First, a stack up, or description of the different layer thicknesses of the PCB, was defined. Next, the various components that made up the TM-4 were placed on the circuit board. An iterative routing process then wired the components up to each other and finally, the signal integrity was simulated for all high-speed nets to insure that both timing and electrical requirements were met.

Each of these steps will be described in the following subsections. The final PCB layout can be found in Appendix A.

4.2 PCB STACK UP

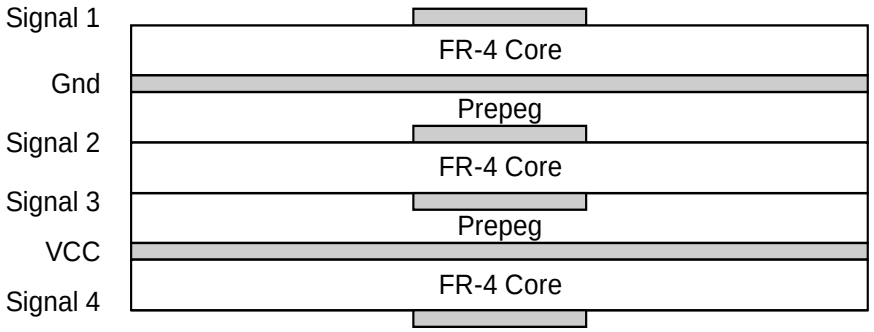


Figure 22: Sample PCB Stack Up

The term PCB stack up refers to a description of the different layers that make up a circuit board and must be defined before any additional PCB design can be completed. To help understand what a PCB stack up is, consider Figure 22. The figure shows the cross section of a simple 6-layer board. Each one of the grey boxes represents a layer of

copper in the PCB, and the white boxes represent the insulator that keeps the copper layers apart. Each of the 6 layers is assigned a specific purpose. The four layers marked as “signal” are used for inter-chip connections. The other two layers, “gnd” and “vcc”, perform a dual purpose. The first purpose is to provide power to the components on the board and the second is to shield layers from electrically coupling to each other.

The design of such a PCB stack up must take a number of considerations into account. These considerations include electrical issues, such as electrical impedance of traces, power handling, inter-layer coupling and inter-plane capacitance of power planes, as well as physical construction issues, such as board warpage, and minimum feature size. Each of these considerations is discussed below.

4.2.1 PCB STACK UP PHYSICAL CONSIDERATIONS

The design of a PCB stack up is restricted by a number of physical requirements. In order to manufacture a PCB reliably, these requirements must be obeyed. The simplest requirements are related to minimum feature sizes. For example, there is a limit on how small a copper wire, or track, can be created and how far from another track it can be placed. The exact values of these limitations are related to cost. A typical volume PCB process can easily handle wires as small a 5 mils with a separation of only 5 mils from other wires. A more expensive PCB process could generate wires even smaller, down to only 3 mils width, with 3 mils separation. In order to limit the cost of the TM-4, wire size and spacing were both limited to 5 mils.

Another requirement that a PCB stack up design must meet is one of layer symmetry. Basically, the stack up of a PCB can be thought of as a heterogeneous material in which each layer has a different coefficient of thermal expansion. A solid copper plane layer will have a much different expansion coefficient than a signal layer, which is made up of both copper and dielectric materials. If the plane layers of the PCB are not designed symmetrically then the different expansion coefficients can cause the PCB to warp.

4.2.2 PCB STACK UP ELECTRICAL CONSIDERATIONS

A design of a PCB must meet a number of different electrical requirements in order to be functional. The design must provide traces that have the proper impedance, which is important in high-speed signalling, be designed to limit the amount of inter-layer coupling, or cross-talk, and provide the necessary power distribution functionality, including current handling, and high-frequency decoupling.

The variables that can be changed in a PCB stack up design are the inter-layer separation, the size, or width, of wires in a layer, the thickness of copper in a layer, the dielectric material and the types of copper layers, either signal or plane layers. The first step in actually designing a stack up is to determine the number of layers and decide which layers are plane layers and which are signal layers. In the case of the TM-4, there were 16 layers total, with 10 signal layers and 6 plane layers. The layers are ordered in such a way that the power planes can provide cross talk shielding between different signal layers.

Once the layer ordering was determined, the next step was to specify the layer spacing and nominal trace widths, such that the correct trace impedance results were obtained and the traces were as tightly coupled to a ground plane as possible. The layer separation of the TM-4 was first selected by using the closed form relationships between the various parameters described in [45]. The equations provided a general idea of how thick the PCB would be, for a given trace size, and allowed the parameters to be adjusted. Once an acceptable solution was found, the PCB stack up information was sent to the PCB manufacturer. The manufacturer then used their knowledge of their manufacturing process and a 3D field solver to further refine the stack up. Figure 23 shows a description of the TM-4's stack up returned by the PCB fabrication house.

							50 ohm single ended geometry line	100 ohm differential geometry line / space
		soldermask				0.0010		
		plating				0.0011		
		plating (bv)				0.0011		
	1	signal		0.5oz		0.0007	6 mil	5 mil / 8 mil
		dielectric				0.0040		
	2	plane	B	0.5oz	T	0.0007	-	-
		dielectric	L		H	0.0050		
	3	signal	I	0.5oz	R	0.0007	6 mil	5 mil / 8 mil
		dielectric	N		U	0.0050		
	4	signal	D	0.5oz		0.0007	6 mil	5 mil / 8 mil
		dielectric			H	0.0050		
16 layer	5	plane	V	0.5oz	O	0.0007	-	-
		dielectric	I		L	0.0050		
Getek/N4000-13	6	signal	A	0.5oz	E	0.0007	6 mil	5 mil / 8 mil
Megtron material		dielectric			S	0.0050		
	7	signal	1	0.5oz		0.0007	6 mil	5 mil / 8 mil
		dielectric	|			0.0050		
Dk=3.85	8	plane	15	2.0oz	1	0.0028	-	-
		dielectric			|	0.0050		
	9	plane		2.0oz	16	0.0028	-	-
		dielectric				0.0050		
	10	signal		0.5oz		0.0007	6 mil	5 mil / 8 mil
		dielectric				0.0050		
	11	signal		0.5oz		0.0007	6 mil	5 mil / 8 mil
		dielectric				0.0050		
	12	plane		0.5oz		0.0007	-	-
		dielectric				0.0050		
	13	signal		0.5oz		0.0007	6 mil	5 mil / 8 mil
		dielectric				0.0050		
	14	signal		0.5oz		0.0007	6 mil	5 mil / 8 mil
		dielectric				0.0050		
	15	plane		0.5oz		0.0007	-	-
		plating				0.0011		
		dielectric				0.0040		
	16	signal		0.5oz		0.0007	6 mil	5 mil / 8 mil
		plating				0.0011		
		soldermask				0.0010		
						0.0948	+/- 10% overall	

Figure 23: The TM-4's PCB Stack Up

4.3 PCB COMPONENT PLACEMENT

The goal of the PCB component placement step of the TM-4's PCB design process was to assign a physical location to every component that made up the TM-4. The components needed to be located either on the top, or on the bottom, of the PCB with a given orientation, such that the amount of routing necessary to inter-connect

components could be minimized. In addition to this, the design requirements, discussed in Section 3.2.4, specified that the TM-4 must fit into a standard computer case. This constraint introduced both a size limitation on the PCB and restricted where certain components could be placed.

Placement was also constrained by electrical requirements. Certain components, such as bypass capacitors and termination resistors, had to be located in specific locations for electrical reasons. The entire placement step was performed by hand.

Figure 24 shows a floor plan diagram of the PCB. Figure 25 shows the placement of those components located on the top of the PCB and Figure 26 shows the placement of those components located on the bottom of the PCB, as viewed from the top.

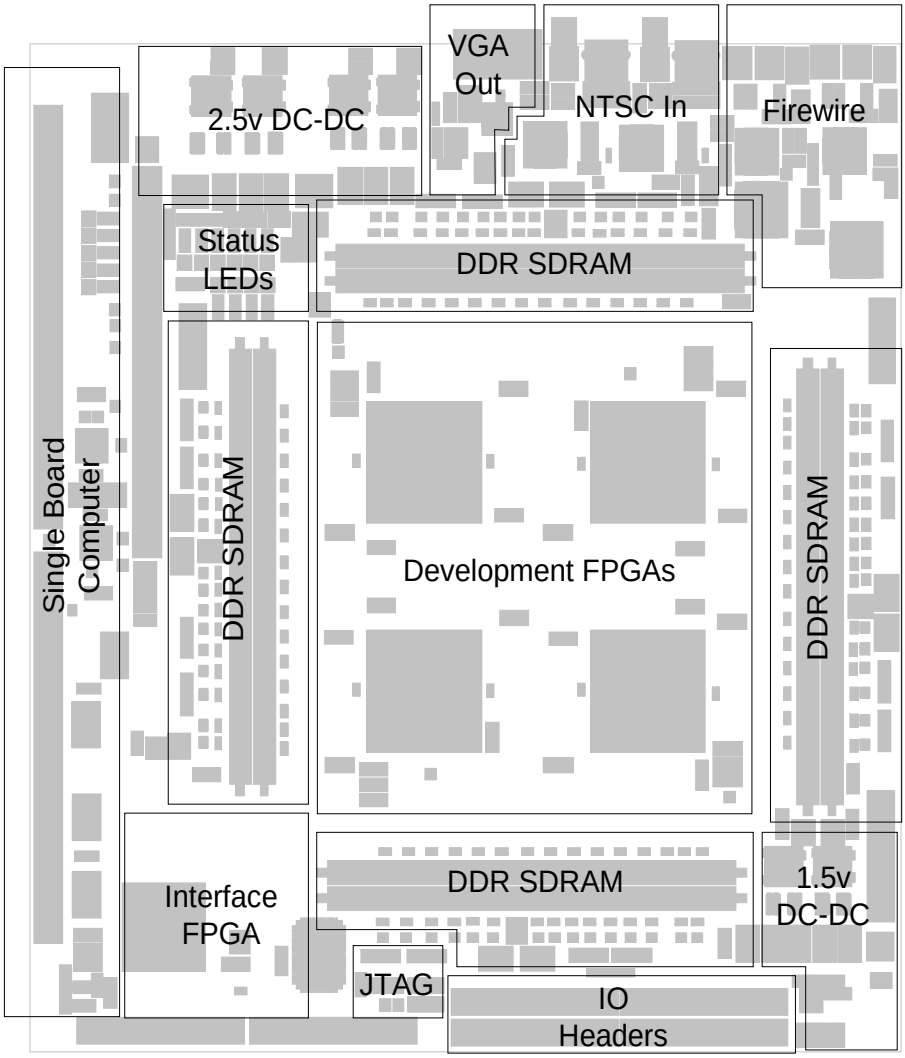


Figure 24: PCB Floor Plan

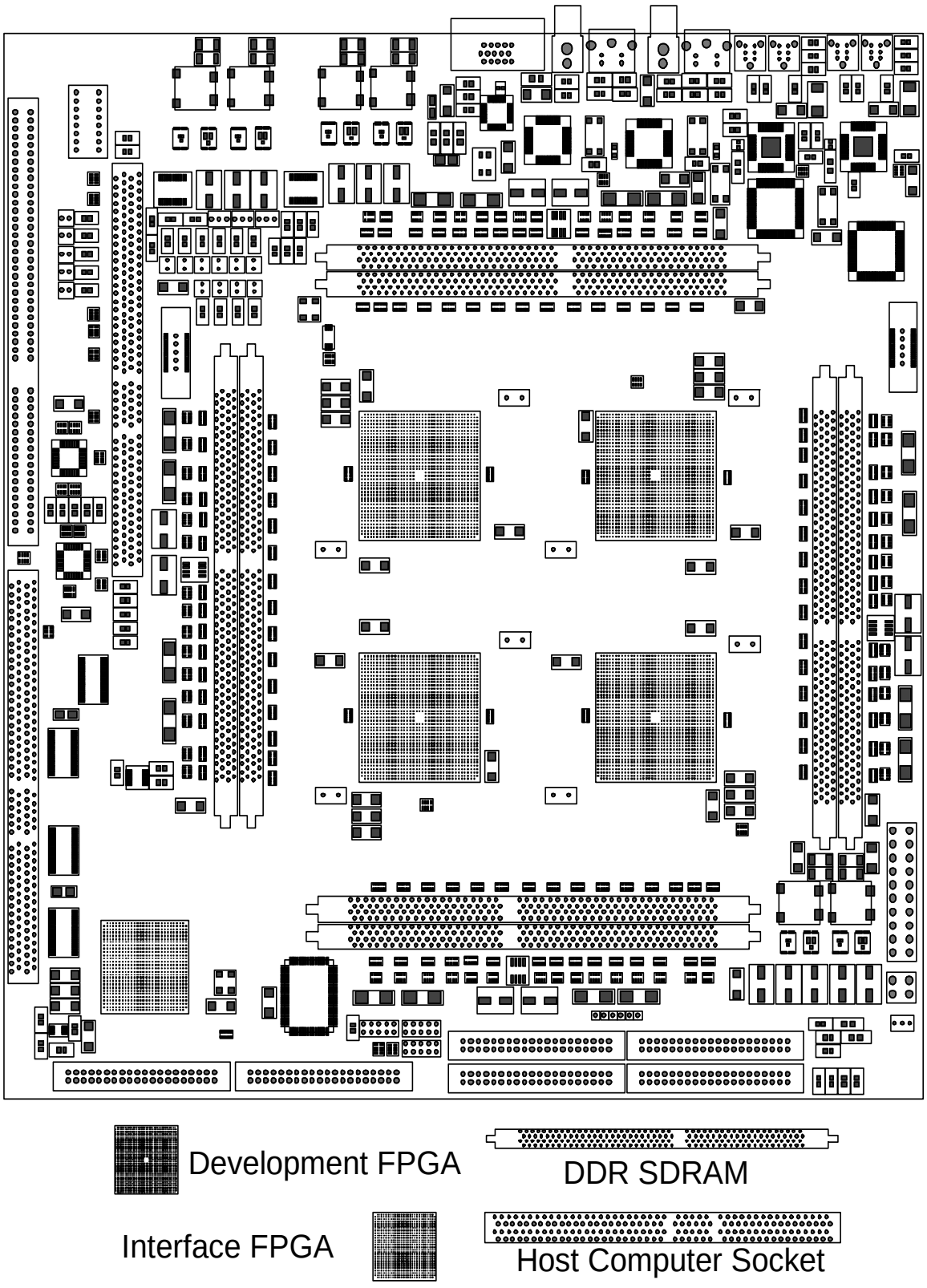


Figure 25: PCB Component Placement Top

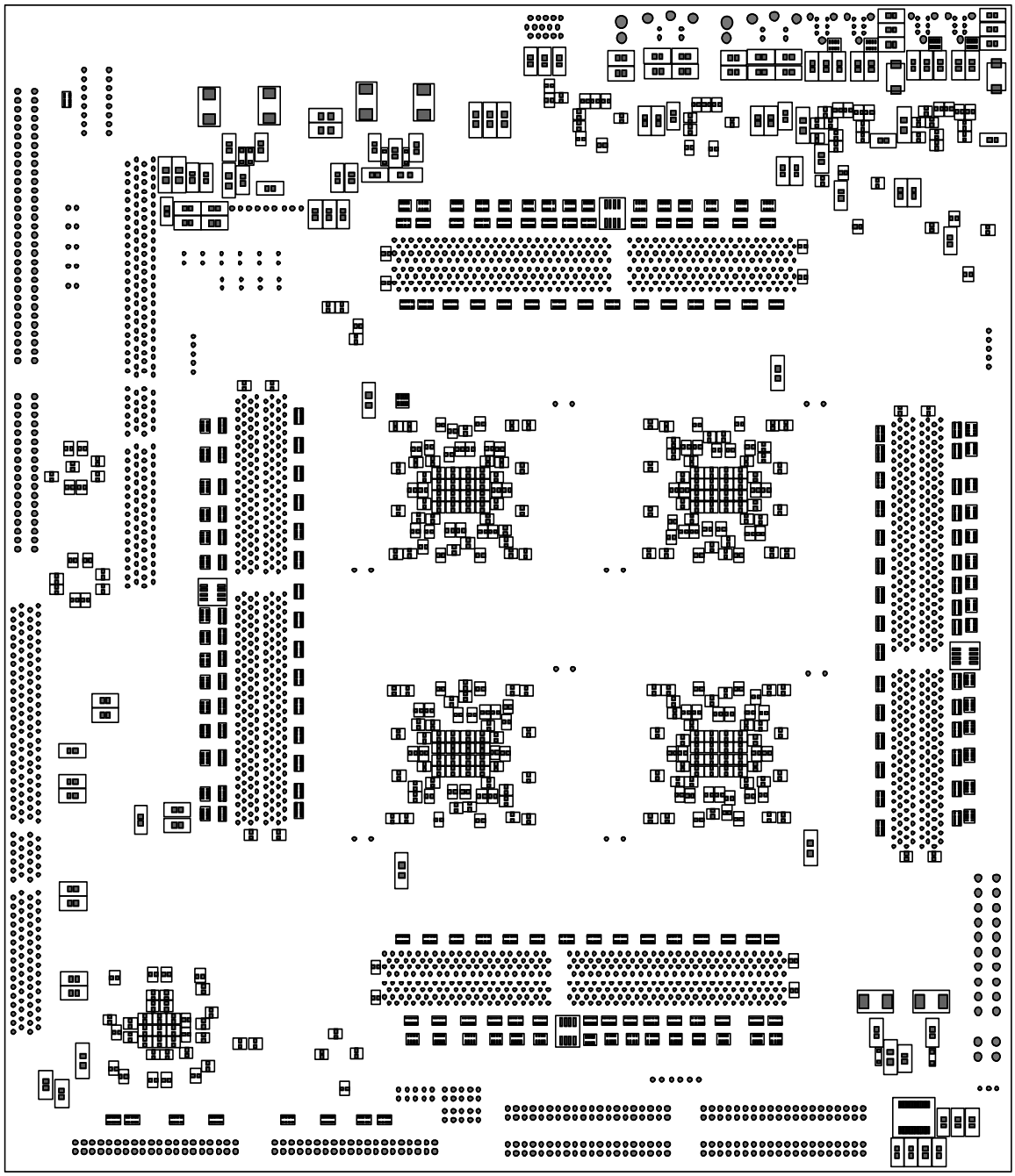


Figure 26: PCB Component Placement Bottom

4.4 PCB ROUTING

The purpose of the PCB routing stage of the TM-4's PCB design process was to design the physical copper layout needed to connect the different components. The routing process consisted of four different stages that were iteratively applied. The first

stage consisted of hand routing a breakout pattern for every component. The second stage performed equivalent pin swapping in order to eliminate unnecessary signal crossing. The next stage used both auto and hand routing to interconnect the component's breakout patterns and the final stage performed signal integrity simulations to insure electrical correctness. These stages were each iteratively applied, as the results from later stages revealed better solutions to earlier stages.

Each of these four stages, and the coupling between them, will be examined in the following sections.

4.4.1 BREAKOUT PATTERN CREATION

Many components used in the TM-4 utilize surface mount technology. That means that the component only connects to the top or bottom layer of the board and does not have any direct connection to the internal layers. A breakout refers to a routing pattern that defines how each surface mount component is connected to internal board layers.

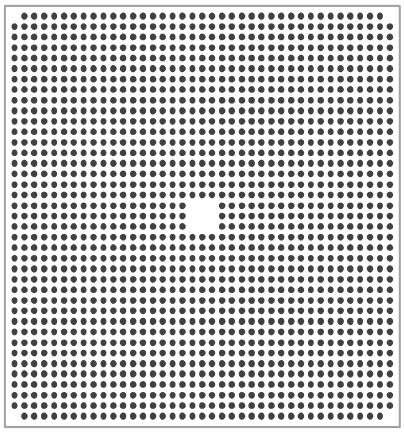


Figure 27: Stratix FPGA Landing Pattern

To better understand this, consider Figure 27. This figure shows the PCB "landing pattern" for the 1508 pin Stratix FPGAs used in the TM-4. Each circle represents a copper pad on the PCB for which a pin of the FPGA will be soldered. The pads are spaced on approximately 40-mil centres and are 18 mils in diameter. This leaves only 22 mils of space between adjacent pads. This space is only large enough for one 5-mil trace to pass through, while maintaining a minimum clearance of 5 mils. In order for

each pad to be able to escape, or breakout of the array of pads, it is necessary to use a multiple layer breakout pattern.

The first step in designing a breakout pattern is to provide a connection from each surface pad to the internal layers of the PCB. This is accomplished using what is called a "via". A via is a copper plated hole in the PCB that provides a connection between layers. Once the pads have been connected to the internal layers, the signals can then be routed out of the array.

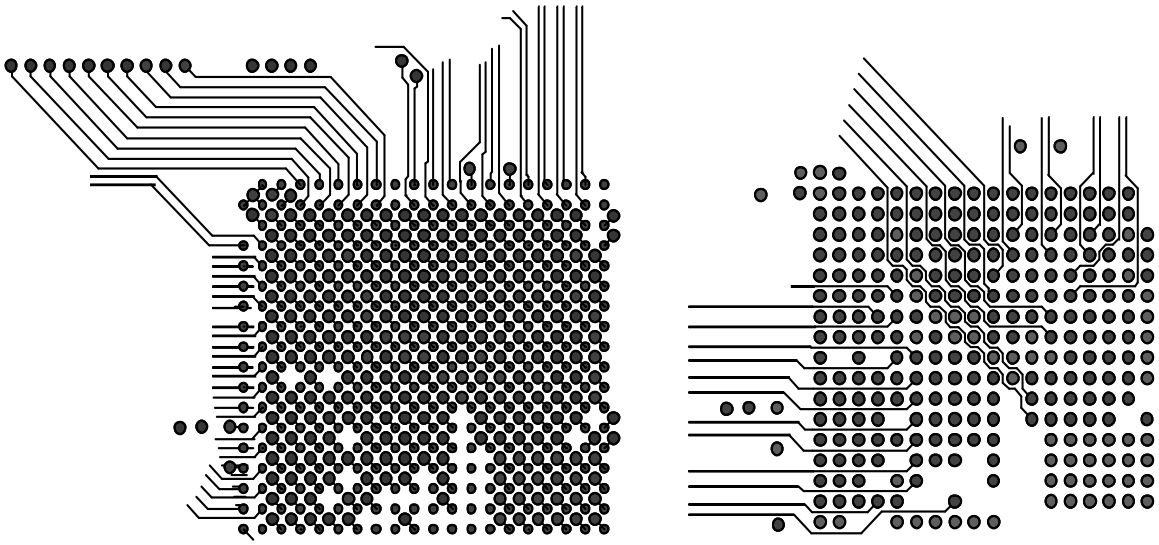


Figure 28: Stratix FPGA Partial Breakout Pattern

Figure 28 illustrates how multiple layers can be used to allow signals to escape the array. The figure shows two layers of the breakout for the 1508pin Stratix FPGA. The left image is that of the top layer. The two rows of pins on the top and left can be directly routed out of the array without violating any spacing requirements. However, the traces used to break these pads out now block the rest of the pads. These pads are instead connected to internal layers using vias, represented by the larger circles in the figure.

The figure on the right shows an internal layer of the breakout pattern. The pads are no longer visible, as they are only on the surface. Instead, only the circular vias are seen. This layer is not blocked by the escaping signals on the top layer and consequently, is used to break several more signals out. In total, 7 additional layers are used to completely breakout all 1508 pins of the Stratix FPGA.

4.4.2 EQUIVALENT PIN SWAPPING

Equivalent pin swapping is the process of modifying a design's net list, by swapping equivalent pins, in order to minimize the number of unnecessary signal crossings in the routing stage. To understand why this can be safely done consider a simple AND gate. Both of the inputs of an AND gate are functionally equivalent. If the input connections are swapped with each other, the function of the AND gate will be unchanged. The equivalency is even more prevalent in FPGAs, as almost any pin can be exchanged by only changing the circuit running within it.

Having this amount of flexibility in pin selection allows for pins to be connected in ways that simplifies the PCB routing. To better understand this, consider Figure 29. Figure 29 shows a bus between two FPGAs. It can be clearly seen that the wires running between each other are straight and therefore do not need to cross over each other. This was accomplished by careful pin swapping, such that the two FPGAs breakout patterns would breakout connecting signals to the same layer, in the proper order.

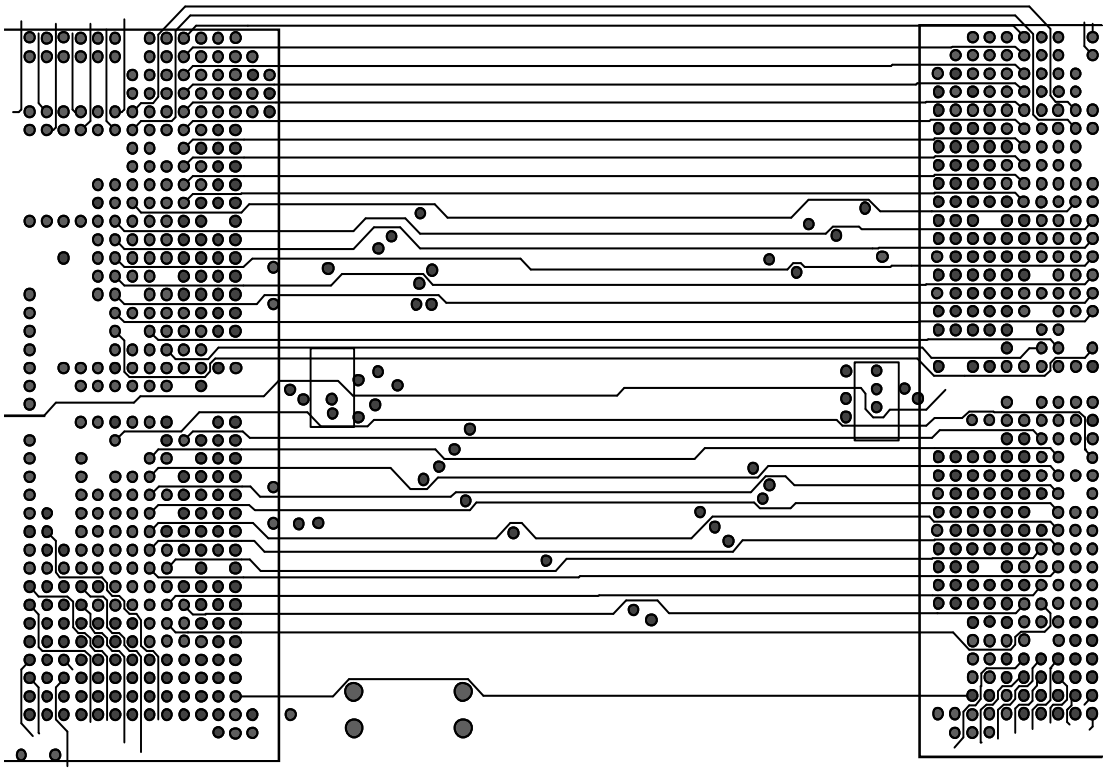


Figure 29: Pin Swapping Example

Accomplishing this was very time consuming due to the coupled nature of the problem. Both pin swapping and changes to the breakout pattern were performed so that the inter-FPGA connections would line up correctly.

4.5 PCB ROUTING

The next stage of routing the PCB was the board-wide routing stage. This stage involved the physical routing of all the connections between the broken out components. A combination of hand routing and computer-driven auto routing was used to accomplish this, with the emphasis on hand routing.

The routing process employed two different auto routers, both the Board Station RE [46] and ICX's auto router. Neither router was able to handle traces that travel in arbitrary directions particularly well, which led to difficulties in completing routing. Both pieces of software used the common approach of routing signals using pairs of layers. The first layer would allow horizontal connections, while the second layer would provide vertical connections. If a signal needed to travel diagonally it would do so by first travelling horizontally and then transferring to the other layer, using a via, and finally, travelling vertically. This approach simplifies the search space for the auto router, but comes at the cost of a large number of vias.

The size constraints placed on the TM-4's PCB meant that it could not be routed using this layer pair approach, due to the number of vias it would require. Since each via is a hole that passes through a number of different layers of the PCB, each via prevents routing on all the layers it passed through. There were locations on the TM-4's PCB where the number of vias necessary to route using pairs would not have not fit in the space provided, while still allowing for routing room on internal layers.

The approach that was used to route the board was to come up with a high level floor plan of the desired PCB. This floor plan consisted of a description of what layers and regions of the circuit board certain signals should traverse. These regions were then individually routed by an auto router, when possible, and by hand, when not.

In total, the routing stage of the TM-4 took between 4 and 6 months.

4.6 PCB SIGNAL INTEGRITY SIMULATION

The circuit board is responsible for providing more than just connectivity between components; it must also meet certain timing and signal integrity requirements. Both timing issues, such as trace delay and skew, as well as signal integrity issues, such as crosstalk and termination, were considered when designing the circuit board for the TM-4. The way that these requirements were verified was through simulation.

The PCB simulation process used a combination of IBIS device models and Mentor's ICX tool to model and simulate PCB interconnects. The ICX tool allowed different signal integrity parameters, such as cross talk, overshoot, ringing, etc., and timing parameters, both minimum and maximum, to be verified.

To better describe the simulation procedure used in the TM-4, the DDR SDRAM subsystem will be used as a case study. The following three sections will examine the process used to design the DDR SDRAM portion of the PCB. The first section will examine how the design of the DDR SDRAM section of the PCB was driven by timing requirements. This will be followed by a description of the timing simulation used in designing the PCB to meet these requirements. The last section will examine signal integrity simulations that were used to verify the DDR SDRAM signal integrity requirements

4.6.1 DDR SDRAM TIMING-DRIVEN DESIGN

The DDR SDRAM standard uses a source-synchronous clocking scheme. This scheme depends on the skew between signals and as such, is dependent on both the minimum and maximum timing of a path. In order for the memory to work correctly, it was necessary to insure that all 72 data bits and 9 data strobe signal lines had as little skew as possible. This requirement translated into the restriction that the delay of each signal line needed to be closely matched.

The method used to achieve the goal of limiting skew was to iteratively perform timing simulations, to determine trace delays, and to use the result to change the lengths of the traces to more closely match. After a large number of iterations, the routing pattern shown in Figure 30 resulted.

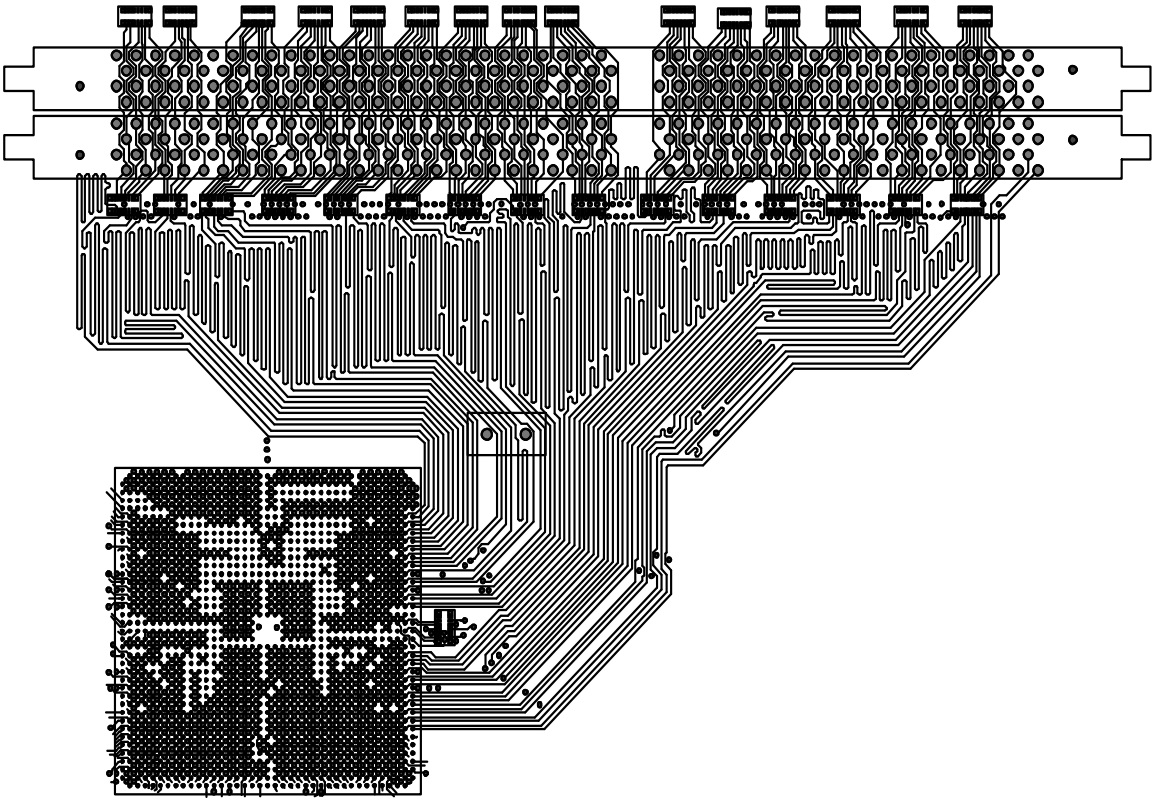


Figure 30: DDR Serpentine Delay Pattern

The component in the bottom left of the figure is one of the four development FPGAs. The two components at the top are the DDR SDRAM modules and the smaller components around the DDR SDRAM are termination resisters. The trace pattern shown has been tuned such that the delay along each trace is tightly matched. This tuning took the form of the serpentine patterns in which the trace backtracks to increase its length.

4.6.2 DDR SDRAM TIMING SIMULATION

The process of routing the DDR SDRAM signal traces, described in the previous section, required extensive use of delay simulations. The goal of the simulations was to determine the propagation delay from a pin on the FPGA to a pin on the DDR SDRAM for a given trace configuration. The information available for use in simulations was IBIS models of the FPGA and DDR SDRAM drivers and the trace configuration itself.

These two pieces of information were applied to a timing simulation tool called Mentor Graphics' ICX [46]. This tool had the ability to extract transmission line models

from the specified trace configuration and perform simulations using the IBIS models. The result of the timing simulation was a minimum and maximum delay number.

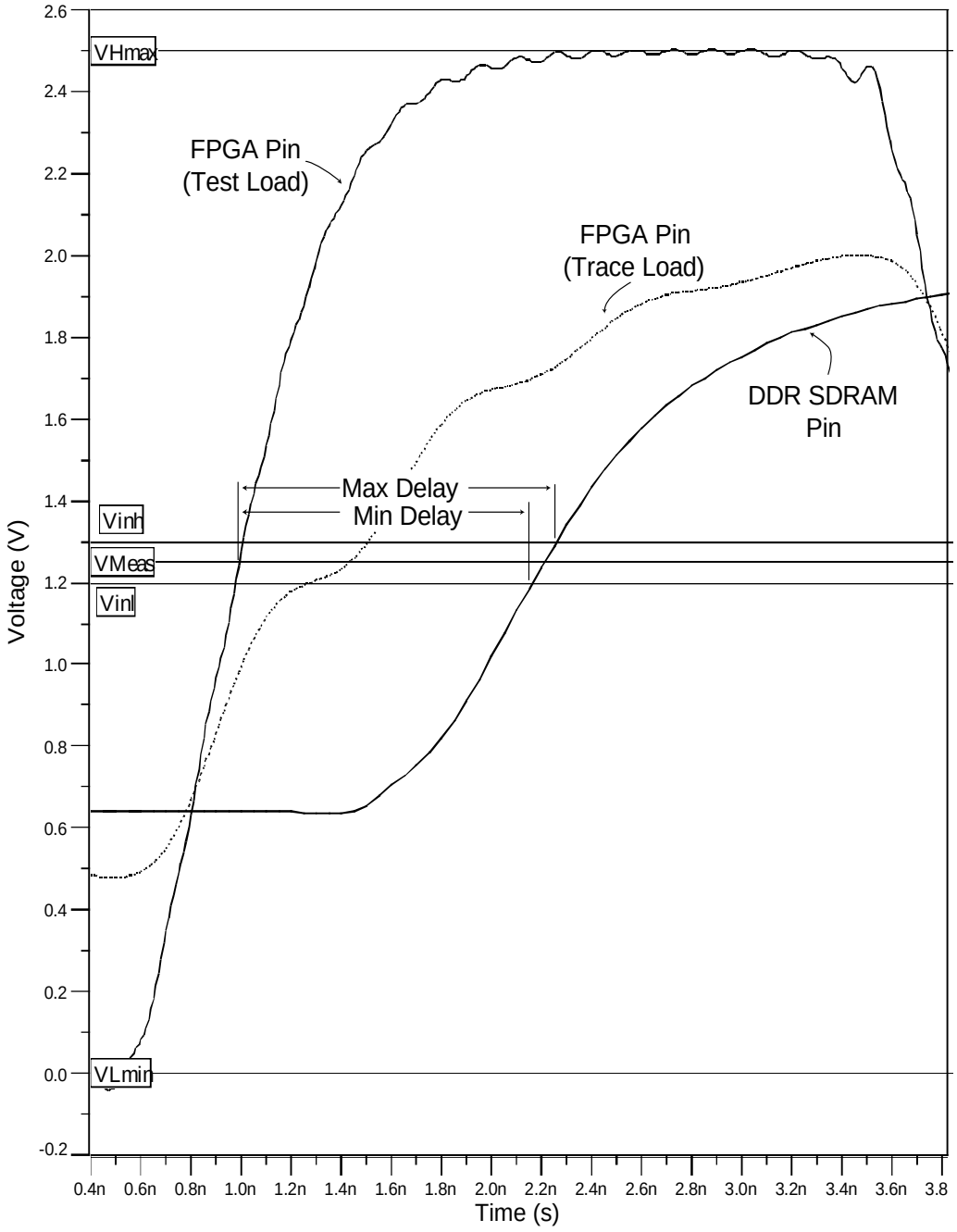


Figure 31: Sample DDR SDRAM Delay Simulation

Figure 31 shows an annotated result of a delay simulation for a DDR SDRAM trace. The graph plots the voltages at various points along the trace versus time. The horizontal lines in the graph represent thresholds specific to the signalling standard used

by the traces. In particular, V_{inl} and V_{inh} are the receiver's voltage input thresholds for both low and high values. The three voltage charted lines represent the voltages at three different points. The first line, labelled FPGA pin (test load), shows the voltage response of the driver IBIS model when connected to a standard test load. The next line, labelled FPGA pin (trace load), shows the voltage response of the driver IBIS model when connected to the extracted transmission line. The final line, labelled DDR SDRAM pin, shows the voltage response where the transmission line meets the DDR SDRAM pin.

The minimum delay is calculated from this graph by measuring the time between when the test load graph crosses the V_{meas} threshold, set at 50% of the voltage swing, until the DDR SDRAM pin crosses the V_{inl} threshold. Similarly, the maximum delay is calculated by measuring the time between the test load V_{meas} crossing and the DDR SDRAM pin V_{inh} threshold crossing.

The same process is also repeated for a falling edge transition to obtain the true maximum and minimum delays.

4.6.3 DDR SDRAM SIGNAL INTEGRITY SIMULATION

Signal integrity issues affect the functionality of a circuit by a number of different means. First, an improperly terminated transmission line can cause unwanted reflections, ringing, undershot and overshoot, which can corrupt a signal. Secondly, cross talk can, at worst, cause unwanted edges to be coupled into clock signals, or at best, change the delay along a trace. In order to insure that none of these problems were introduced in the design of the TM-4, three different simulation types were used. The first type verified that the transmission lines were properly terminated. The next type verified that coupling would not cause unwanted edges in clock signals and the final type verified that the affect of cross talk induced timing variations were within acceptable limits.

Verification of signal termination was accomplished through the use of the transmission line simulation functionality of Mentor Graphics' ICX [46] tool. This tool combined IBIS models, extracted transmission line data and termination resistor information together to perform a complete electrical simulation. The simulation generated a voltage versus time curve for the signal at the receiving end of the

transmission line and also automatically calculated the rise and fall times, the over and undershoot and a measurement of ringing. These parameters allowed for the verification of proper termination.

The second type of simulation that was performed was to verify that cross talk did not induce unwanted edges in edge sensitive signals. Once again, Mentor's ICX tool was used. The ICX tool has a function that allows coupled transmission line models to be extracted from a PCB layout. These coupled models are then combined with IBIS models and are used to determine the maximum voltage that is coupled into a trace. The method ICX uses to calculate cross talk for a given net, or target net, is to consider the group of nets that are coupled to the target net, called aggressor nets. ICX performs a time domain simulation where the aggressor nets are all switching at the same time and the resulting voltage change on the target net is observed. The resulting induced voltage value can then be checked against a noise margin for the trace to ensure that unwanted edges are not a problem.

The third type of simulation performed to verify the DDR SDRAM design was a coupled delay simulation. Cross talk from neighbouring traces affects the delay on a target trace because it induces additional voltage into the trace. The resulting voltage at the receiver pin is a superposition of the induced voltage and the voltage wave from the transmitter. If the two voltages are in phase, then they reinforce each other and cause the resulting edge to be faster, thereby reducing the time necessary for the voltage to cross the switching threshold of the receiver, which reduces the trace's propagation delays. Similarly, if the two voltages are out of phase then they will destructively interfere with each other. This results in a slower edge and an increased propagation delay. Mentor's ICX tools provided a simulation mode that combined a timing simulation with a cross talk simulation. The results were a minimum and maximum delay number for each trace.

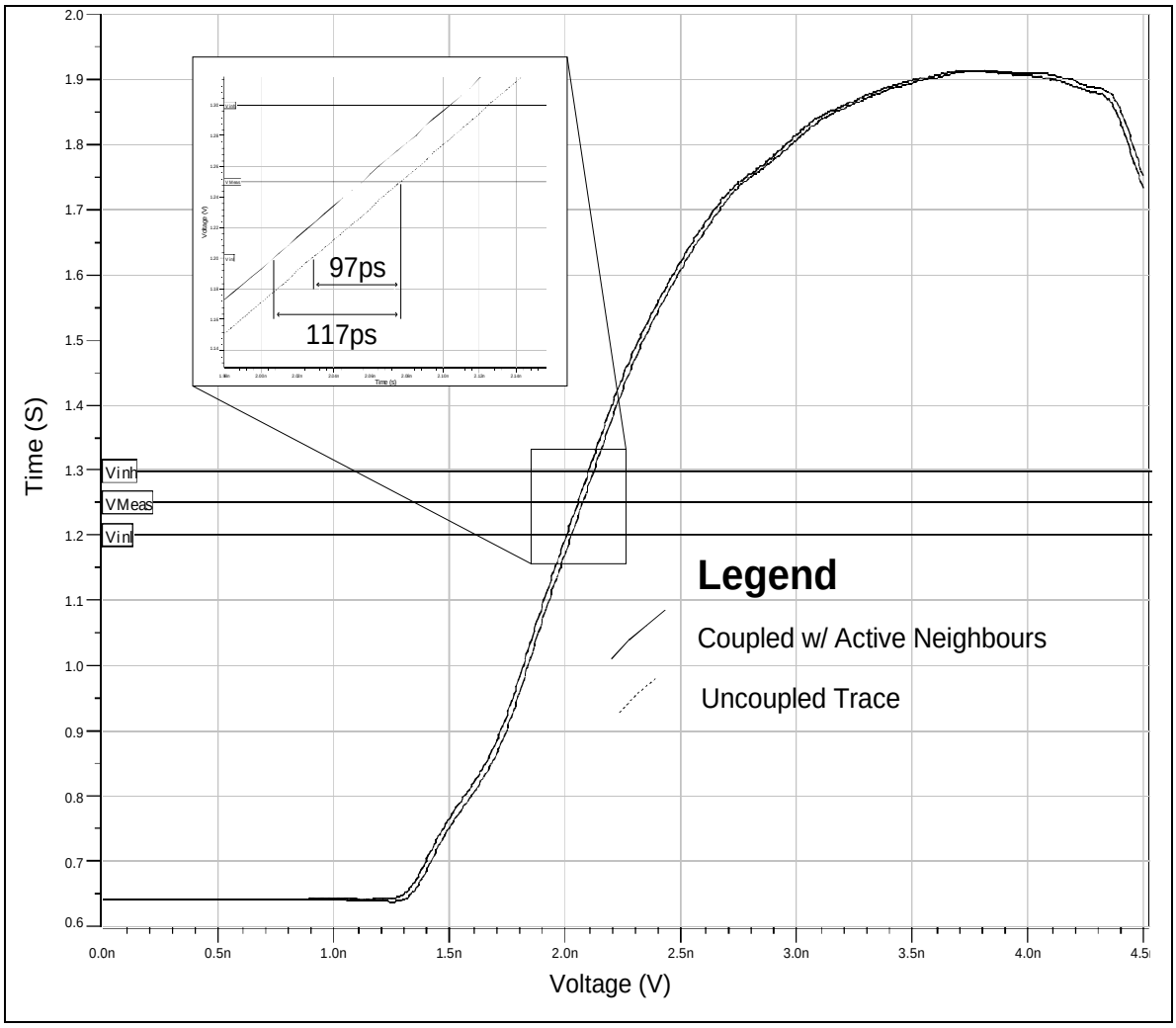


Figure 32: Sample Coupled Propagation Delay Simulation

Figure 32 shows a sample simulation result that illustrates the effect that coupling has on delay. The graph plots voltage versus time for the signal received at the end of a transmission line. The left most line shows the result of an uncoupled simulation, whereas the right most line shows the result of a coupled simulation, in which the neighbouring traces are switching in the same direction. The exploded view, of the portion of the graph where the waveforms cross the switching threshold of the receiver, clearly illustrates the effect of coupling. The coupled trace crosses into the switching threshold sooner then the uncoupled trace, by 20 ps. This results in the total time the signal remains in the switching region to increase from 97 ps, when only the uncoupled simulation is considered, and to 117 ps, when both coupled and uncoupled are considered. This results in a change of 20%.

When similar coupled delay simulations were performed on the TM-4's critical nets, it was found that the amount of induced uncertainty, usually in the range of 20-40 ps, was not sufficient to harm functionality.

4.7 SUMMARY

This chapter presented the final step of the TM-4's design process, the design of the printed circuit board. Each of the different phases of PCB design was discussed, including the stack up design, the component placement, the trace routing, and the signal integrity and timing validation. A case study of the design and simulation process involved in implementing the DDR SDRAM subsystem of the TM-4 was also presented.

The next chapter will examine the performance of an actual assembled prototype of the TM-4.

5 RESULTS

5.1 INTRODUCTION

The TM-4 was designed with the goals of having as much memory depth and bandwidth, inter-FPGA bandwidth and host-to-FPGA bandwidth as possible. This Chapter will describe how well the TM-4 system meets each of these goals and examine what factors affected the achieved system performance with respect to each goal.

Each of the major goals, memory performance, inter-FPGA performance, and host-to-FPGA performance, will be discussed in the following three sections.

5.2 MEMORY

There were two primary goals of the TM-4 in regards to memory. First, that the TM-4 must contain at least 4GB of memory and second, that the memory must provide as much bandwidth as feasible. The approach taken to meet this goal in the design of the TM-4 was to provide 8 independent 72bit DDR SDRAM modules. This allowed for the goal of memory capacity to easily be reached and provided a large amount of memory bandwidth.

Individual Module Capacity	Price Each	Total Board Memory	Total Price
256MB	$80	2GB	$640
512MB	$140	4GB	$1120
1024MB	$525	8GB	$4200
2048MB	$1200	16GB	$9600

Table 3: DDR SDRAM Module Capacity and Price

Table 3 shows the size of available DDR SDRAM memory modules that are usable in the TM-4. The table also shows the current market price, as listed by Crucial RAM [47], for each module, as well as the total cost and total memory capacity of the TM-4, if it was fully populated with 8 such modules.

The remaining goal of memory bandwidth will be examined in the following subsections. First, the theoretical maximum performance of the memory system will be examined, then a description of the experimental measurement procedure employed to determine the actual performance of the system will be presented and then the results will be discussed.

5.2.1 THEORETICAL MAXIMUM MEMORY PERFORMANCE

The performance of the memory subsystem of the TM-4 is theoretically limited by the bandwidth of the memory used and the ability of the FPGAs to communicate with the memory. Although DDR SDRAM is available with clock rates well above 200MHz, the FPGAs used in the TM-4 were only specified to run at 166MHz.

The performance of a 72bit 166MHz DDR SDRAM module is limited by the number of data transfers that it can perform per second. In this case, where each of these modules has a theoretical clock rate of 166MHz and can transfer data on both the rising and falling edges of the clock, the resulting theoretical maximum data transfer rate is 144 bits per clock cycle, or 2.8GB per second. If all eight modules are used simultaneously, the total available peak theoretical memory bandwidth is 22.9GB per second.

In practice, the sustained bandwidth will be somewhat lower due to the need for DRAM to stop transferring data while it receives a refresh command.

5.2.2 MEASURING ACTUAL MEMORY PERFORMANCE

Measuring memory performance is a very difficult task, due to the fact that the performance is significantly dependent on the data access patterns that the test uses. For example, a test consisting of reads to random addresses will return a relatively low bandwidth number because of the need to constantly switch pages, whereas a test that

reads only consecutive addresses will have a much higher bandwidth because of the fact that burst transactions can be used.

The access pattern selected for measuring the performance of the memory subsystem on the TM-4 was a block address pattern. In this pattern, an entire page of data, consisting of 32KBs, is read from the memory in one large burst. This pattern was an appropriate selection because the same pattern can be found in applications that work with data streams, such as video processing or genome searching.

To measure the actual performance of the TM-4's memory subsystem, a circuit was placed in the development FPGAs that implemented a DDR SDRAM controller, a simple memory test circuit, a timer and an interface to the host computer. The memory test circuit was designed to initiate 10,000 complete page memory transfers at the request of the host computer. At the same time that the memory test circuit was activated, the timer circuit would commence counting clock cycles. At the end of the test, the timer would stop and its value was then read back to the host computer.

5.2.3 ACTUAL MEMORY PERFORMANCE

Table 4 shows the memory performance results of a single DDR SDRAM module, as measured using the procedure described in the previous section. The actual measured performance of a single memory module in the TM-4 was 2.3GB per second; somewhat lower then the theoretical maximum of 2.8GB per second. The large discrepancy was due to the fact that timing mismatches between the delays of the DDR data traces prevented the memory from being run at a full 166MHz.

Clock Rate	Data Set Size	Transaction Time		Transfer Bandwidth
		Clock Cycles	Seconds	
133.3MHz	~351MB	20920132	0.16s	2.3 GB / Second

Table 4: Memory Bandwidth Results

Contrary to the simulations results from validating the design of the memory subsystem's PCB traces, the skew between data bits was large enough to violate necessary setup and hold time and prevented the system from operating at 166MHz. The

fact that the surface mount pins of the FPGA were not accessible for probing means, it was not possible to perform actual timing measurements on the traces. Ergo, without these measurements, it was not possible to determine the exact nature of the failure.

To combat the skew problem, the memory needed to be clocked at only 133MHz instead of 166MHz. However, even at this lower speed, the total memory bandwidth of the TM-4's eight memory modules reaches 17.6GB per second.

The effect of memory refresh and controller overhead can be examined by comparing the theoretical maximum transfer rate, at 133.3Mhz, to the actual measured transfer rate. The 2.30 GB/s actual measured memory bandwidth is only slightly lower then the 2.39 GB/s peak theoretical bandwidth. The measured value is only 4% below the theoretical peak value.

5.3 INTER-FPGA PERFORMANCE

One of the goals of the TM-4 was to provide as much communication bandwidth between the four development FPGAs as possible. The design of the TM-4 provides this bandwidth through the use of point-to-point buses between each pair of FPGAs. These buses are comprised of a combination of CMOS single-ended signals and LVDS differential signals. An actual detailed description of the bus architecture can be found in Section 3.3.2.2.

The theoretical maximum performance of these inter-FPGA buses will be examined in the following subsection. This is followed by a description of the experimental measurement procedure employed to determine the actual inter-FPGA bandwidth. A discussion of the results is then presented.

5.3.1 THEORETICAL MAXIMUM INTER-FPGA BANDWIDTH

Two things govern the theoretical maximum inter-FPGA bandwidth: the number of signals connecting each pair of FPGAs and the data rate that each signal can sustain. Between each pair of FPGAs there is either 20 or 40 high-speed LVDS communication channels, in addition to some single-ended signals, as illustrated in Figure 14 in Section

3.3.2.2. Each of these LVDS channels has a maximum data rate of 840 Mbps, or 105 MB per second, as specified by the manufacturer.

When taken together, the total aggregate LVDS bandwidth between each pair of FPGAs is either 2.1 or 4.2 GB/s, depending on the number of channels providing the connection.

5.3.2 MEASURING ACTUAL INTER-FPGA BANDWIDTH

The method used to determine the actual inter-FPGA bandwidth of the TM-4 was to measure the maximum data rate of a single LVDS channel and then to extrapolate this result to the entire set of channels. The procedure used to measure the data rate of a single channel consisted of a small test circuit running on two of the development FPGAs, under the control of software running on the host computer.

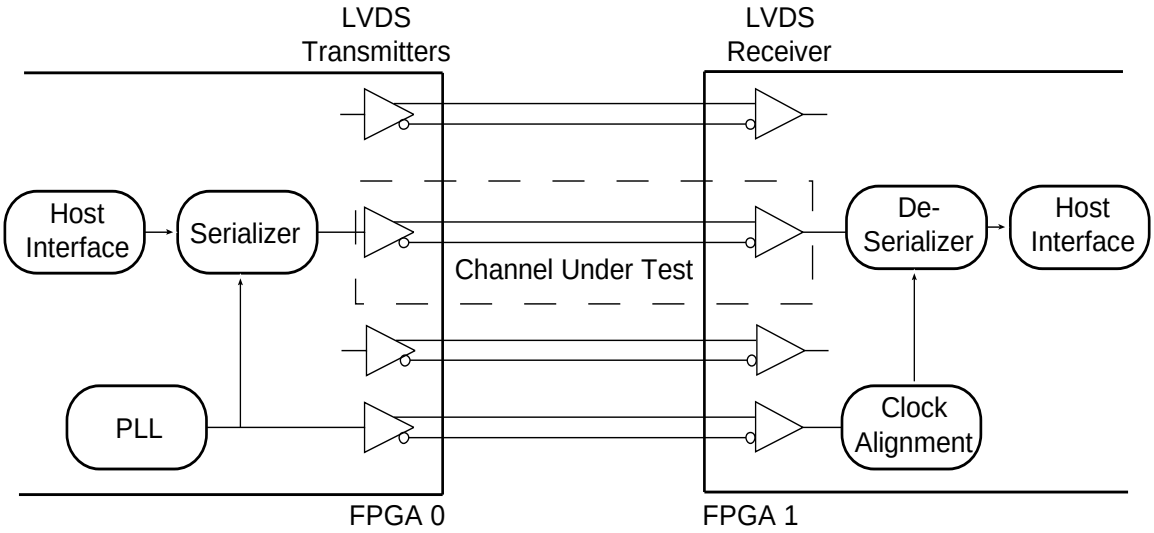


Figure 33: LVDS Performance Test Circuit

Figure 33 shows the test circuit used to measure the performance of an LVDS channel. The circuit consisted of all the components necessary to transmit data across an LVDS channel. This included serialization/deserialization hardware, a transmitter, a receiver and clock alignment hardware. An interface was provided that allowed for the host computer to provide an 8-bit test vector to the transmitter side of the LVDS channel, which it could then read back from the receiver side.

In order to further simulate the conditions of an LVDS channel in a real system, the LVDS channels that ran adjacent to the channel under test were also driven by the test circuit. The goal of this was to insure that any performance limiting effects of crosstalk were taken into account.

The testing procedure consisted of verifying the functionality of the test circuit at a set of different clock frequencies. For each frequency, every possible test vector was transmitted across the LVDS channel being validated, while the adjacent channels were being randomly driven. If the resulting received vectors were correct, the frequency was increased and the circuit was tested again. This process was repeated until the test finally failed. At this point, the maximum operating frequency of the LVDS link was revealed.

5.3.3 ACTUAL INTER-FPGA BANDWIDTH

The actual inter-FPGA communication bandwidth was measured with the procedure described above. The test circuit was found to be operational up to a data rate of 462 Mbps. Unfortunately, an error in the design of the clocking system for the LVDS channels meant that the Stratix FPGAs could not generate a clock with a sufficient edge rate necessary to transmit data faster than 462 Mbps. The problem arose from the fact that the Stratix architecture has two types of LVDS transmitters, slow transmitters and fast transmitters.

In order to use as many fast transmitters as possible, to transmit data, the design of the TM-4 called for a slower transmitter to be used to transmit the clock between FPGAs. The idea behind this was that since the clock transmitting between FPGAs is only a fraction of the data rate, it could be transmitted using a slow LVDS transmitter. The error in this approach was that the clock distribution architecture within the Stratix was not designed to provide a sufficiently low skew clock to these types of transmitters. This meant that in order to have an acceptable amount of skew between the clock and data transmitters, the data rate could not be higher than 462 MBps.

A related problem in the design of the LVDS inter-FPGA connections was that the clocking architecture, within the Stratix FPGA, could only drive half of the available

high-speed LVDS transmitters when it was also driving a slow speed transmitter. This meant that half of the LVDS channels could not be used at all.

Combining the 462 Mbps data rate with the 10 or 20 functional LVDS signals, between each FPGA, resulted in a total aggregate inter-FPGA bandwidth of 577 or 1155 MB per second, respectively. It is the intention of the author to fix this design error by changing the clock output from a slow to a fast LVDS transmitter in the next revision of the TM-4. This should then allow 19 LVDS channels to be functional, as one is now used for the clock, possibly up to their theoretical maximum data rate of 840 Mbps per channel.

5.4 HOST-TO-FPGA PERFORMANCE

The final key goal of the TM-4 was to provide as much host-to-FPGA communication bandwidth as feasible. The design of the TM-4 implemented this communication channel using the system described in Section 3.3.5. The channel consists of software running on a host computer that communicates with the TM-4 via a PCI bus that is bridged to the custom design development communication bus. The theoretical maximum performance of this channel will be examined in the next subsection. This is then followed by a description of the experimental measurement procedure employed to determine the actual channel bandwidth. The section concludes with a presentation and discussion of the results. a discussion of the results is presented.

5.4.1 THEORETICAL MAXIMUM HOST-FPGA BANDWIDTH

The host-to-FPGA communication channel consists of many different components, each of which has the potential to be the limiting factor of performance. At one end of the channel is software running on a Pentium III 1.4 GHz processor. This processor is connected to the interface FPGA by a 64-bit 66 Mhz PCI bus. The interface FPGA bridges the PCI bus to a local 32 bit 100 Mhz bus that is connected to the development FPGAs. The channel then ends with a logic circuit within the development FPGAs.

If we assume that both the host computer and the development FPGAs are able to generate and consume data at a sufficient rate that they are not the bottleneck, then the performance of the channel will be limited by the maximum data throughput of the two buses.

The maximum theoretical performance of a 64 bit 66 Mhz PCI bus is 528 MB per second. In practice, this number cannot be achieved due to the overhead of addressing, and the shared nature of the PCI bus.

The maximum theoretical performance of the 32 bit 100 Mhz local development bus is 400 MB per second. In practice, this level of performance can be sustained for writes from the host to the FPGAs, due to the fact that the bus utilizes separate command and data lines. However, the maximum performance of read from the FPGAs to the host will be less than the theoretical maximum, due to the overhead of issuing the read command.

When considered together, the local development bus will limit the theoretical maximum performance at a rate of 400 MB per second. However, in practice, the overhead of the PCI bus will likely be the limiting factor.

5.4.2 MEASURING ACTUAL HOST-TO-FPGA BANDWIDTH

There were two separate procedures used to measure the actual host-to-FPGA communication channel bandwidth. The first measured write bandwidth to the FPGAs from the host and the other measured read bandwidth from the FPGA to the host. Each procedure consisted of an identical hardware circuit, running on the development FPGAs, and a unique software component, running on the host computer.

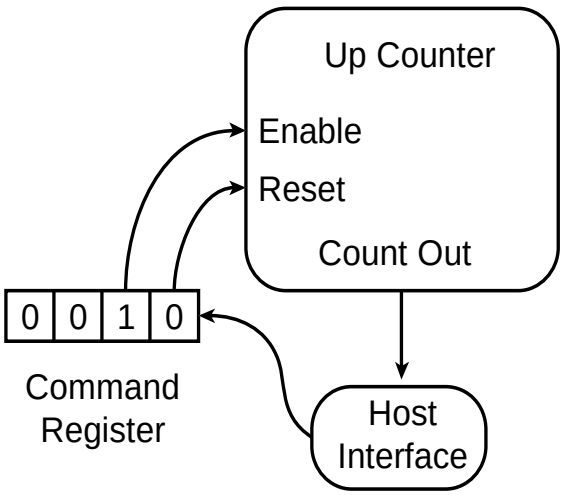


Figure 34: Host-To-FPGA Bandwidth Test Circuit

Figure 34 shows the simple hardware circuit involved in the test. The circuit consists of a resetable cycle counter, with a host computer interface. The interface allows the host computer to treat the circuit as a 32bit data sink, in the form of the command register, and a 32bit data source, in the form of the current count output. The software component of the testing procedure consisted of a program that would read or write a large amount of data to the TM-4 and use the cycle counter to measure the transfer time. In order to try to decouple the measurement from either the hard drive or network bandwidth limitations, the communication transactions were performed between the TM-4 and a buffer in the host computer's memory.

To test the write bandwidth from the host to the FPGAs, the software first writes a 32 bit reset command to the command register in the test circuit. This resets the cycle counter to zero. The software then issues the counter enable command to the command register, starting the cycle counter counting. Next, the software transfers 4 million more enable commands in one burst. The burst will not affect the counter as it is already enabled. Once the burst is complete, a single read of the counter's value is made. The resulting cycle count indicates the number of clock cycles it took to perform the 4 million 32bit command-burst. By combining this information with knowledge of the clock rate, a resulting transfer data rate can be calculated.

To test the read bandwidth from the FPGAs to the host, the software issues a reset command to the command register in the development FPGAs. This is then followed by

a single count enable command. The software then performs 4 million burst reads of the current cycle counter value from the hardware circuit. Upon completion of a fixed number of reads, the value of the cycle counter is read back and used to calculate the transfer data rate using the same method as for writes.

5.4.3 ACTUAL HOST-TO-FPGA BANDWIDTH

The actual host-to-FPGA communication channel bandwidth was measured using the procedure described above, using a test dataset of 125MB. Table 5 and Table 6 present the measured results, for both writing from the host to the FPGAs and reading from the FPGAs to the Host.

Run #	Dataset Size	Write Cycles	Write Time	Write Data Rate
1	125MB	49080384	0.49sec	267.04MB/s
2	125MB	49197584	0.49sec	266.40MB/s
3	125MB	49082848	0.49sec	267.03MB/s
			Average	266.8MB/s

Table 5: Measured Host-Write-To-FPGA Bandwidth

Run #	Dataset Size	Read Cycles	Read Time	Read Data Rate
1	125MB	84683792	0.85sec	154.78MB/s
2	125MB	84620688	0.85Sec	154.89MB/s
3	125MB	84712320	0.85Sec	154.73MB/s
			Average	154.8MB/s

Table 6: Measure Host-Read-From-FPGA Bandwidth

The first column indicates the test set run number, the second indicates the dataset size, and the third and fourth columns indicate the number of cycles and the time it took to transfer the data set, respectively.

The average bandwidth for writing data to the FPGAs from the host computer is 266 MB per second. This data rate is just over 50 percent of the peak bandwidth of the PCI bus. Through the use of a logic analyzer it was determined that the performance loss was due to two factors. The first factor was that the PCI bus was often left idle while the host computer's bridge chip retrieved data from the memory. The second factor was found to be software related. In order to perform a DMA transfer it was necessary to

have a fixed memory buffer in the host computer. However, due to Linux memory protection, it is not possible to have a user mode program directly access such a buffer. Instead, every byte of data needed to be transferred between a user-mode buffer and a kernel-mode DMA buffer. While this transfer was occurring, the PCI bus remained idle.

The effect of this memory protection problem could be reduced through the use of multiple buffers. While one buffer is being transferred across the PCI bus, via DMA, the other buffer could be copied between user and kernel space, effectively masking the cost of the copy.

The average bandwidth for reading data from the FPGAs to the host computer was found to be 154 MB/s. The data rate is somewhat below that of the writing rate, but is still respectable. Once again, a logic analyzer was used to determine the source of the performance loss. In this case, it was determined that the performance-limiting factor was the parameterizable bus interface logic core's ability to provide data to the local development bus. Although the development bus can sustain a transfer rate of 400 MB per second, the logic core could not. The reason for this is that the logic core uses a handshaking protocol to interface with the rest of the logic in the development FPGA. Figure 35 shows the waveform of the handshaking protocol.

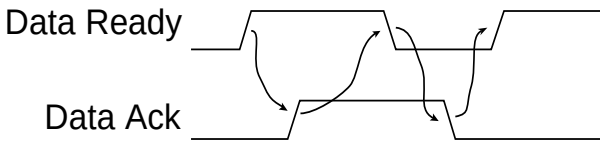


Figure 35: Handshaking Protocol

The handshaking protocol consists of a full, four-step handshake. This approach was taken, as it would allow a slow circuit to easily interface with a fast development bus. This benefit came with a performance cost, however, because there are four different handshaking steps that must be taken in order to transfer a single word of data. The handshaking protocol could be changed to provide higher communication bandwidth but would come at the cost of a more complicated user interface.

Overall, the performance of the host-to-FPGA communication channel is 266 MB per second for writes, and a very respectable 154 MB per second for reads.

5.5 SUMMARY

The three measurement procedures, presented in this chapter, show how the TM-4 design meets the goals of providing significant memory bandwidth, inter-FPGA bandwidth, and host-to-FPGA bandwidth. In total, the system has a measured memory bandwidth of 17.6 GB per second, an inter-FPGA LVDS communication channel bandwidth, between each pair of FPGAs, of up to 1.15 GB per second, and a host-to FPGA bandwidth of 266 MB per second for writes and 154 MB per second for reads.

6 CONCLUSIONS

6.1 SUMMARY

In this thesis, the design of an FPGA-based rapid prototyping system was presented. The objective of this work was to provide a development platform with as much memory capacity, memory bandwidth, inter-FPGA bandwidth and host-to-FPGA bandwidth as feasible. The resulting tests, on a physical prototype system, showed that the TM-4 was able to deliver large amounts of bandwidth in all of these categories. Table 7 summarizes each of the seven design steps undertaken in the creation of the TM-4, along with the approximate time that each stage took to complete.

It is the hope of this author that the creation of this prototyping system will enable future researchers to implement designs not possible with previous technologies.

Task	Time (Months)
Requirement Identification	4
Circuit Design	7
Placement	1
Routing	6
Verification	3
Software Design	1
Integration/Testing	5
Total	27

Table 7: Time Spent Working On Each Step Of TM-4 Design Process

6.2 CONTRIBUTIONS

This thesis provides the following significant contributions:

1. The design of an FPGA-based rapid prototyping system that provides:
    a. Multi-gigabyte memory capacity
    b. Significant memory bandwidth
    c. Significant inter-FPGA bandwidth
    d. Significant host-to-FPGA bandwidth
2. A design procedure for creating FPGA based systems
3. A verification procedure for validating large system designs

6.3 FUTURE WORK

There are a number of different ways that the performance of the TM-4 could be refined, both through hardware and software changes. The first step would clearly be to repair the design flaw that causes half the LVDS channels to not function and the other half to run much lower then their specified data rate. The other improvements would require more significant work, but could be implemented without changing the existing hardware of the TM-4.

The host-to-FPGA communication link runs at well below its theoretical maximum speed. In particular, retrieving data from the FPGAs runs quite slowly relative to its theoretical maximum speed. As noted, this is a result of the fact that protocols were designed to ensure easy use of the TM-4,while still providing relatively high performance. Further research and subsequent use of a better bus protocol could provide higher communication bandwidth, while maintaining a simple interface for users.

7 REFERENCES

[1] D. Galloway, D. Karchmer, D. Chow, D. Lewis, J. Rose, "*The Transmogrifier: The University of Toronto Field-Programmable System*," Second Canadian Workshop on Field-Programmable Devices, Kingston, June 1994.

[2] D. Lewis, D. Galloway, M. van Ierssel, J. Rose, P. Chow, "*The Transmogrifier-2: A 1 Million Gate Rapid Prototyping System*," in IEEE Transactions on VLSI, Vol. 6, No. 2, June 1998. pp 188-198.

[3] Transmogrifier 3A, University of Toronto, "http://www.eecg.toronto.edu/~tm3", Jan 2005.

[4] A. Darabiha, J. Rose, W. J. MacLean "*Video-Rate Stereo Depth Measurement on Programmable Hardware*," Proceedings of the 2003 IEEE Computer Society Conference on Computer Vision & Pattern Recognition, June 2003, Madison, Vol. 1, pp. 203-210.

[5] J. Fender, J. Rose, "*A High-Speed Ray Tracing Engine Built on a Field-Programmable System*," in IEEE International Conf. On Field-Programmable Technology, December 2003, pp. 188-195

[6] A. Alex, J. Rose, R. Isserlin-Weinberger, C. Hogue, "*Hardware Accelerated Novel Protein Identification*," in Int'l Symp. on Field-Programmable Logic, Aug 2004, pp. 13-22.

[7] Aptix's System Explorer, "http://www.aptix.com/products/mp4.htm", Jan 2005.

[8] Emulation and Verification Engineering's ZeBu-XL, "http://www.eve-team.com/zebu-xl.html", Jan 2005.

[9] Mentor Graphics' VStationPRO, "http://www.mentor.com/products/fv/emulation/vstation_pro/index.cfm", Jan 2005.

[10] Cadence's Palladium II, "http://www.cadence.com/products/functional_ver/palladiumII/index.aspx", Jan 2005.

[11] AMO GmbH's Venus-X Emulator, "http://www.amo.de/venus.html", Jan 2005.

[12] A. Lew, R. Halverson, Jr., "*Dynamic programming, decision tables, and the Hawaii parallel computer,*" Computer Math w/Applications, 1994.

[13] V. Salapura, M. Gschwind, O. Maischberger, "*A fast FPGA implementation of a general purpose neuron*". In Proc. of the Fourth International Workshop on Field Programmable Logic and Applications, Prag, Czech Republic, Sept 1994.

[14] Bee, "http://bwrc.eecs.berkeley.edu/Research/BEE/", Jan 2005.

[15] PCI Special Interest Group, "http://www.pcisig.com/", Jan 2005.

[16] A. Ferrucci, M. Martin, T. Geocaris, M. Schlag, P. K. Chan, "*ACME: A Field-Programmable Gate Array Implementation of a Self-Adapting and Scalable Connectionist Network*", 2nd International ACM/SIGDA Workshop on Field-Programmable Gate Arrays, 1994.

[17] T. A. Petersen, D. A. Thomae, D. E. Van den Bout. "*The AnyBoard: A Rapid-Prototyping System for Use in Teaching Digital Circuit Design,*" In Proceedings, The First IEEE International Workshop on Rapid System Prototyping RSP-90, Computer Society Press, 1991. pp. 25-32.

[18] K. Bouazza, J. Champeau, P. Ng, B. Pottier, and S. Rubini. "*Implementing cellular automata on the ArMen machine,*" In P. Quinton and Y. Robert, editors, Proceedings of the Workshop on Algorithms and Parallel VLSI Architectures II, Bonas, France, June 1991. pp. 317-322.

[19] P. K. Chan, M. Schlag, M. Martin, "*BORG: A Reconfigurable Prototyping Board Using Field-Programmable Gate Arrays*", Proceedings of the 1st International ACM/SIGDA Workshop on Field-Programmable Gate Arrays, 1992. pp. 47-51.

[20] W. Eatherton, T. Schiefelbein, H. Pottinger. "*An FPGA-based Reconfigurable Coprocessor Board Utilizing a Mathematics of Arrays,*" Technical report, University of Missouri--Rolla, Computer Science Department, 1995.

[21] G.M. Quenot, I.C. Kraljic, J. Serot, and B. Zavidovique. *"A Reconfigurable Compute Engine for Real-Time Vision Automata Prototyping,"* In IEEE Workshop on FPGAs for Custom Computing Machines, 1994. pp. 91-100

[22] H. Hogl, A. Kugel, J. Ludvig, R. Manner, K. H. Noffz, and R. Zoz. *"Enable++: A Second Generation FPGA-Processor for ATLAS,"* ATLAS internal note DQS-NO-026, CERN, 1994.

[23] L. Agarwal, M. Wazlowski, S. Ghosh. *"An asynchronous approach to efficient execution of programs on adaptive architectures utilizing FPGAs,"* In D. A. Buell and K. L. Pocek, editors, Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines, Napa, California, April 1994. pp. 101-110.

[24] M. Wazlowski, L. Agarwal, T. Lee, A. Smith, E. Lam, P. Athanas, H. Silverman, S. Ghosh. *"PRISM-II compiler and architecture"*. In Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines, Napa, California, April 1993. pp. 9-16.

[25] D. Smith and D. Bhatia. *"RACE: Reconfigurable and Adaptive Computing Environment,"* In 6th International Workshop 117 on Field-Programmable Logic and Applications, Darmstadt, Germany, September 1996. pp. 87-95.

[26] K. Oner, L. A. Barroso, S. Iman, J. Jeong, K. Ramamurthy, M. Dubois, *"The Design of RPM: An FPGA-based Multiprocessor Emulator,"* Proc. 3rd ACM International Symposium on Field-Programmable Gate Arrays (FPGA'95), Monterey, CA, February 1995.

[27] J.M. Arnold et al., "*The Splash 2 Processor and Applications*," Proc. Int'l Conf. Computer Design, CS Press, Los Alamitos, Calif.. 1993, pp. 482-485.

[28] R. Amerson, R. J. Carter, W. B. Culbertson, P. Kuekes, G. Snider, *"Teramac -- Configurable Custom Computing"*, Proceedings of the 1995 IEEE Symposium on FPGA's for Custom Computing Machines, 1995. pp 32-38.

[29] R. Mccready, "*Real-Time Face Detection on a Configurable Hardware Platform,"* M.A.Sc. Thesis, University of Toronto, 2000.

[30] A. G. Ye, D. M. Lewis, "*Procedural Texture Mapping on FPGAs*," ACM/SIGDA International Symposium on Field Programmable Gate Arrays, Monterey, CA, February 1999, pp. 112-120

[31] Hardi ASIC Prototyping System, “http://www.hardi.se/haps/haps.htm”, Jan 2005.

[32] Gidel’s PROCStar II, “http://www.gidel.com/PROCStar%20II.htm”, Jan 2005.

[33] Dini Group’s DN600K10, “http://www.dinigroup.com/index.php?product=DN6000k10”, Jan 2005.

[34] Annapolis Micro Systems’ WILDSTAR II Pro, “http://www.annapmicro.com/wsiipro.html”, Jan 2005.

[35] Spectrum Signal’s PRO 3100, “http://www.spectrumsignal.com/products/sdr/pro_3100.asp”, Jan 2005.

[36] T.J. Chaney, C.E. Molnar, “*Anomalous behavior of synchronizer and arbiter circuits,*” IEEE Transactions on Computers, vol. C-22, April 1973. pp.421-422.

[37] D.J. Kinniment, J.V. Woods, “*Synchronisation and arbitration circuits in digital systems,*” Proceedings of Institute of Electrical Engineers, vol.123, Oct. 1976. pp.961-966.

[38] H.J. Veendrick, “*The behavior of flip-flops used as synchronizers and prediction of their failure rate,*” IEEE Journal of Solid-State Circuits, vol. SC-15, April 1980. pp. 169-176.

[39] N. Azizi, I Kuon, A. Egier, A. Darabiha, and P. Chow, “*Reconfigurable Molecular Dynamics Simulator*”, Proceedings of the Internation Symposium on Field-Programmable Gate Arrays 2004, Feb 2004. pp. 190-199.

[40] TM-3 Ports Package Information, “http://www.eecg/~tm3/”, Jan 2005.

[41] The IEEE-1394 Trade Association, “http://www.1394ta.org/”, Jan 2005.

[42] Universal Serial Bus Information, “http://www.usb.org/”, Jan 2005.

[43] Gigabit Ethernet Alliance, “http://www.10gea.org/”, Jan 2005.

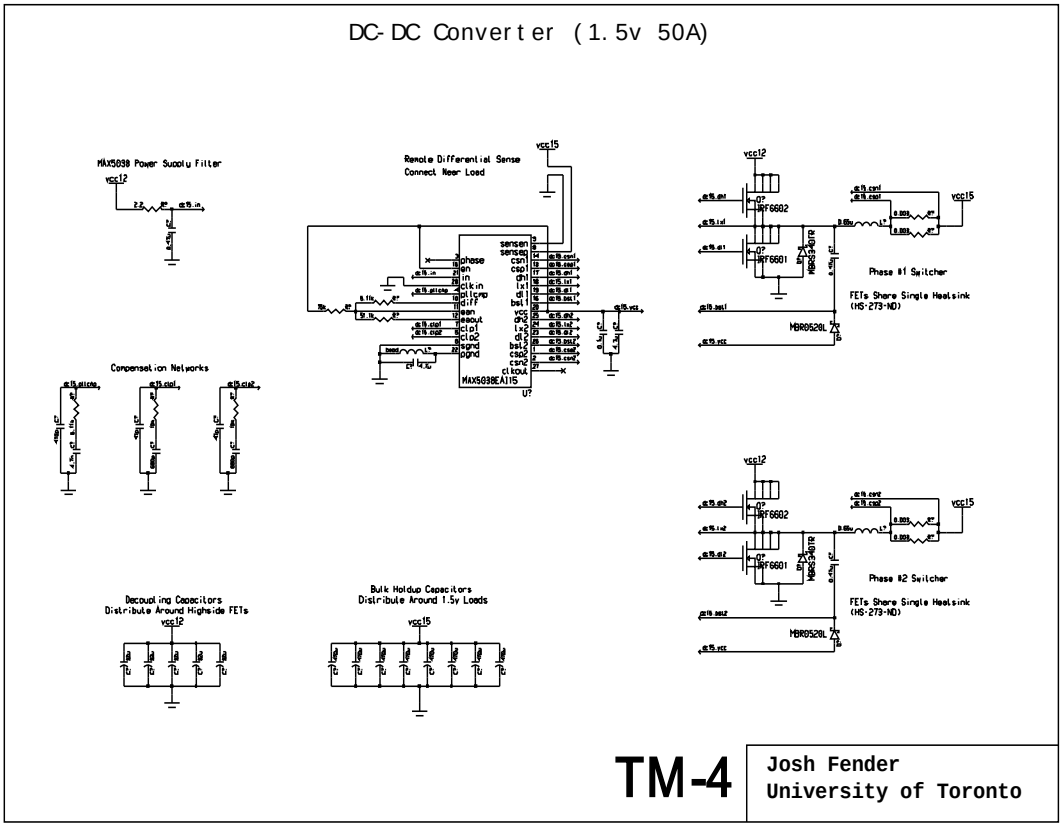
[44] Altera’s 64-Bit Master/Target IP Core, “http://www.altera.com/products/ip/iup/pci/m-alt-pci_mt64.html”, Jan 2005.

[45] H. Johnson and M. Graham, High-speed digital design: A handbook of black magic. New Jersey: Prentice Hall. 1993.

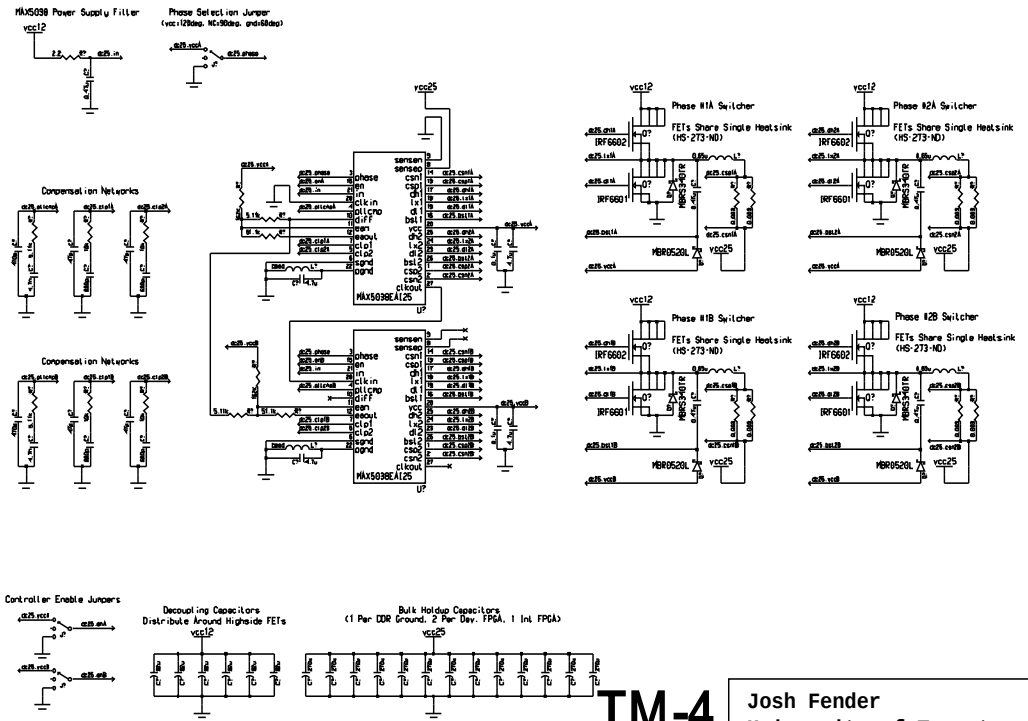
[46] Mentor Graphics, “http://www.mentor.com”, Jan 2005.

[47] Crucial RAM, “http://www.crucial.com”, Jan 2005.

A SCHEMATIC


DC-DC Converter (1.5v 50A)
MAX5038 Power Supply Filter
Remote Differential Sense
Connect Near Load
Compensation Networks
Decoupling Capacitors
Distribute Around Highside FETs
Bulk Holdup Capacitors
Distribute Around 1.5v Loads
Phase #1 Switcher
FETs Share Single Heatsink
(HS-273-ND)
Phase #2 Switcher
FETs Share Single Heatsink
(HS-273-ND)
MAX5038EAI15
IRF6602
IRF6601
MBR0520L
TM-4
Josh Fender
University of Toronto

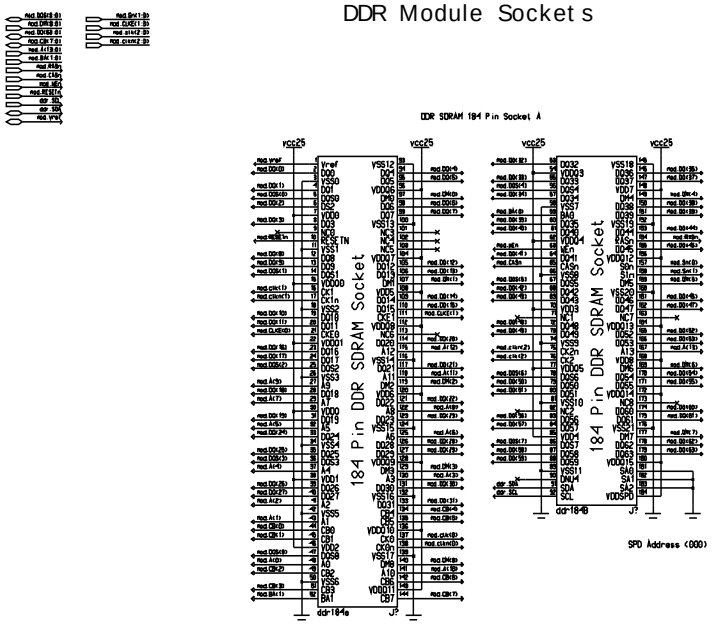
DC-DC Converter (2.5v 100A Peak)



TM-4

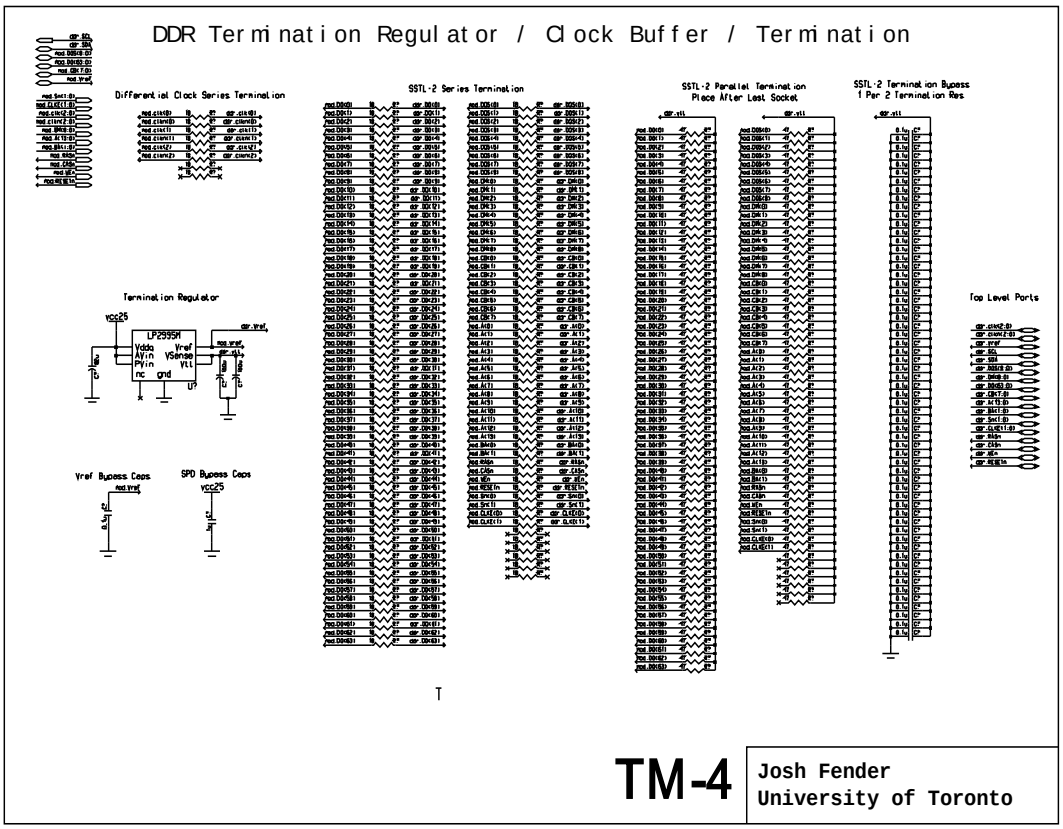
Josh Fender
University of Toronto

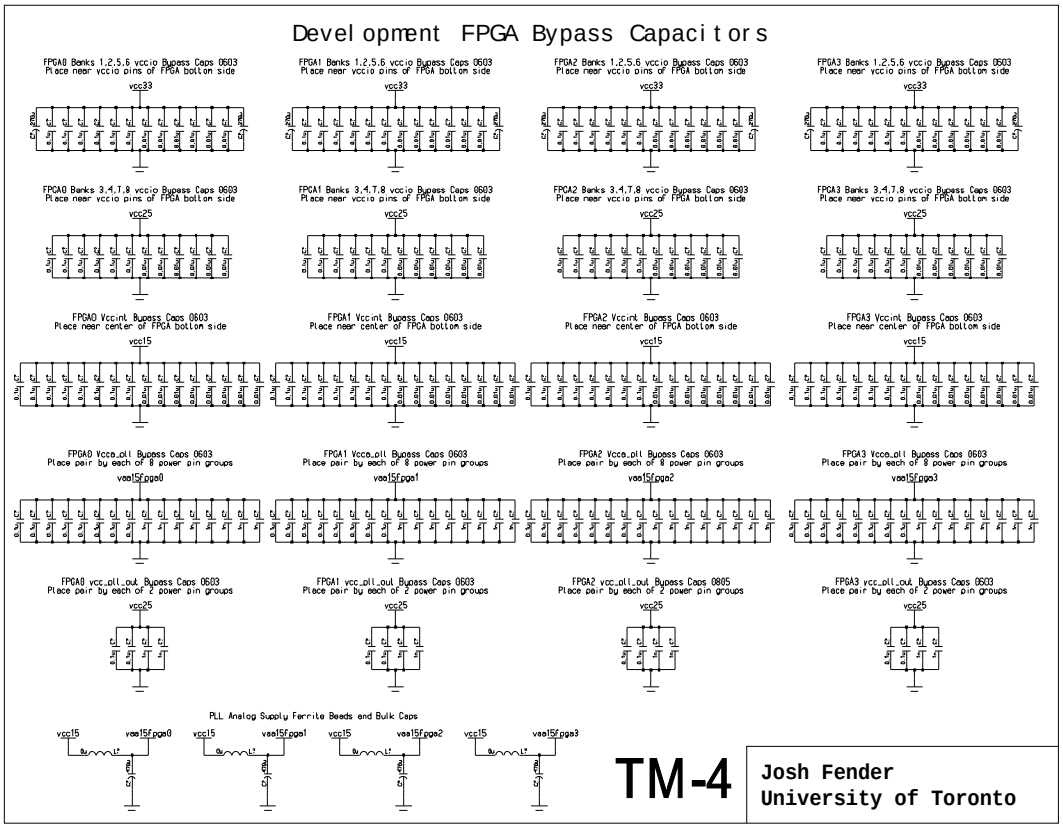
DDR Module Sockets



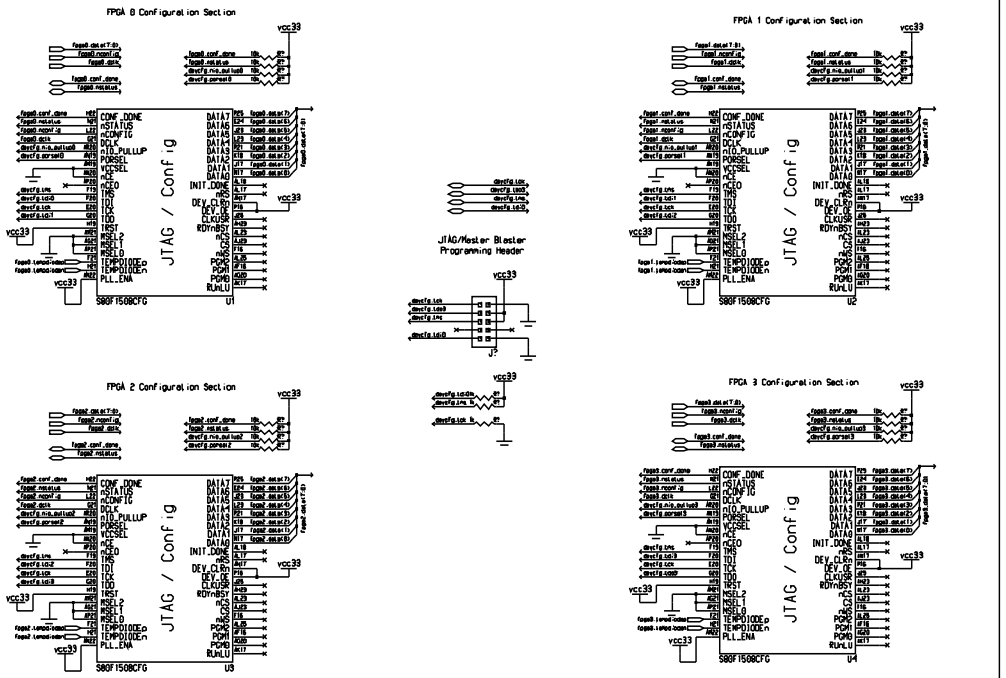
TM-4

Josh Fender
University of Toronto


DDR Termination Regulator / Clock Buffer / Termination
Differential Clock Series Termination
SSTL-2 Series Termination
SSTL-2 Parallel Termination
Place After Last Socket
SSTL-2 Termination Bypass
1 Per 2 Termination Res
Termination Regulator
LP2995M
Vddq
AVin
PVin
nc
gnd
Vref
VSense
Vtt
Vref Bypass Caps
SPD Bypass Caps
Top Level Ports
TM-4
Josh Fender
University of Toronto


Development FPGA Bypass Capacitors
FPGA0 Banks 1,2,5,6 vccio Bypass Caps 0603
Place near vccio pins of FPGA bottom side
FPGA1 Banks 1,2,5,6 vccio Bypass Caps 0603
Place near vccio pins of FPGA bottom side
FPGA2 Banks 1,2,5,6 vccio Bypass Caps 0603
Place near vccio pins of FPGA bottom side
FPGA3 Banks 1,2,5,6 vccio Bypass Caps 0603
Place near vccio pins of FPGA bottom side
FPGA0 Banks 3,4,7,8 vccio Bypass Caps 0603
Place near vccio pins of FPGA bottom side
FPGA1 Banks 3,4,7,8 vccio Bypass Caps 0603
Place near vccio pins of FPGA bottom side
FPGA2 Banks 3,4,7,8 vccio Bypass Caps 0603
Place near vccio pins of FPGA bottom side
FPGA3 Banks 3,4,7,8 vccio Bypass Caps 0603
Place near vccio pins of FPGA bottom side
FPGA0 Vccint Bypass Caps 0603
Place near center of FPGA bottom side
FPGA1 Vccint Bypass Caps 0603
Place near center of FPGA bottom side
FPGA2 Vccint Bypass Caps 0603
Place near center of FPGA bottom side
FPGA3 Vccint Bypass Caps 0603
Place near center of FPGA bottom side
FPGA0 Vcca_pll Bypass Caps 0603
Place pair by each of 8 power pin groups
FPGA1 Vcca_pll Bypass Caps 0603
Place pair by each of 8 power pin groups
FPGA2 Vcca_pll Bypass Caps 0603
Place pair by each of 8 power pin groups
FPGA3 Vcca_pll Bypass Caps 0603
Place pair by each of 8 power pin groups
FPGA0 vcc_pll_out Bypass Caps 0603
Place pair by each of 2 power pin groups
FPGA1 vcc_pll_out Bypass Caps 0603
Place pair by each of 2 power pin groups
FPGA2 vcc_pll_out Bypass Caps 0805
Place pair by each of 2 power pin groups
FPGA3 vcc_pll_out Bypass Caps 0603
Place pair by each of 2 power pin groups
PLL Analog Supply Ferrite Beads and Bulk Caps
TM-4
Josh Fender
University of Toronto

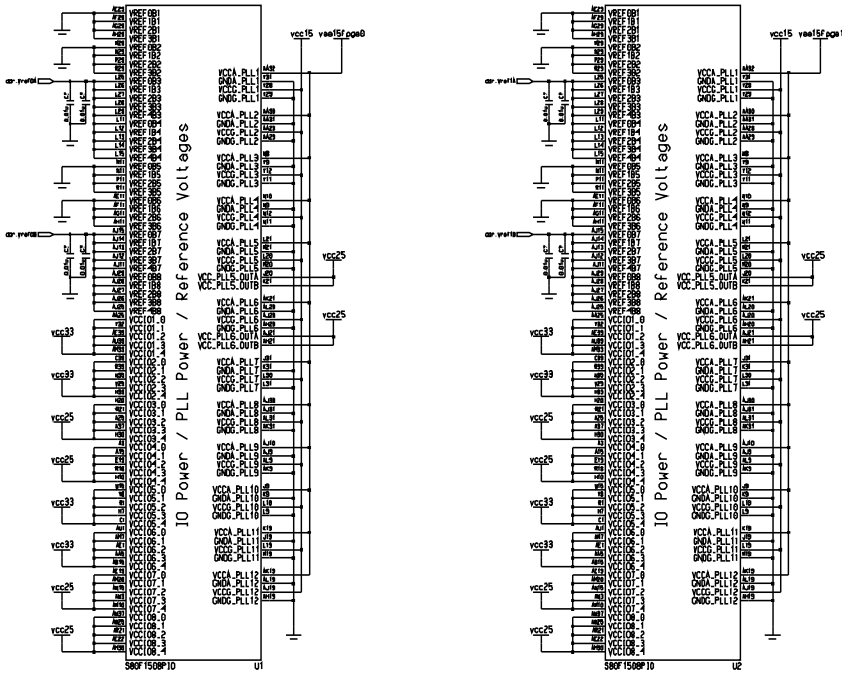
Development FPGA Configuration



TM-4

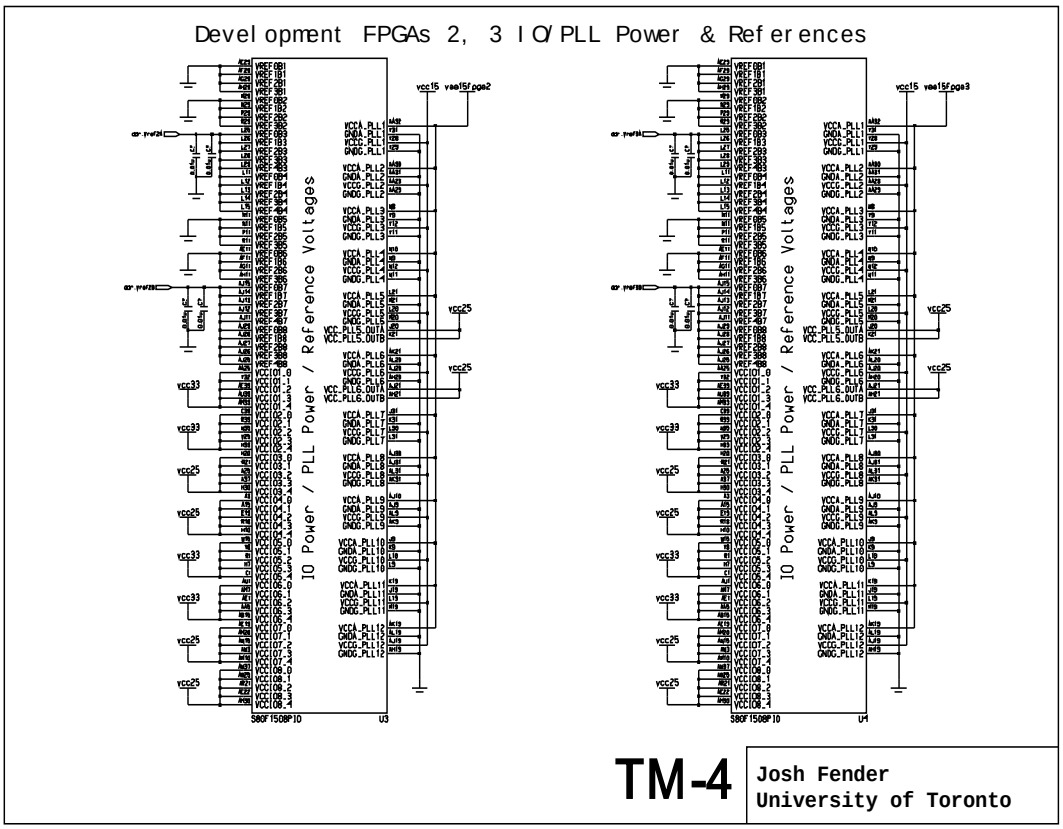
Josh Fender
University of Toronto

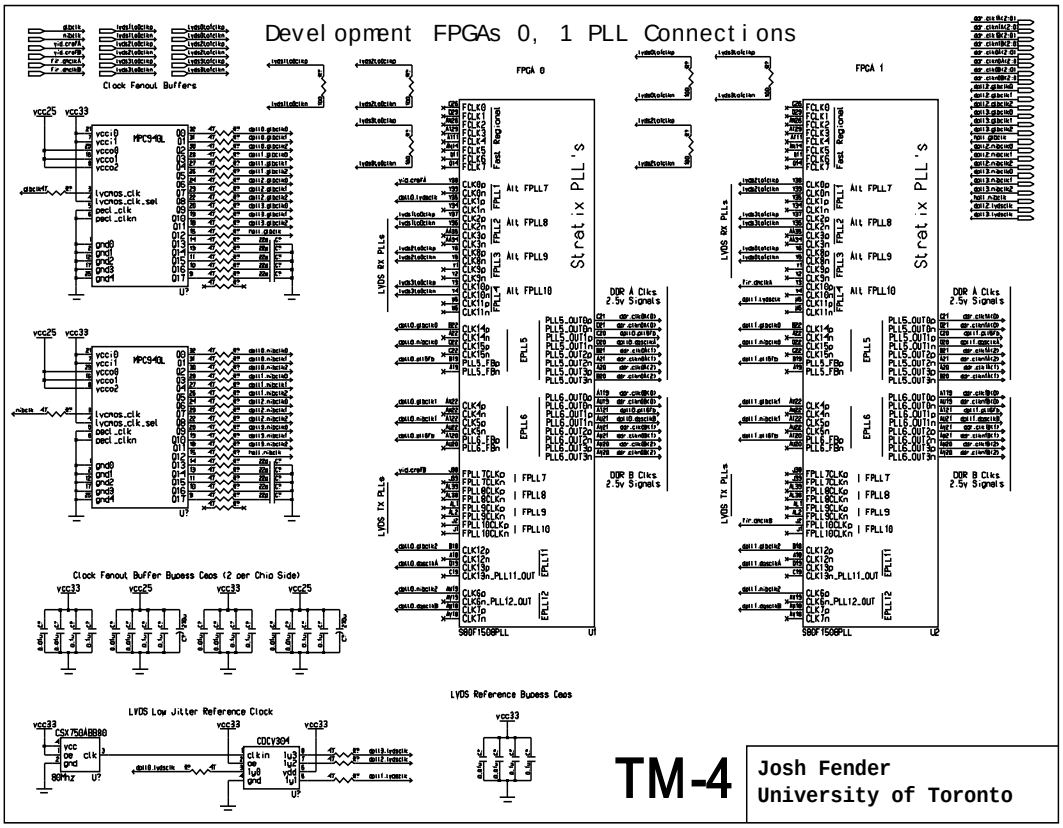
Development FPGAs 0, 1 IO/PLL Power & References

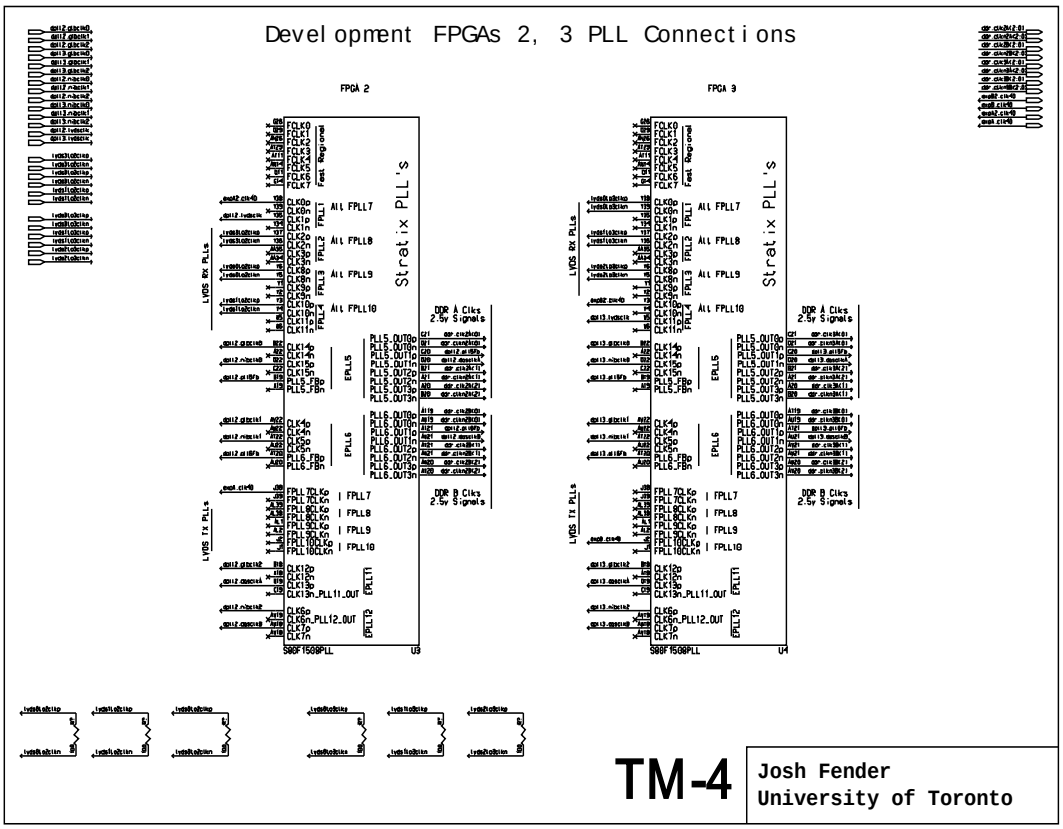


TM-4

Josh Fender
University of Toronto

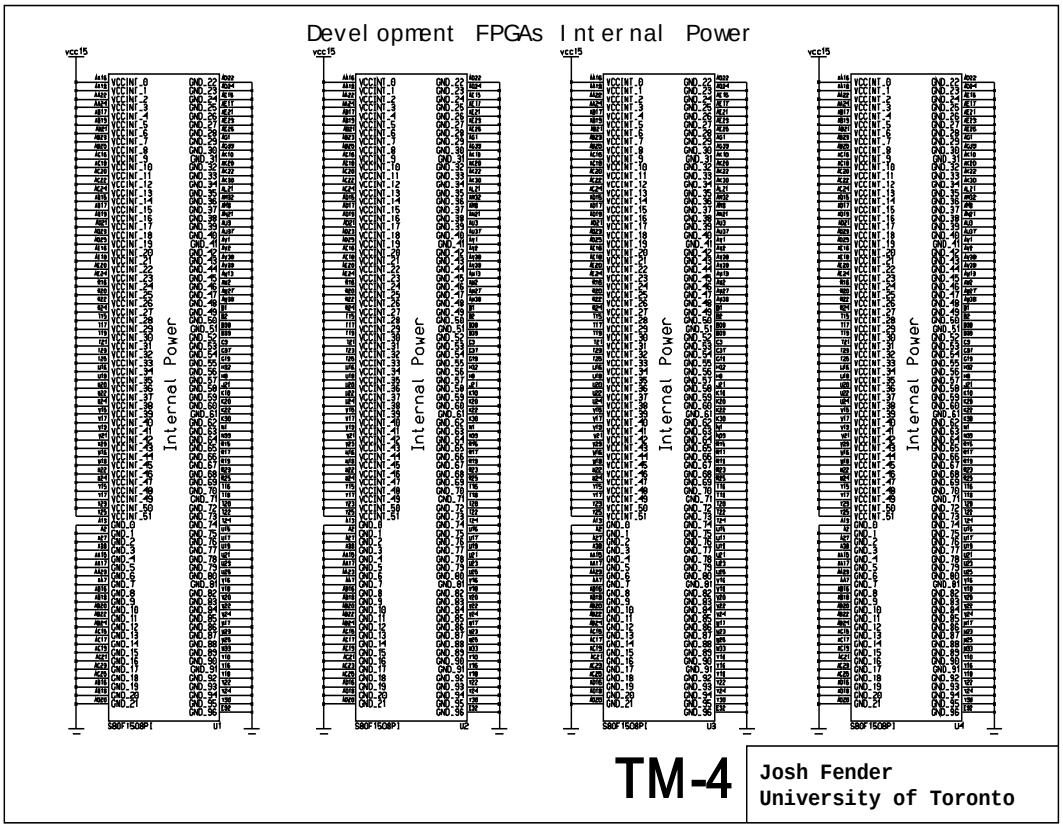

Development FPGAs 2, 3 IO/PLL Power & References
IO Power / PLL Power / Reference Voltages
IO Power / PLL Power / Reference Voltages
TM-4
Josh Fender
University of Toronto


Development FPGAs 0, 1 PLL Connections
Clock Fanout Buffers
Clock Fanout Buffer Bypass Caps (2 per Chip Side)
LVDS Low Jitter Reference Clock
LVDS Reference Bypass Caps
FPGA 0
FPGA 1
Stratix PLL's
TM-4
Josh Fender
University of Toronto



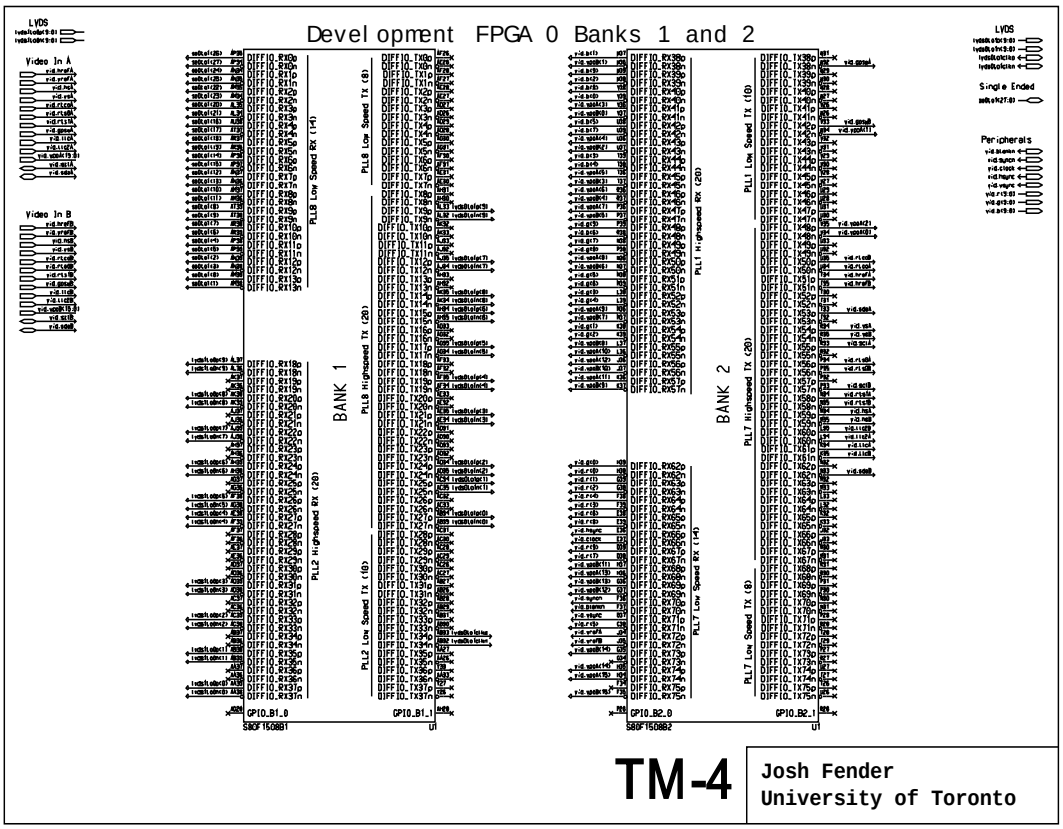
TM-4

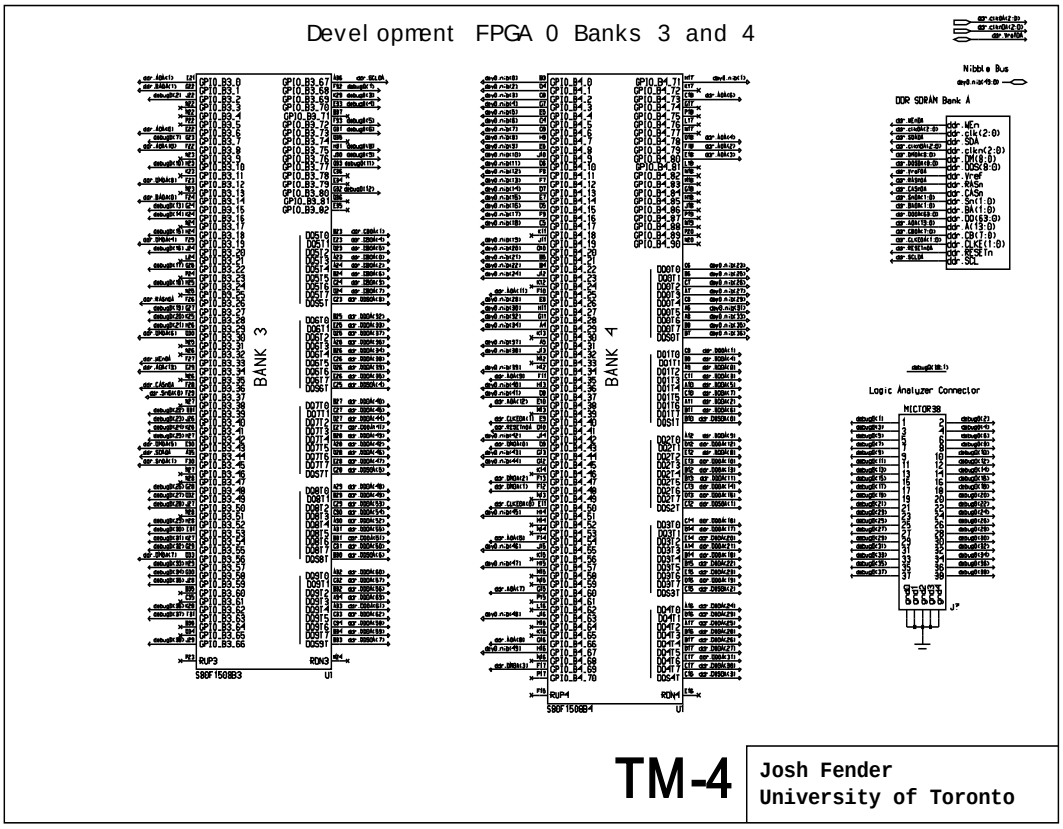
Josh Fender
University of Toronto

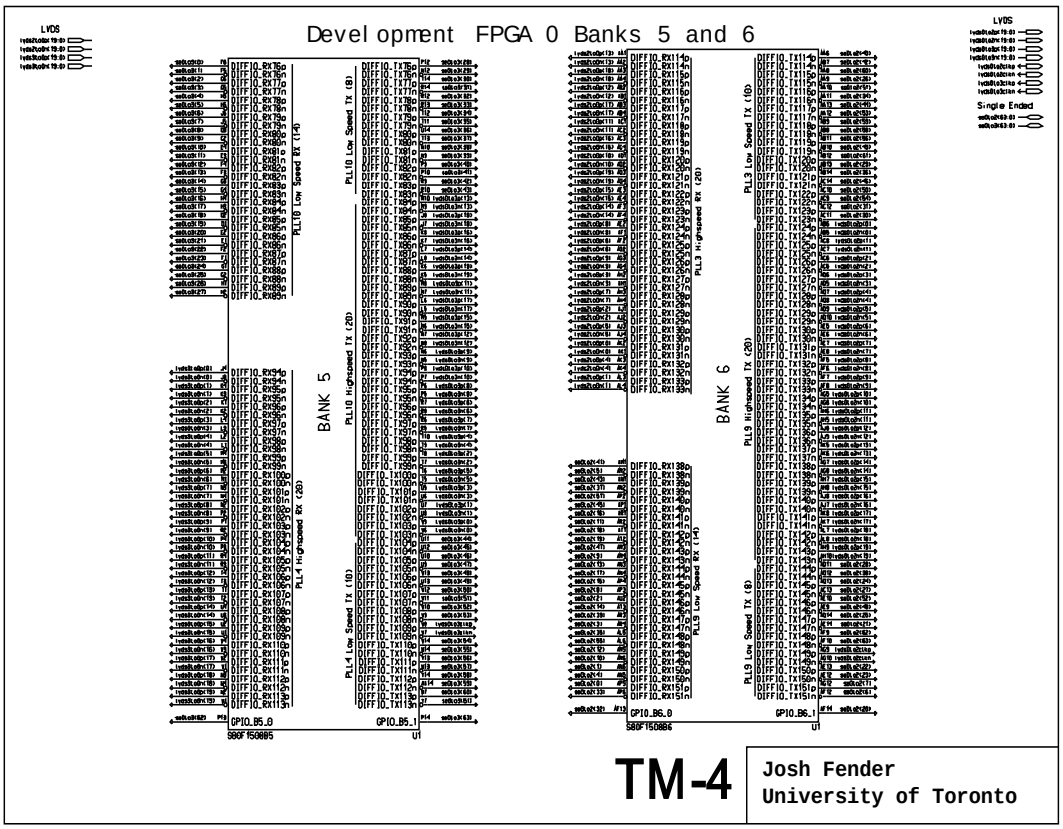


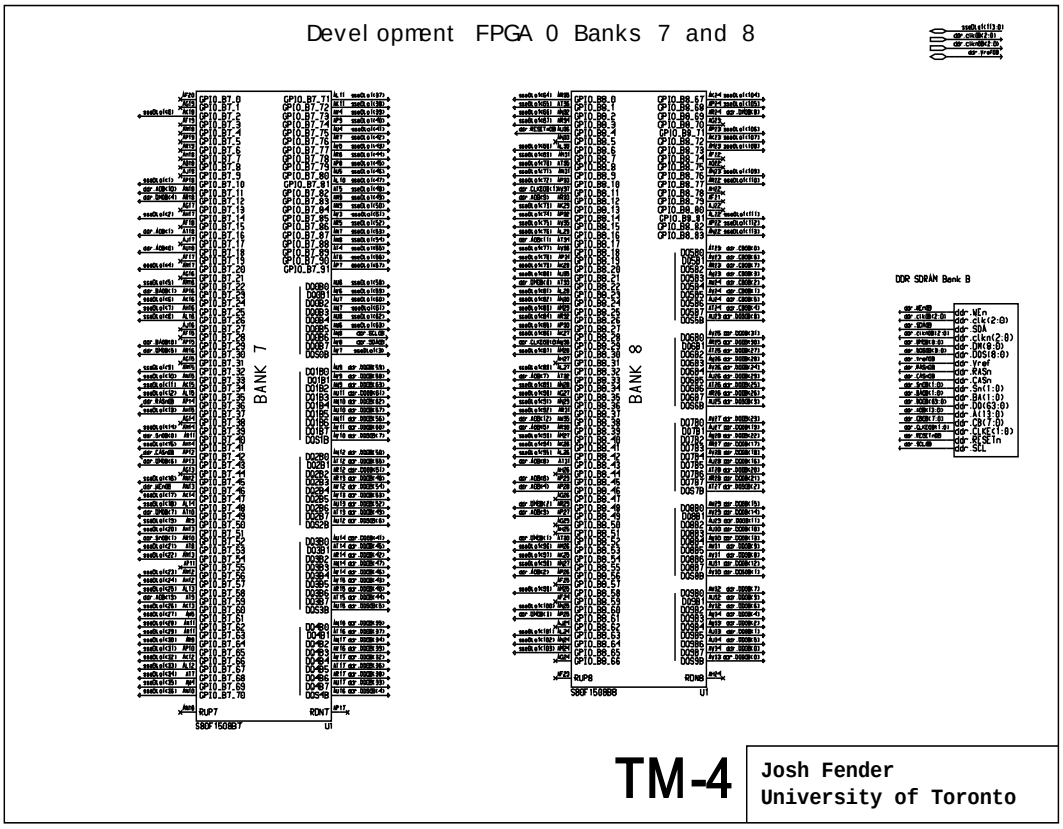
TM-4

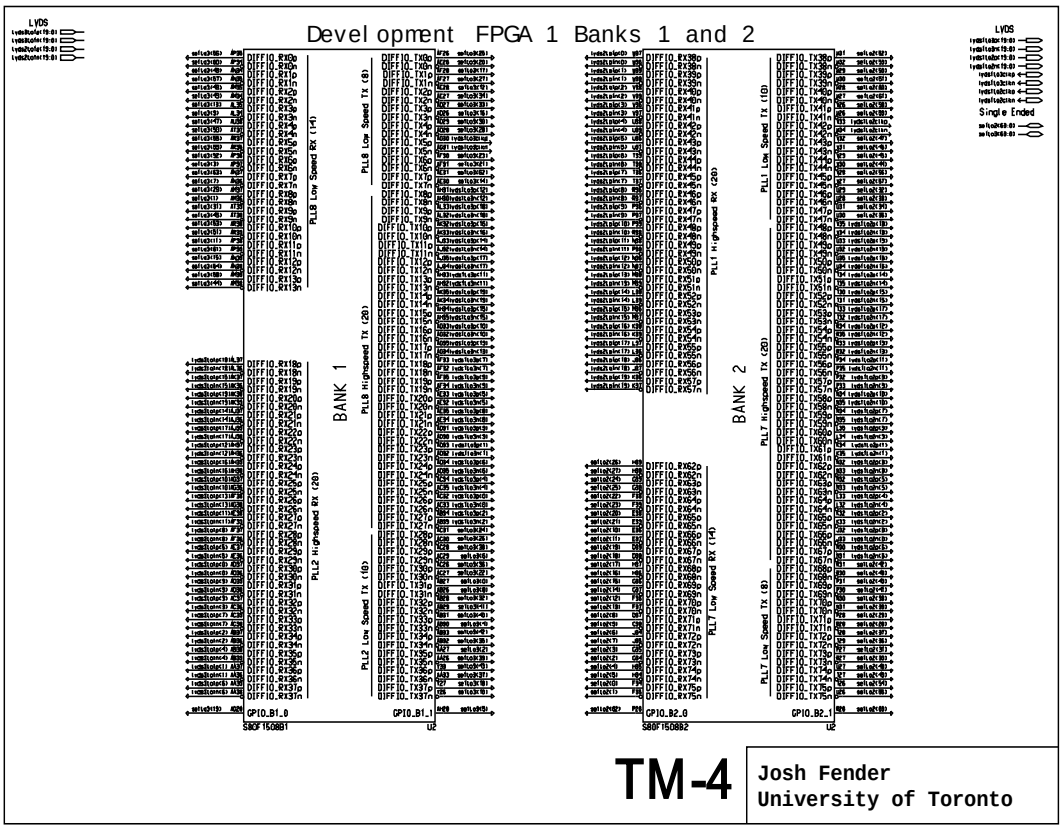
Josh Fender
University of Toronto

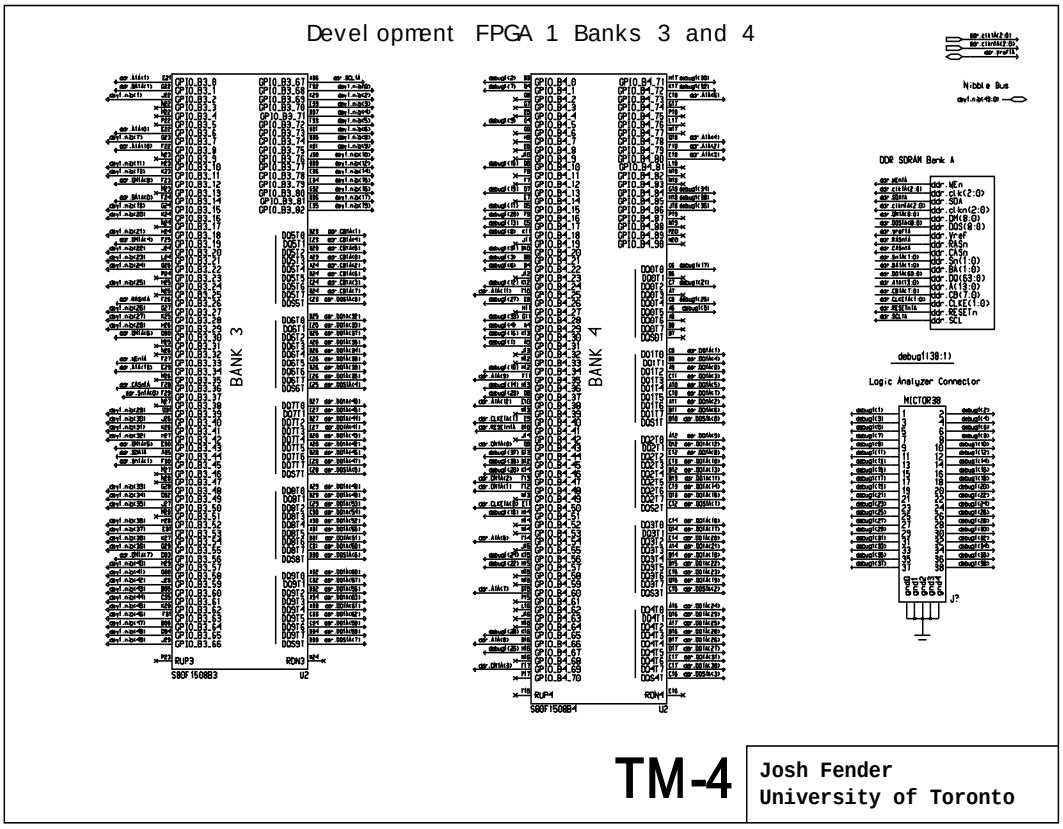

Development FPGA 0 Banks 1 and 2
LVDS
Video In A
Video In B
BANK 1
BANK 2
Single Ended
Peripherals
TM-4
Josh Fender
University of Toronto

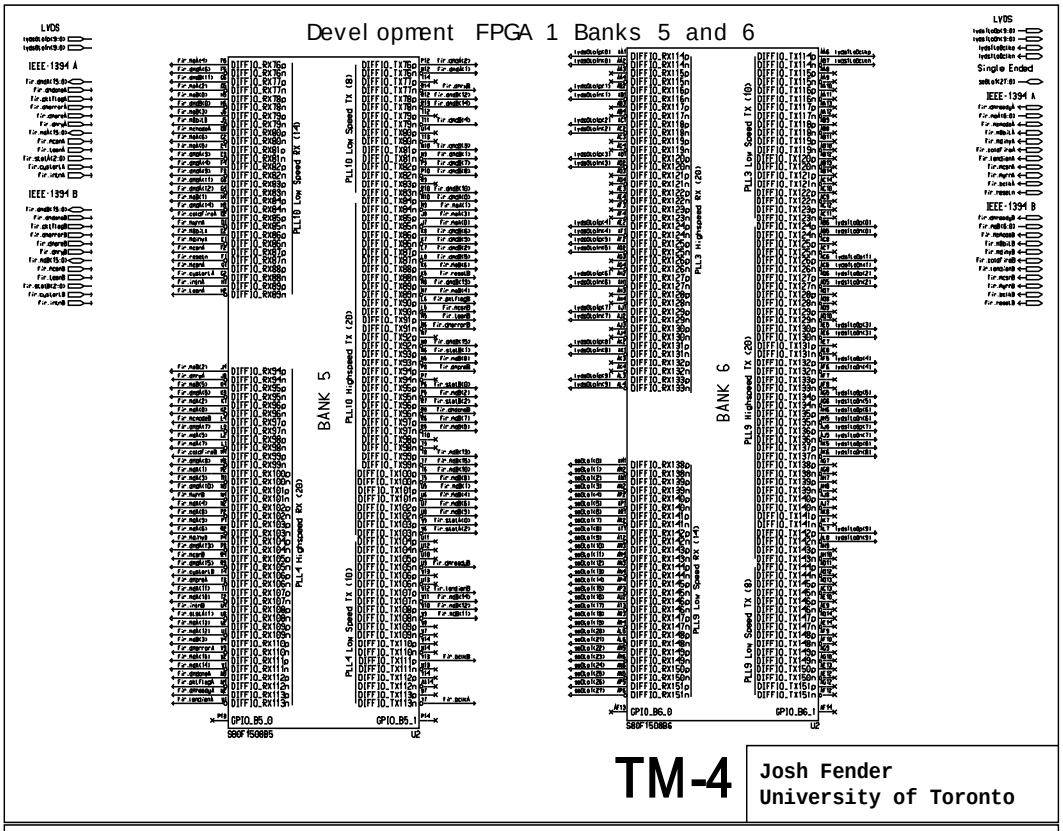

Development FPGA 0 Banks 3 and 4
BANK 3
BANK 4
Nibble Bus
DDR SDRAM Bank A
Logic Analyzer Connector
TM-4
Josh Fender
University of Toronto

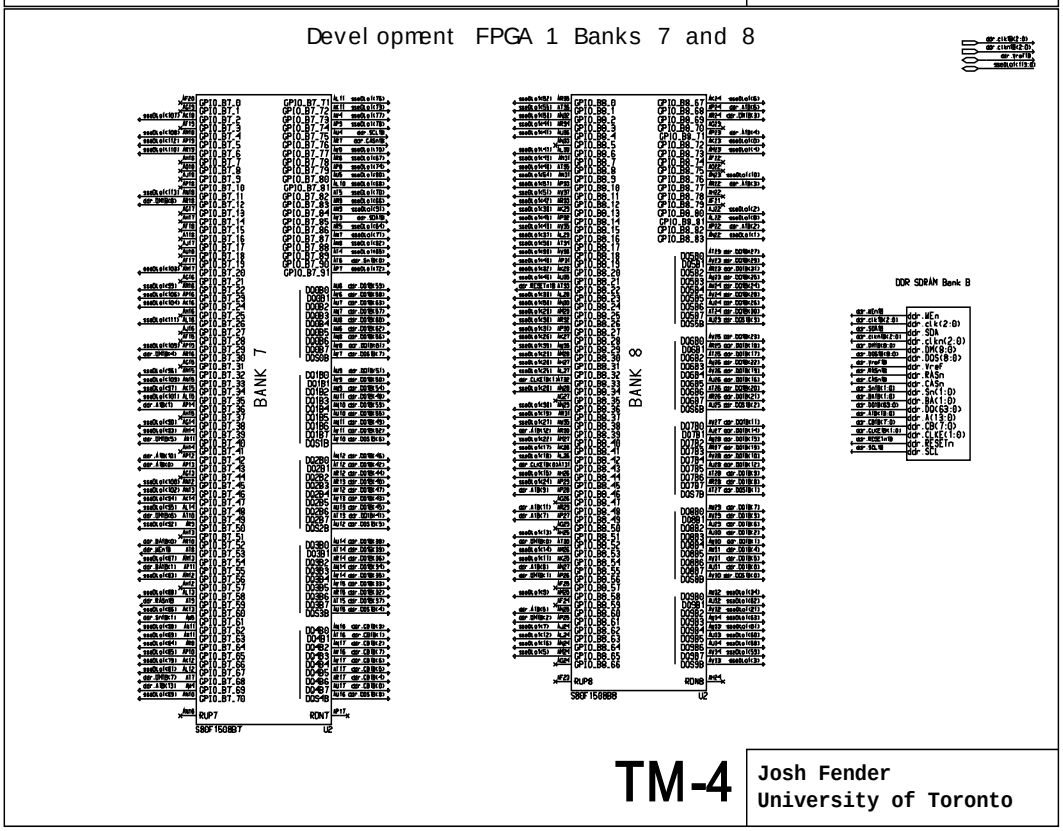

Development FPGA 0 Banks 5 and 6
LVDS
Single Ended
BANK 5
BANK 6
TM-4
Josh Fender
University of Toronto

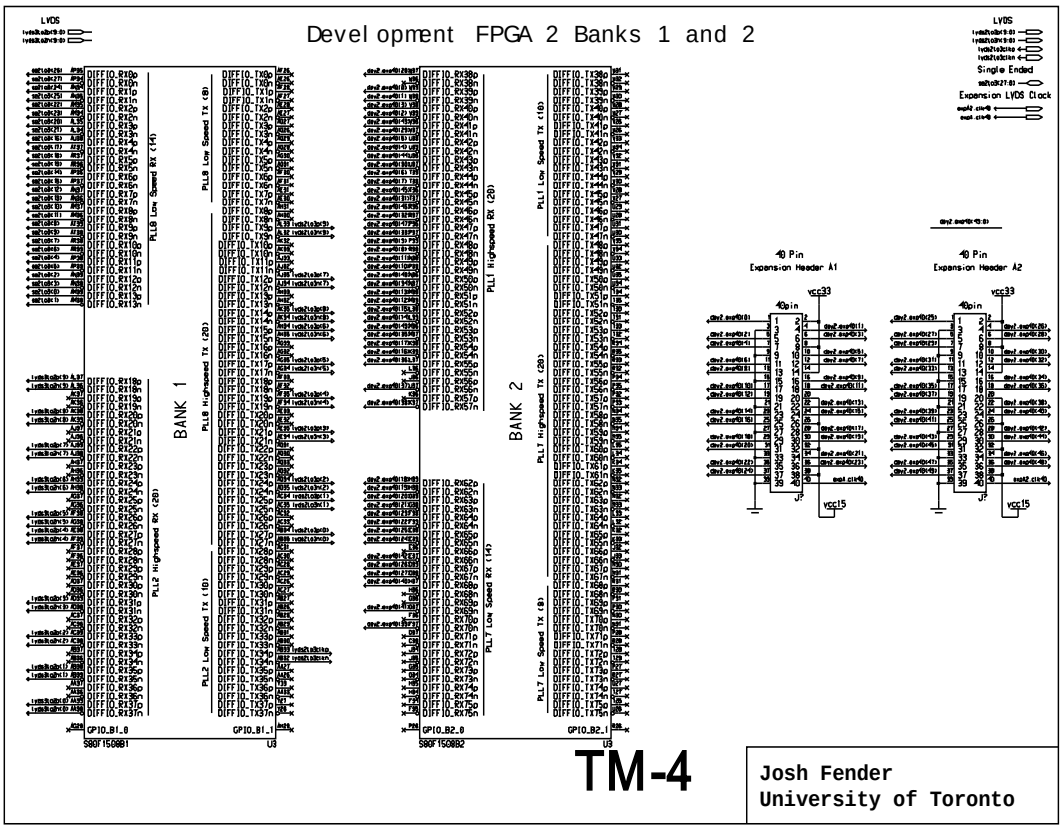

Development FPGA 0 Banks 7 and 8
DDR SDRAM Bank B
BANK 7
BANK 8
TM-4
Josh Fender
University of Toronto

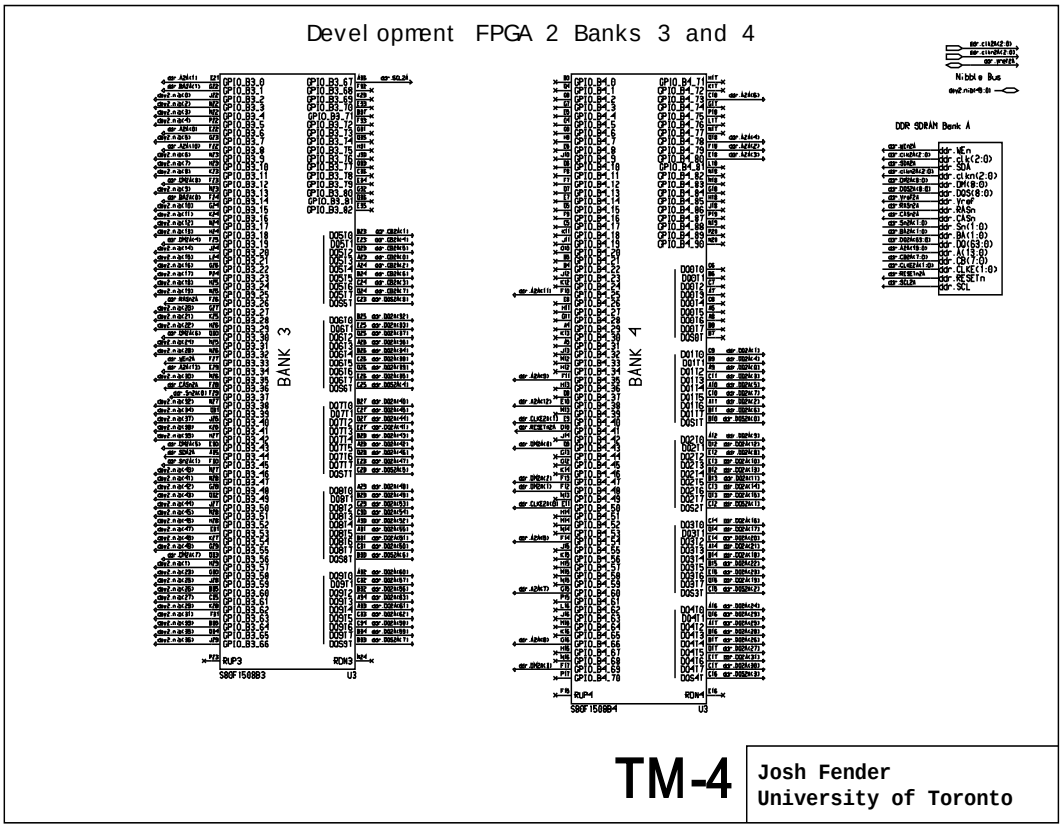

Development FPGA 1 Banks 1 and 2
LVDS
Single Ended
BANK 1
BANK 2
TM-4
Josh Fender
University of Toronto

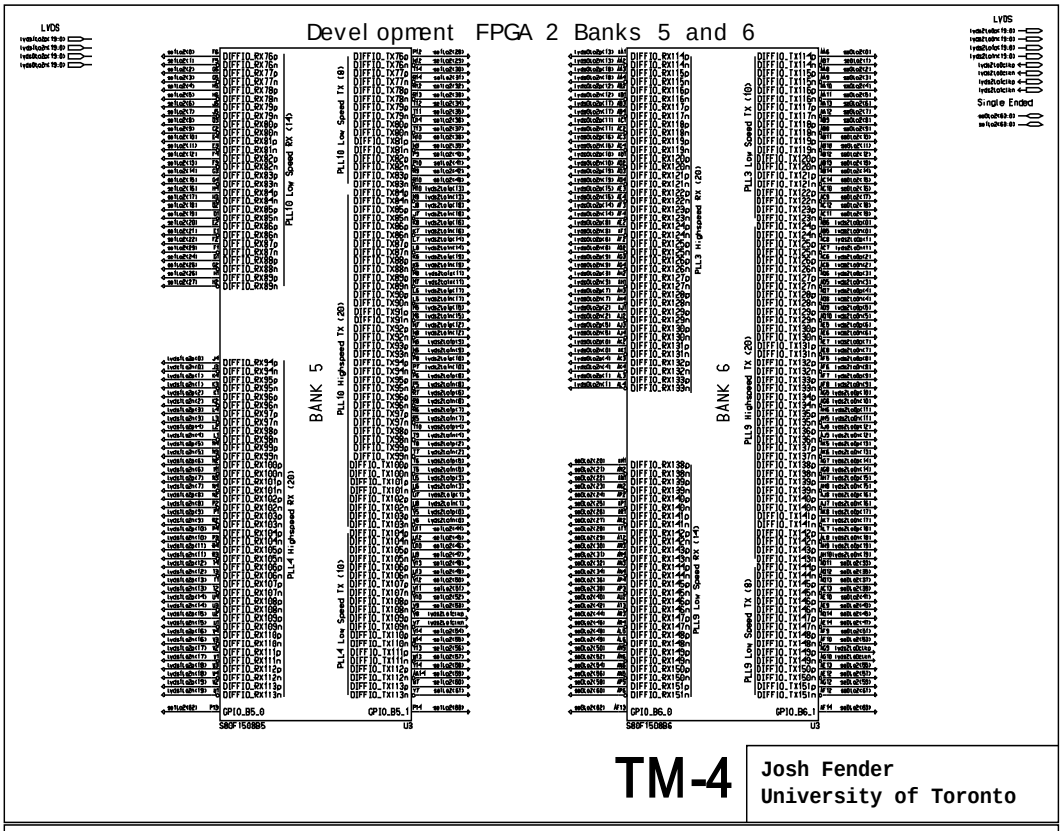

Development FPGA 1 Banks 3 and 4
Nibble Bus
DDR SDRAM Bank A
Logic Analyzer Connector
BANK 3
BANK 4
TM-4
Josh Fender
University of Toronto

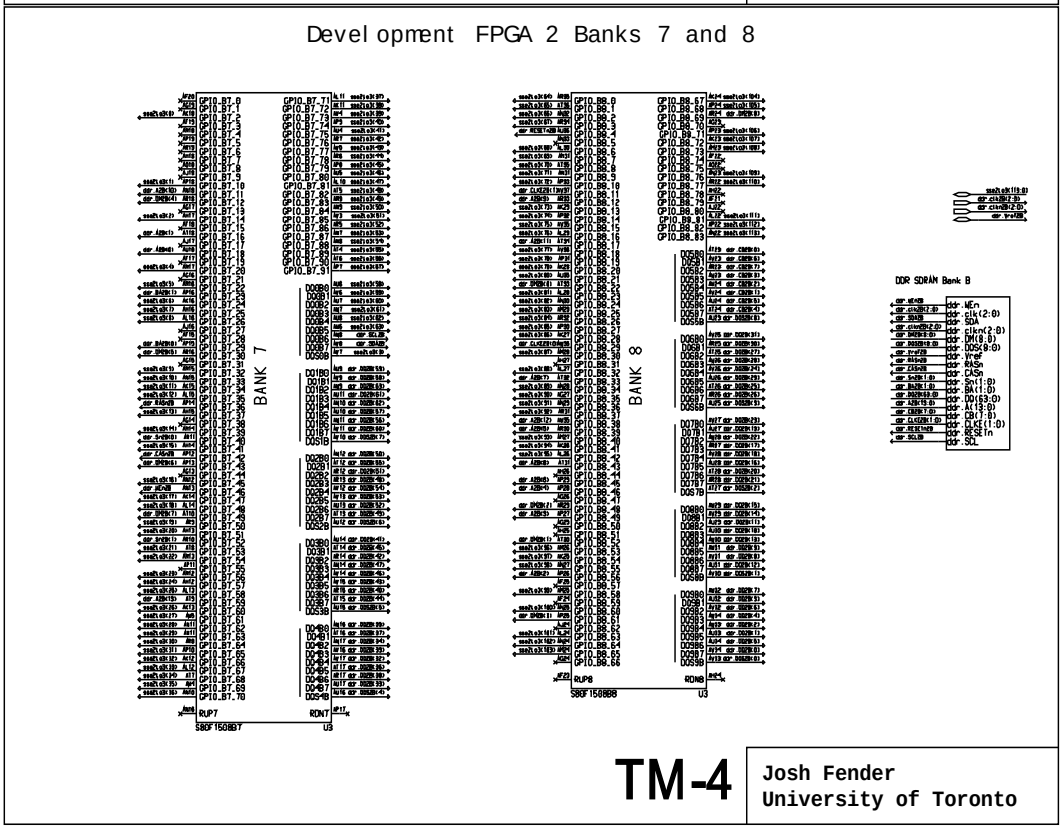

Development FPGA 1 Banks 5 and 6
LVDS
IEEE-1394 A
IEEE-1394 B
BANK 5
BANK 6
PLL10 Low Speed RX (14)
PLL10 Highspeed TX (20)
PLL4 Highspeed RX (20)
PLL4 Low Speed TX (10)
PLL3 Low Speed TX (10)
PLL3 Highspeed RX (20)
PLL9 Highspeed TX (20)
PLL9 Low Speed RX (14)
PLL9 Low Speed TX (8)
Single Ended
TM-4
Josh Fender
University of Toronto

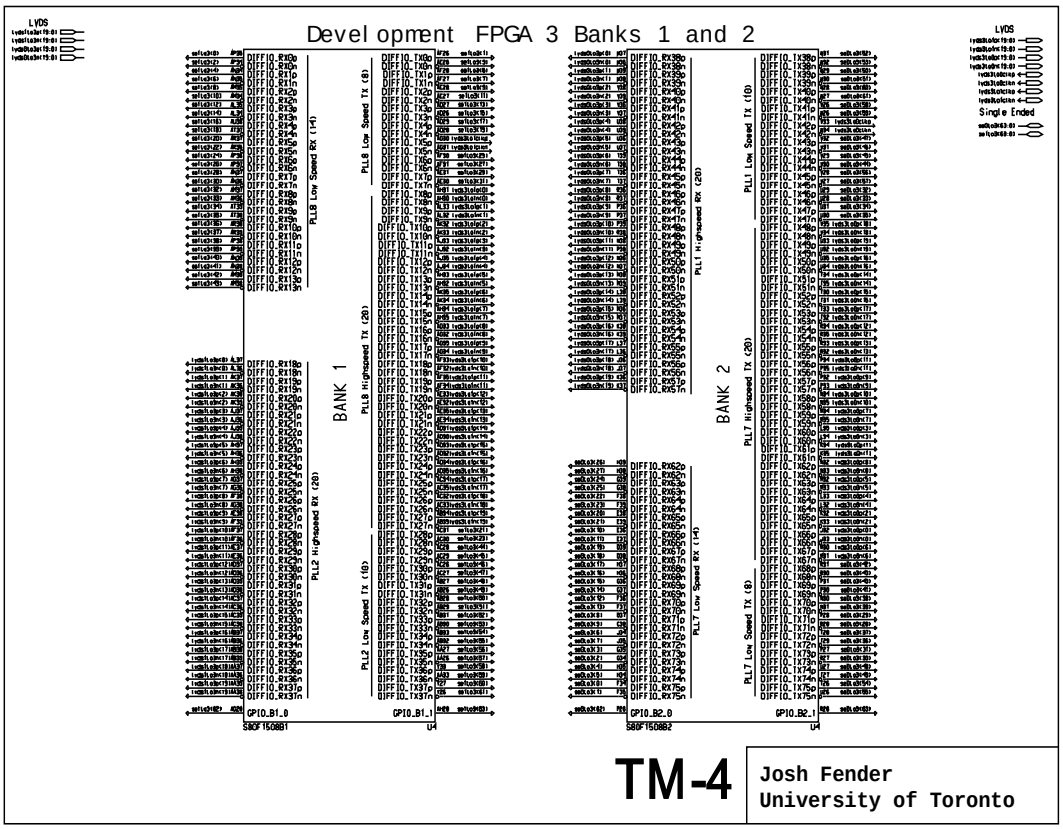

Development FPGA 1 Banks 7 and 8
BANK 7
BANK 8
DDR SDRAM Bank B
TM-4
Josh Fender
University of Toronto

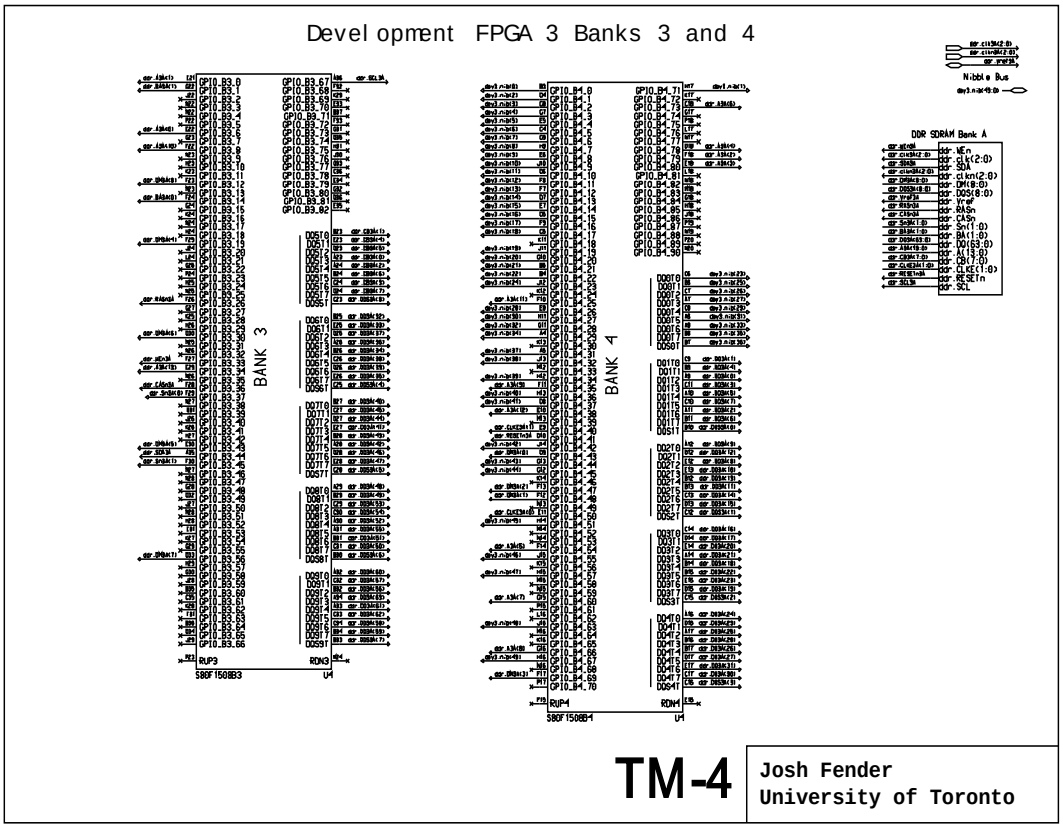

Development FPGA 2 Banks 1 and 2
LVDS
Single Ended
Expansion LVDS Clock
BANK 1
BANK 2
40 Pin
Expansion Header A1
40 Pin
Expansion Header A2
vcc33
vcc15
TM-4
Josh Fender
University of Toronto

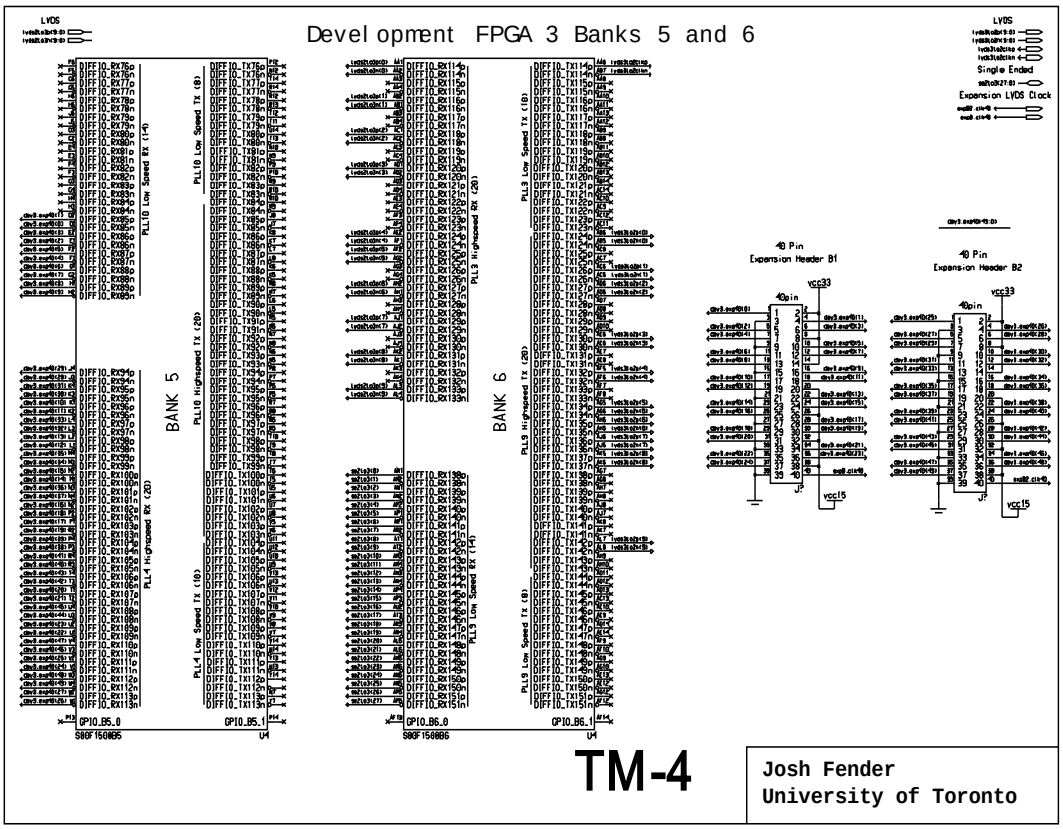

Development FPGA 2 Banks 3 and 4
Nibble Bus
DDR SDRAM Bank A
BANK 3
BANK 4
TM-4
Josh Fender
University of Toronto

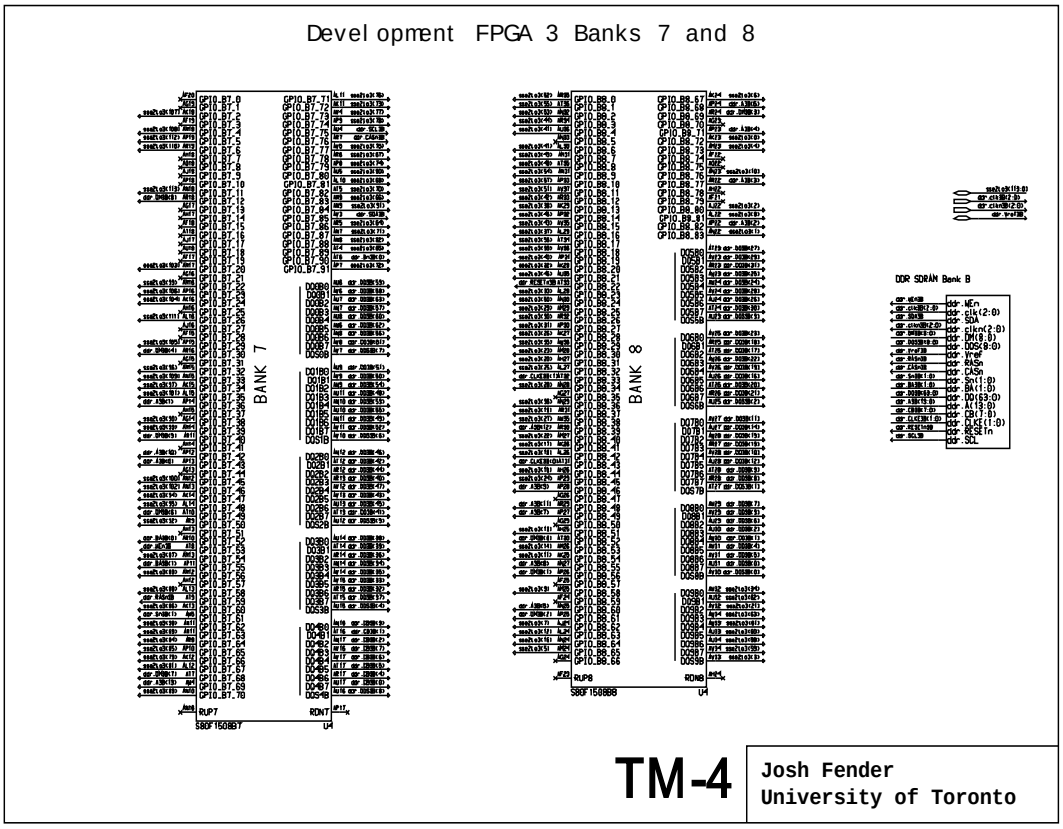

Development FPGA 2 Banks 5 and 6
LVDS
Single Ended
BANK 5
BANK 6
TM-4
Josh Fender
University of Toronto


Development FPGA 2 Banks 7 and 8
BANK 7
BANK 8
DDR SDRAM Bank B
TM-4
Josh Fender
University of Toronto

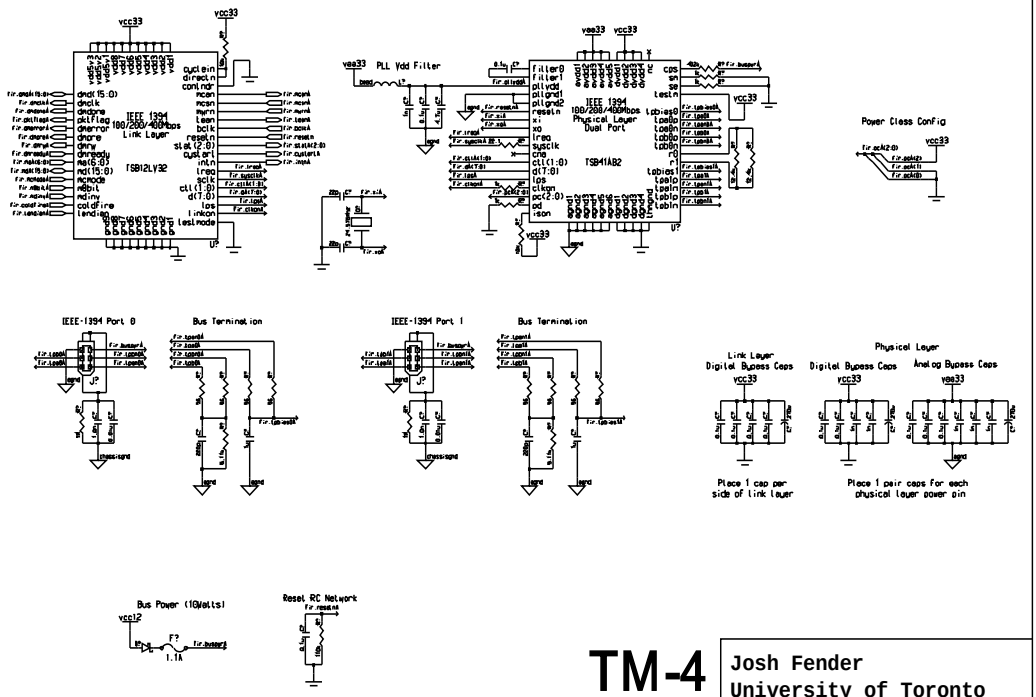

Development FPGA 3 Banks 1 and 2
LVDS
BANK 1
BANK 2
PLL8 Low Speed RX (14)
PLL8 Low Speed TX (8)
PLL8 Highspeed TX (28)
PLL2 Highspeed RX (28)
PLL2 Low Speed TX (16)
PLL1 Low Speed TX (16)
PLL1 Highspeed RX (28)
PLL7 Highspeed TX (28)
PLL7 Low Speed RX (14)
PLL7 Low Speed TX (8)
Single Ended
GPIO_B1_0
GPIO_B1_1
GPIO_B2_0
GPIO_B2_1
5SGF150BB1
5SGF150BB2
U4
TM-4
Josh Fender
University of Toronto


Development FPGA 3 Banks 3 and 4
BANK 3
BANK 4
Nibble Bus
DDR SDRAM Bank A
WEn
CLK(2:0)
SCL
clkn(2:0)
DM(8:0)
DQS(8:0)
Vref
BA(1:0)
CSn
Sn(1:0)
BA1(1:0)
DQ(63:0)
A(13:0)
CB(7:0)
CLKE(1:0)
RESETn
SCL
RUP3
RDN3
RUP4
RDN4
5SGF150BB3
5SGF150BB4
U4
TM-4
Josh Fender
University of Toronto


Development FPGA 3 Banks 5 and 6
LVDS
BANK 5
BANK 6
40 Pin
Expansion Header B1
40 Pin
Expansion Header B2
Single Ended
Expansion LVDS Clock
TM-4
Josh Fender
University of Toronto


Development FPGA 3 Banks 7 and 8
BANK 7
BANK 8
DDR SDRAM Bank B
TM-4
Josh Fender
University of Toronto

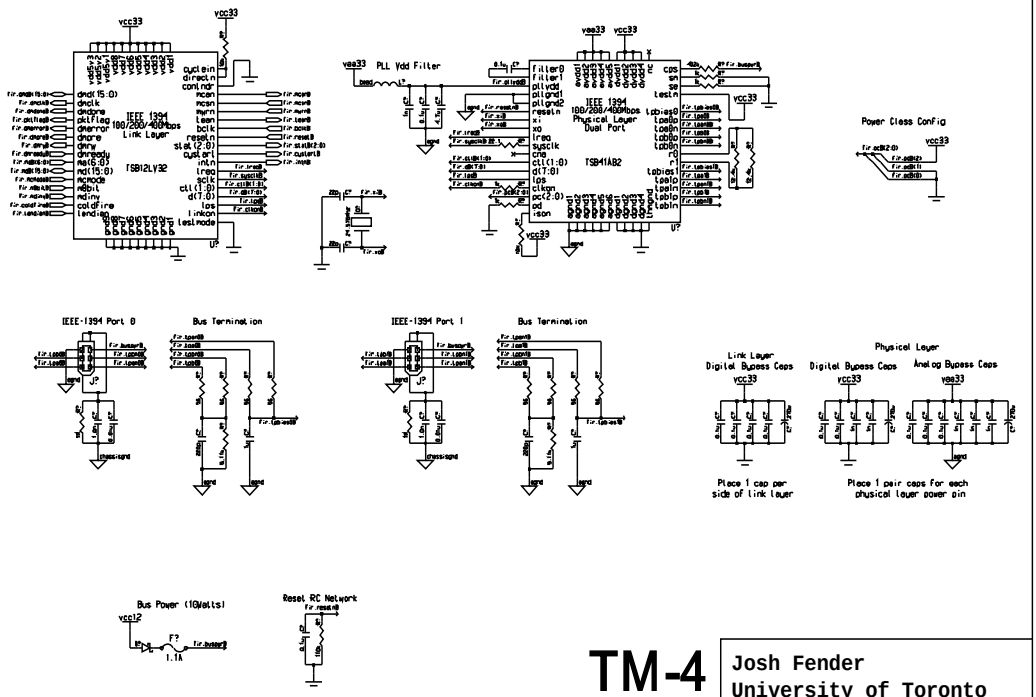
IEEE-1394 (Firewire) Channel A



TM-4

Josh Fender
University of Toronto

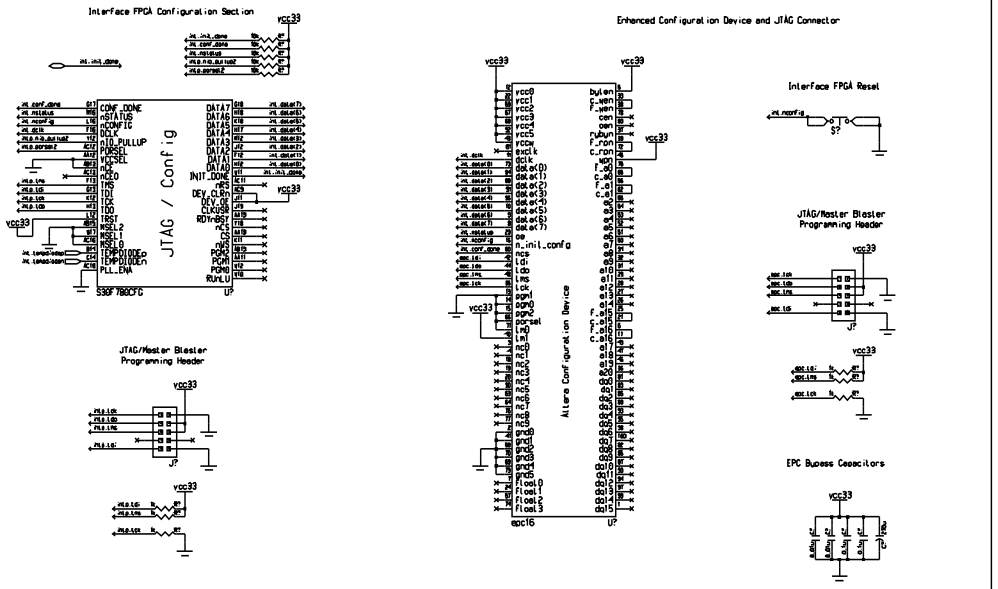
IEEE-1394 (Firewire) Channel B



TM-4

Josh Fender
University of Toronto

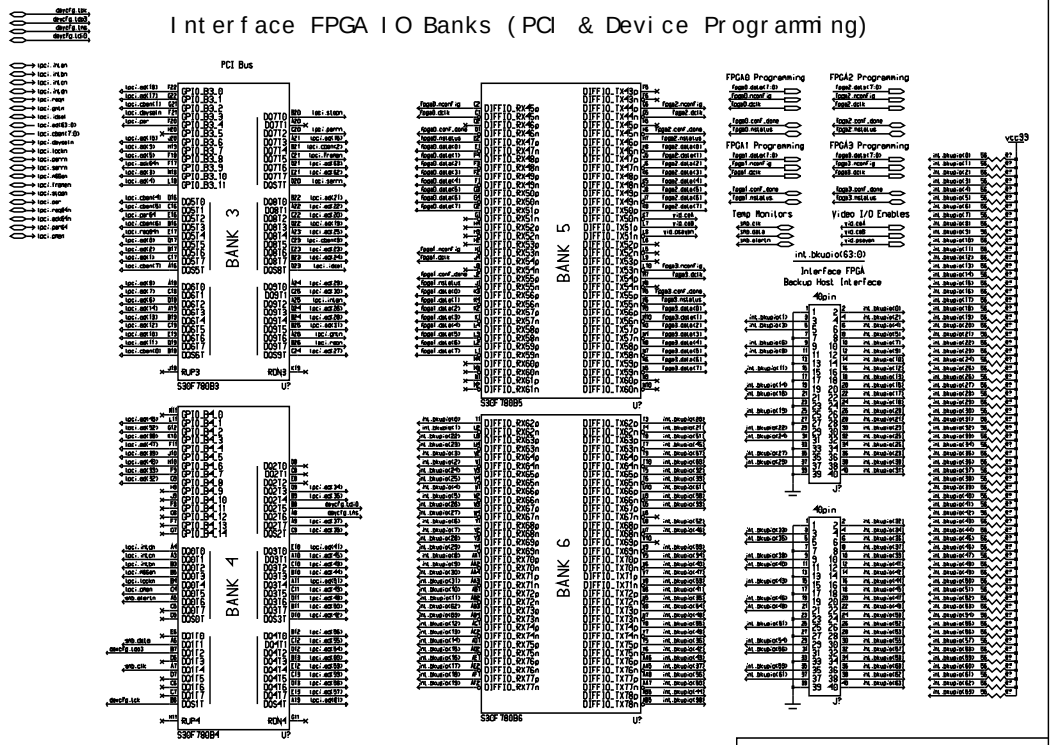
Interface FPGA Device Configuration



TM-4

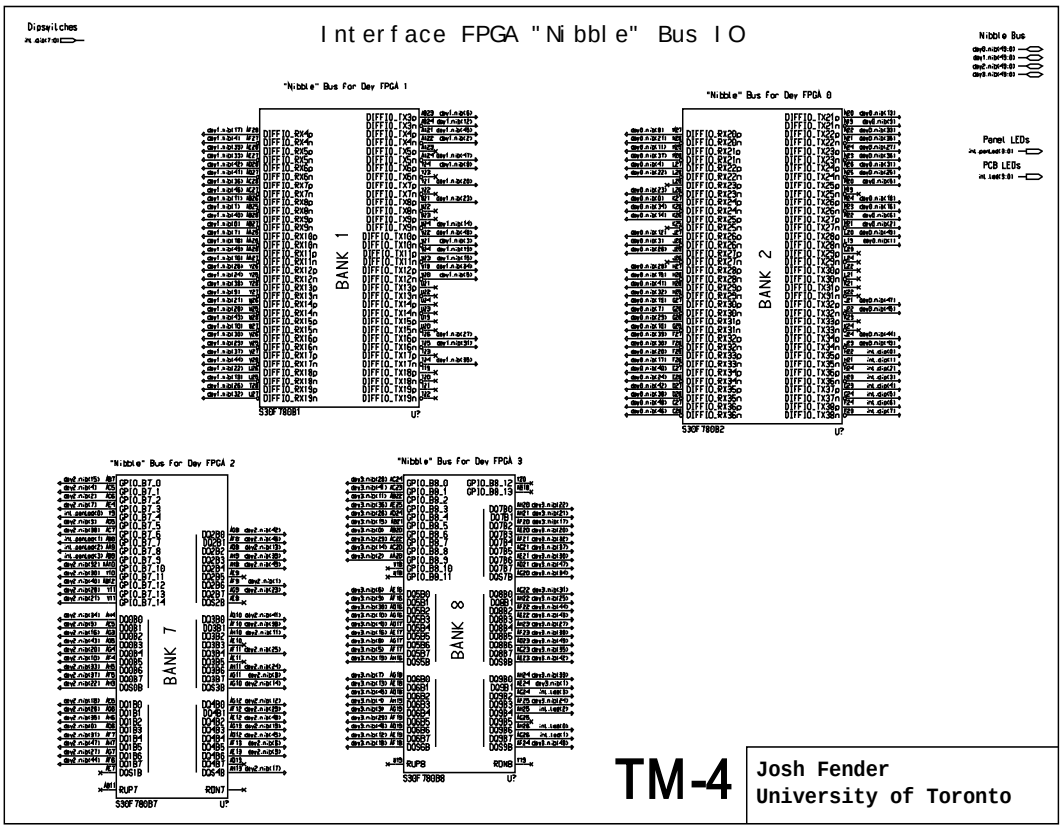
Josh Fender
University of Toronto

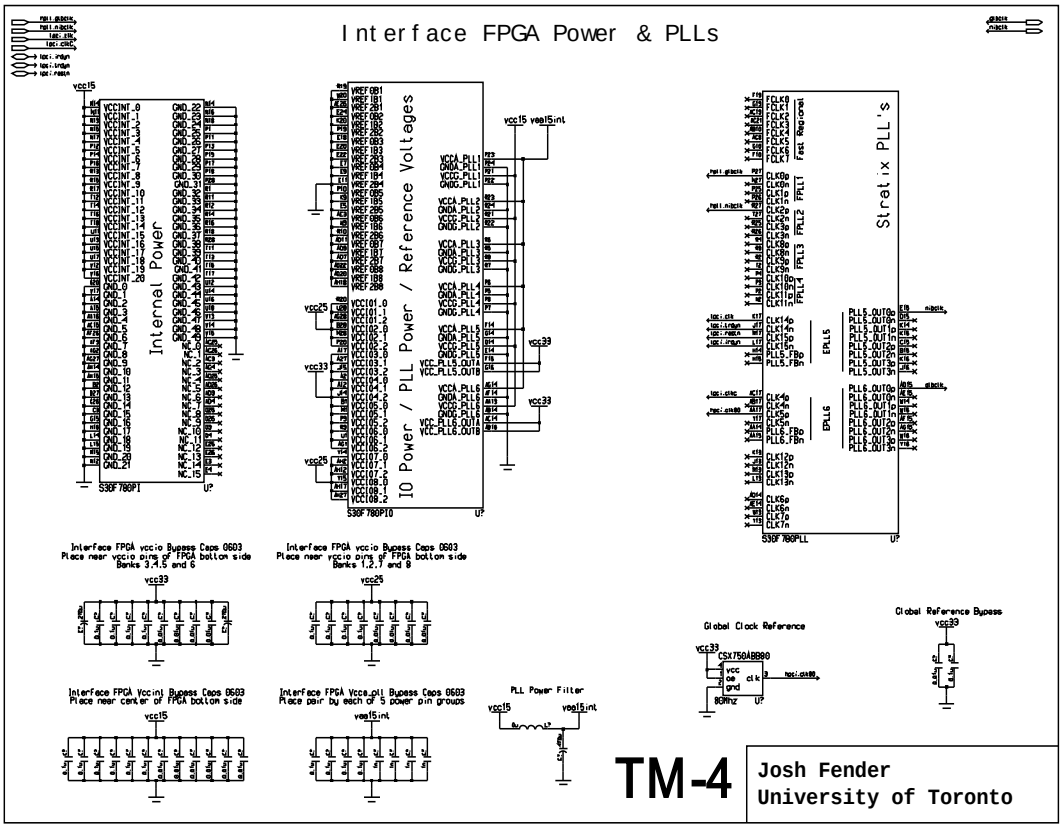
Interface FPGA IO Banks (PCI & Device Programming)

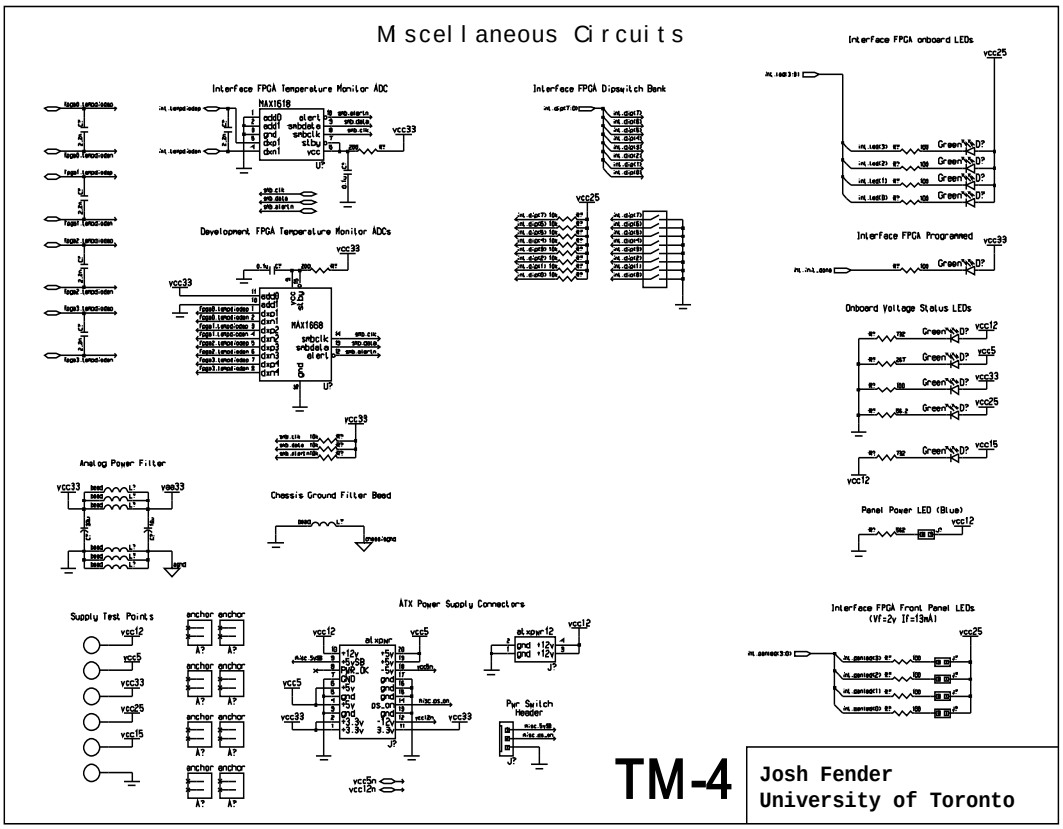


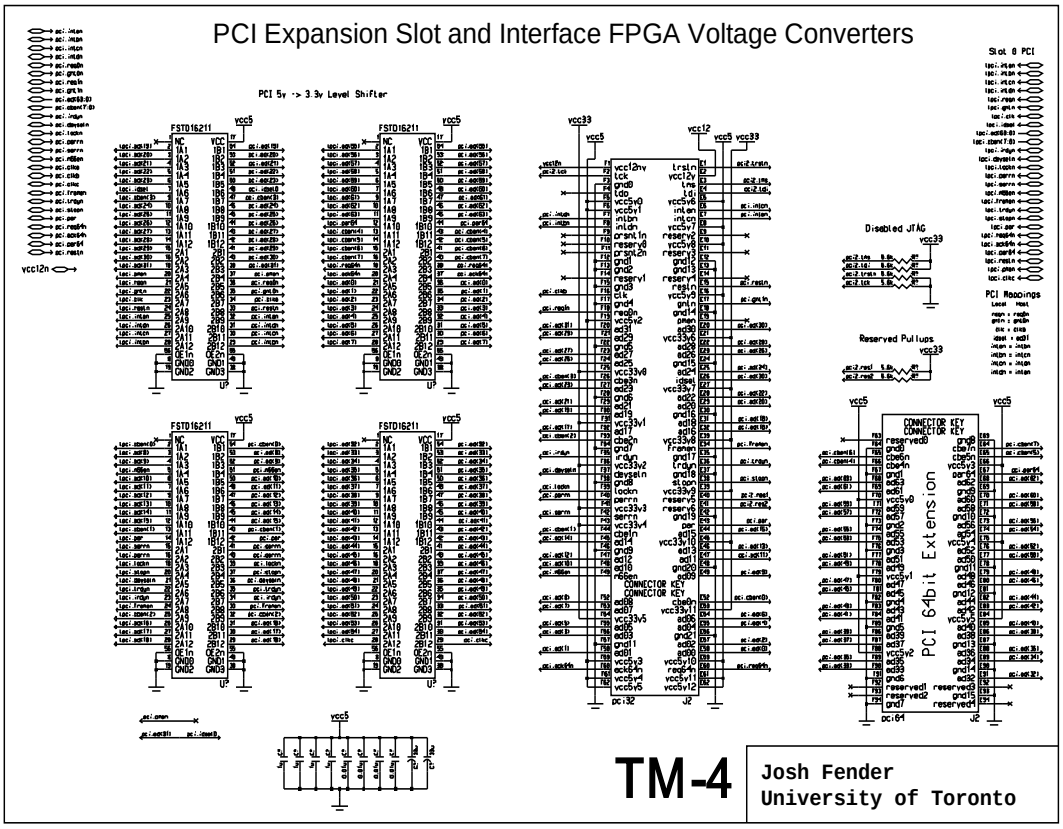
TM-4

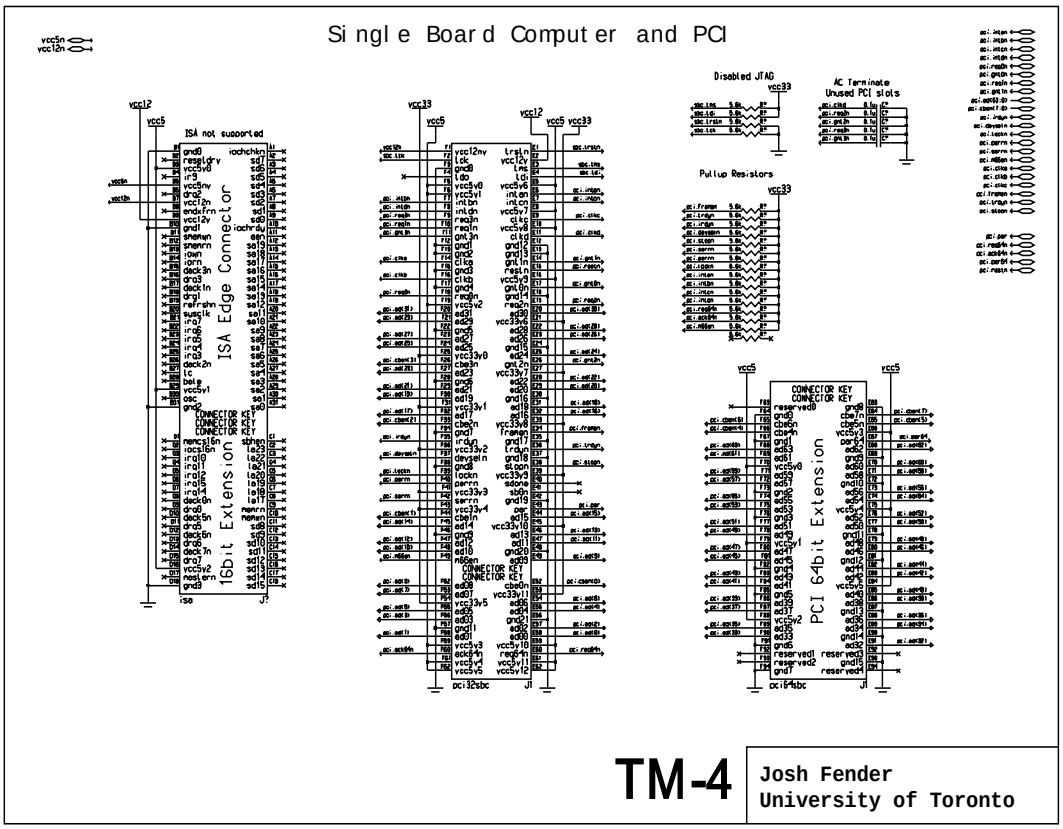
Josh Fender
University of Toronto

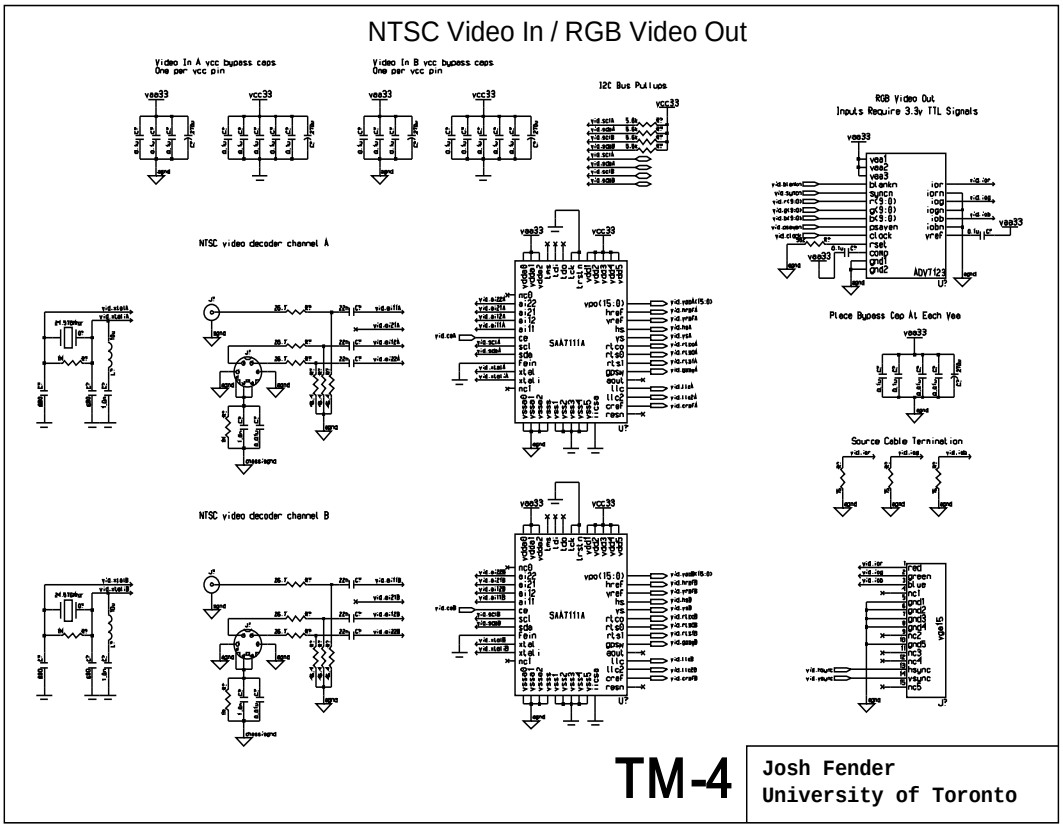

Interface FPGA "Nibble" Bus IO
Dipswitches
Nibble Bus
"Nibble" Bus For Dev FPGA 1
"Nibble" Bus For Dev FPGA 0
"Nibble" Bus For Dev FPGA 2
"Nibble" Bus For Dev FPGA 3
Panel LEDs
PCB LEDs
BANK 1
BANK 2
BANK 7
BANK 8
TM-4
Josh Fender
University of Toronto


Interface FPGA Power & PLLs
Internal Power
IO Power / PLL Power / Reference Voltages
Stratix PLL's
Interface FPGA vccio Bypass Caps 0603
Place near vccio pins of FPGA bottom side
Banks 3,4,5 and 6
Interface FPGA vccio Bypass Caps 0603
Place near vccio pins of FPGA bottom side
Banks 1,2,7 and 8
Interface FPGA Vccint Bypass Caps 0603
Place near center of FPGA bottom side
Interface FPGA Vcca_pll Bypass Caps 0603
Place pair by each of 5 power pin groups
PLL Power Filter
Global Clock Reference
Global Reference Bypass
TM-4
Josh Fender
University of Toronto

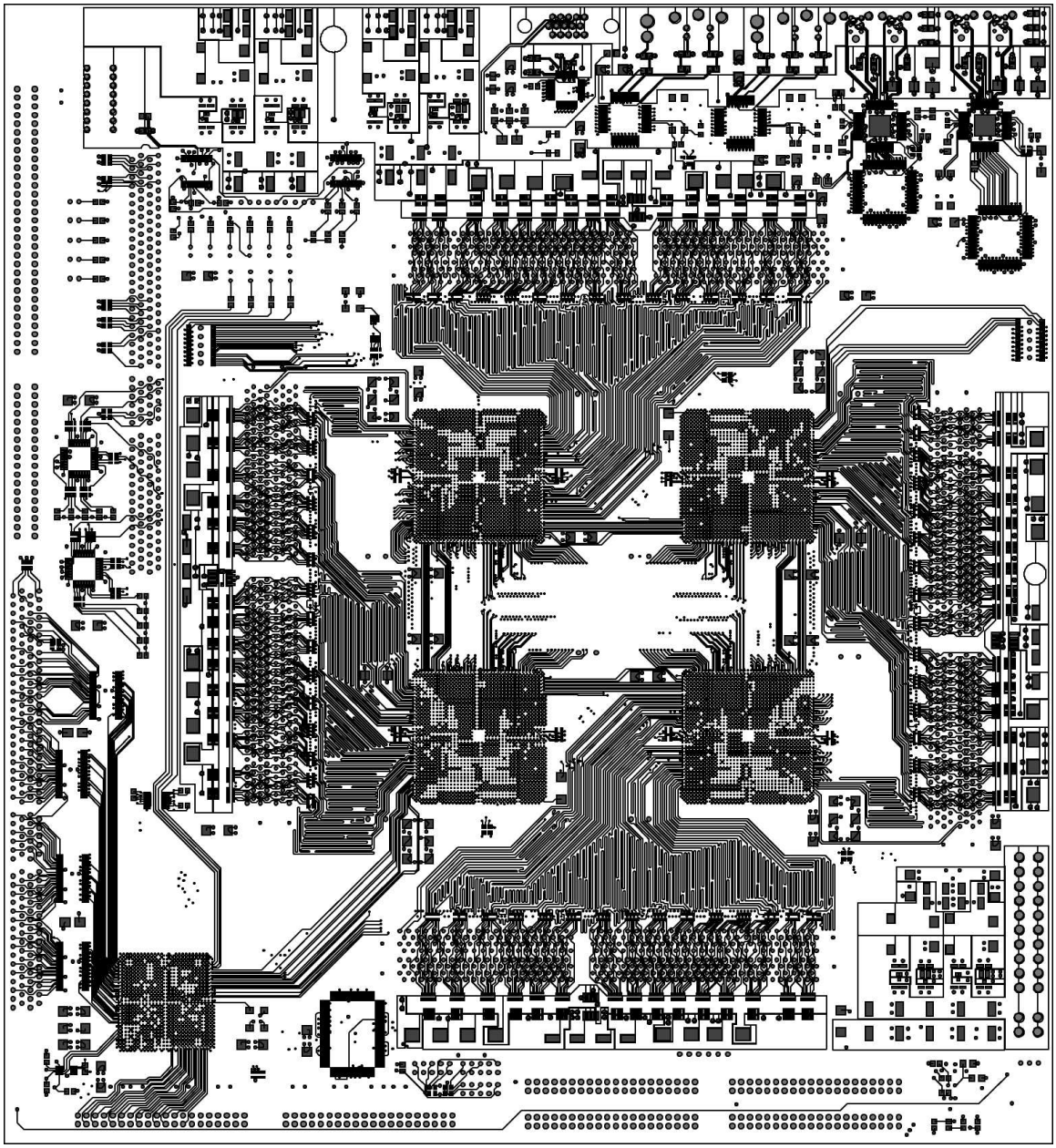

Miscellaneous Circuits
Interface FPGA Temperature Monitor ADC
MAX1618
Development FPGA Temperature Monitor ADCs
MAX1668
Interface FPGA Dipswitch Bank
Interface FPGA onboard LEDs
Interface FPGA Programmed
Onboard Voltage Status LEDs
Analog Power Filter
Chassis Ground Filter Bead
Panel Power LED (Blue)
ATX Power Supply Connectors
Supply Test Points
Pwr Switch Header
Interface FPGA Front Panel LEDs (Vf=2v If=13mA)
TM-4
Josh Fender
University of Toronto


PCI Expansion Slot and Interface FPGA Voltage Converters
PCI 5v -> 3.3v Level Shifter
FST16211
Slot 0 PCI
Disabled JTAG
Reserved Pullups
PCI Reservings
CONNECTOR KEY
PCI 64bit Extension
TM-4
Josh Fender
University of Toronto

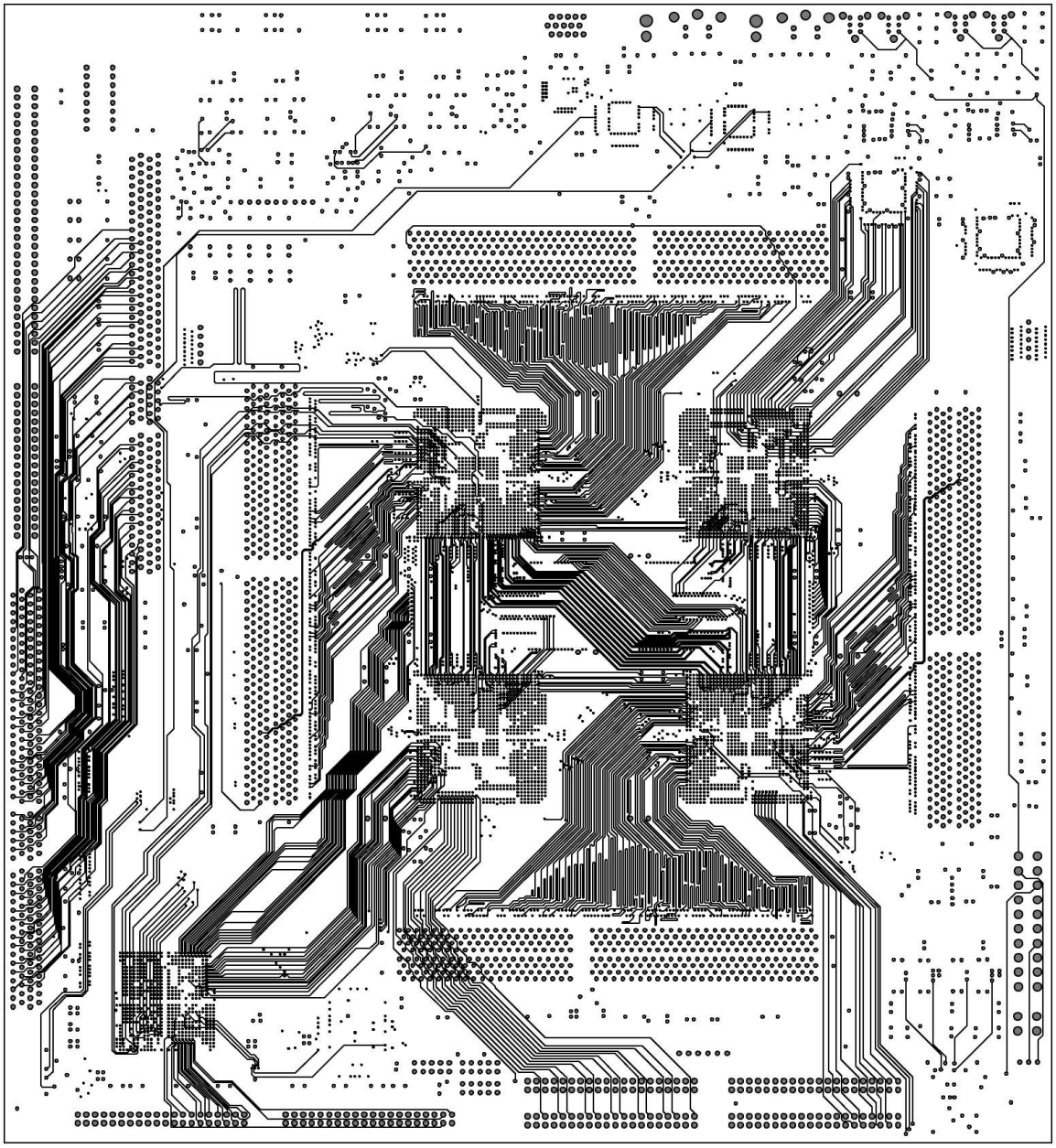

Single Board Computer and PCI
ISA not supported
ISA Edge Connector
16bit Extension
Disabled JTAG
AC Terminate Unused PCI slots
Pullup Resistors
PCI 64bit Extension
TM-4
Josh Fender
University of Toronto


NTSC Video In / RGB Video Out
Video In A vcc bypass caps One per vcc pin
Video In B vcc bypass caps One per vcc pin
I2C Bus Pullups
RGB Video Out Inputs Require 3.3v TTL Signals
NTSC video decoder channel A
NTSC video decoder channel B
SAA7111A
ADV7125
Place Bypass Cap At Each Vee
Source Cable Termination
TM-4
Josh Fender
University of Toronto

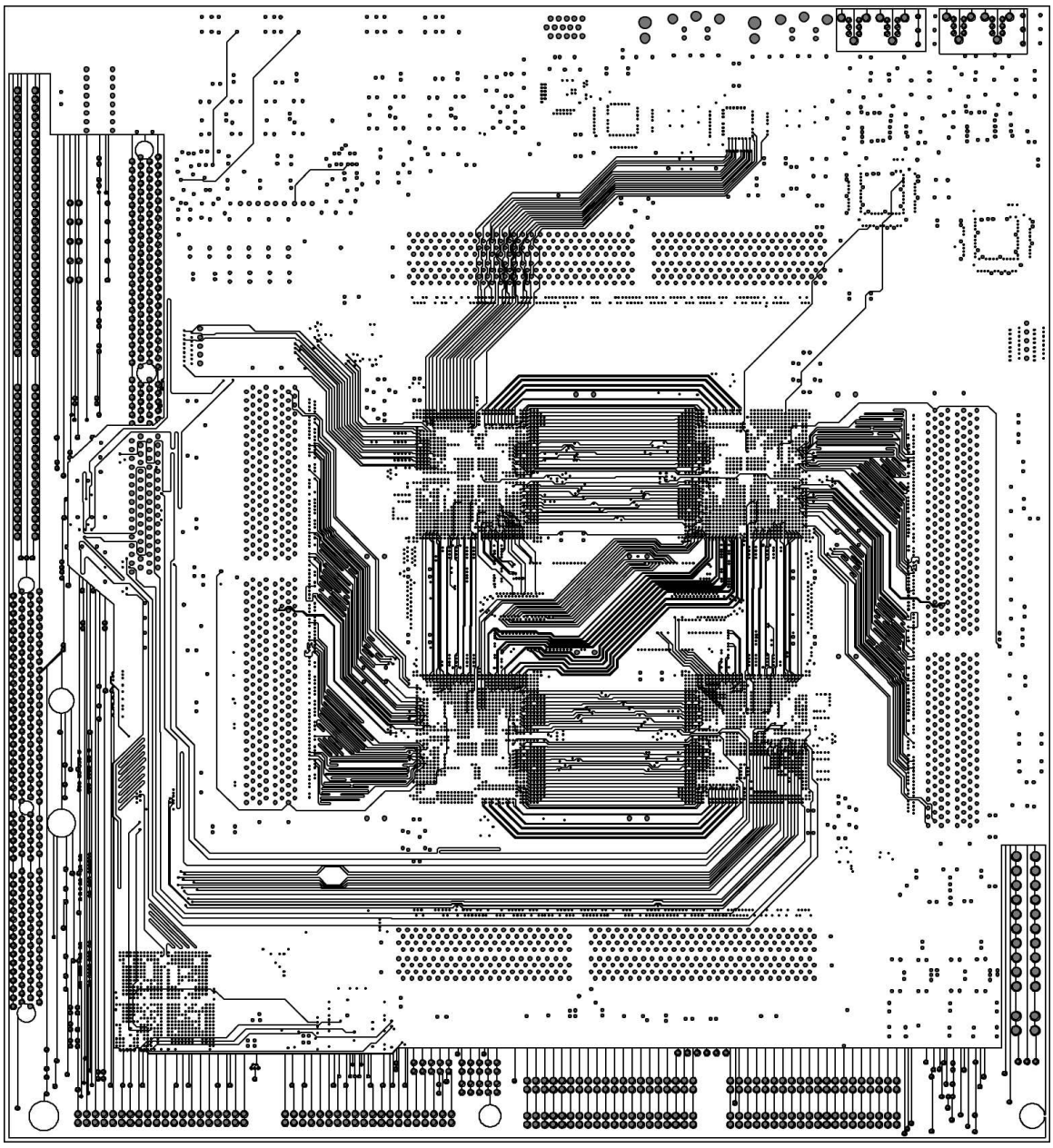
B PCB LAYOUT



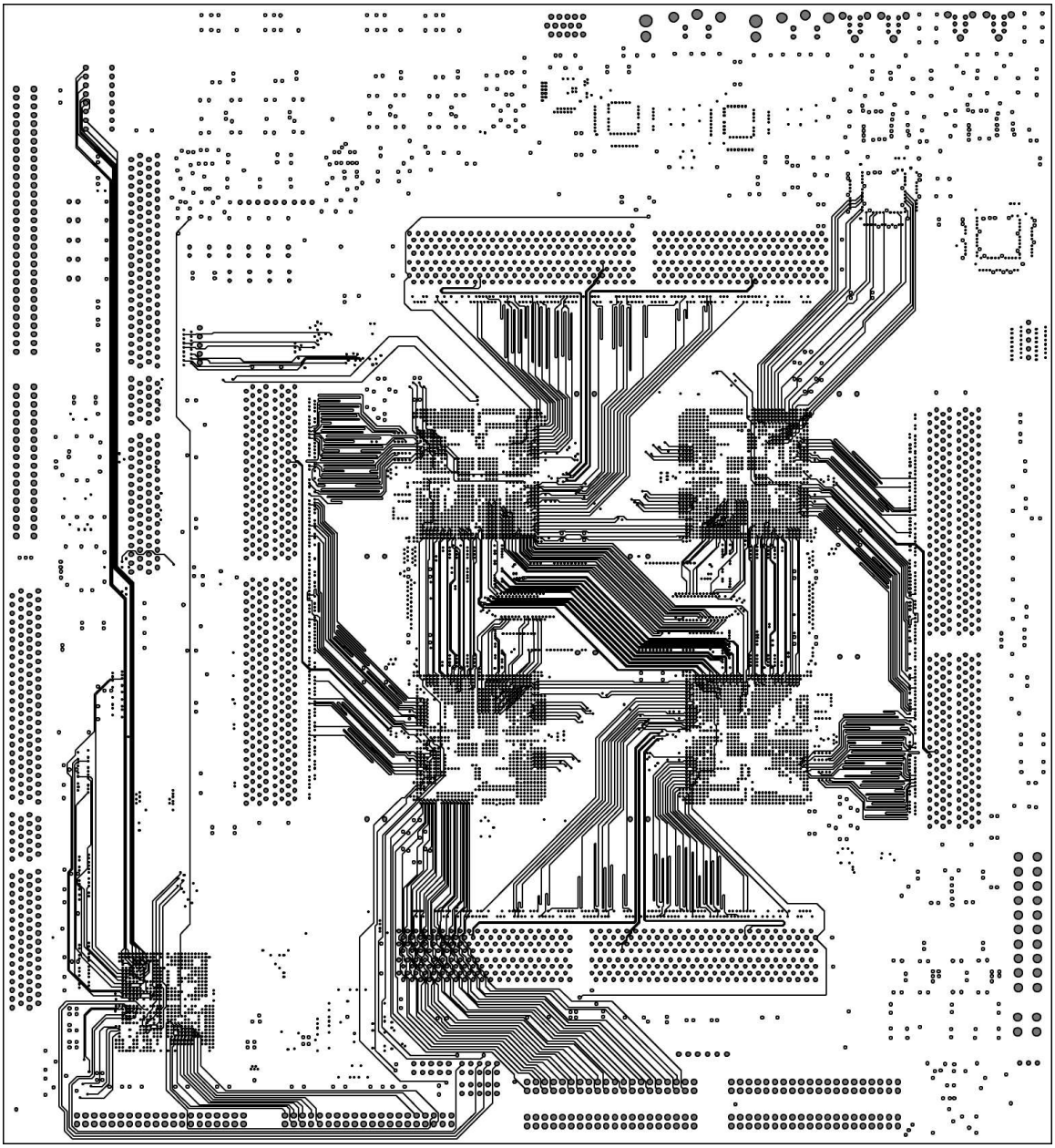
Layer 1



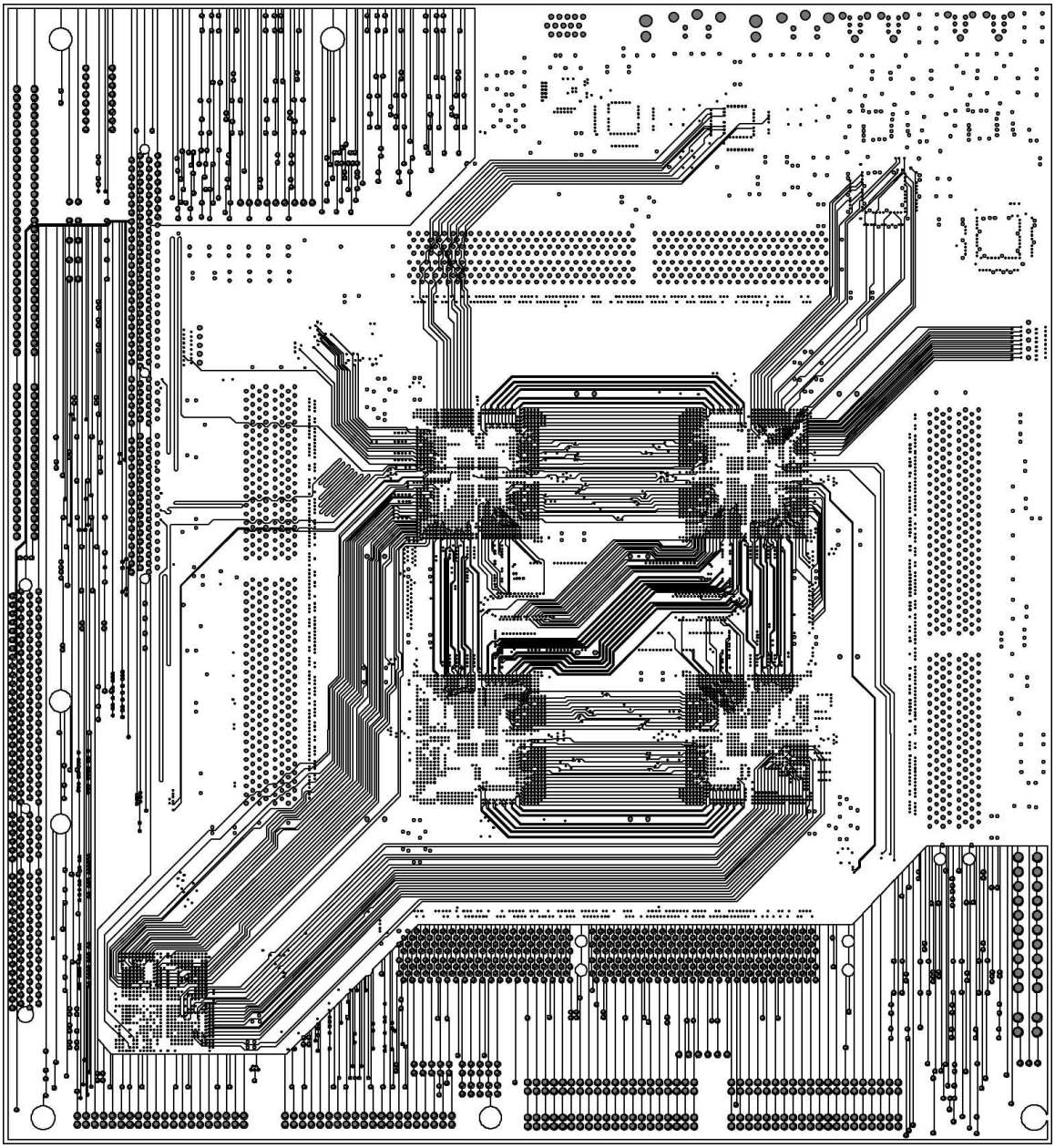
Layer 3



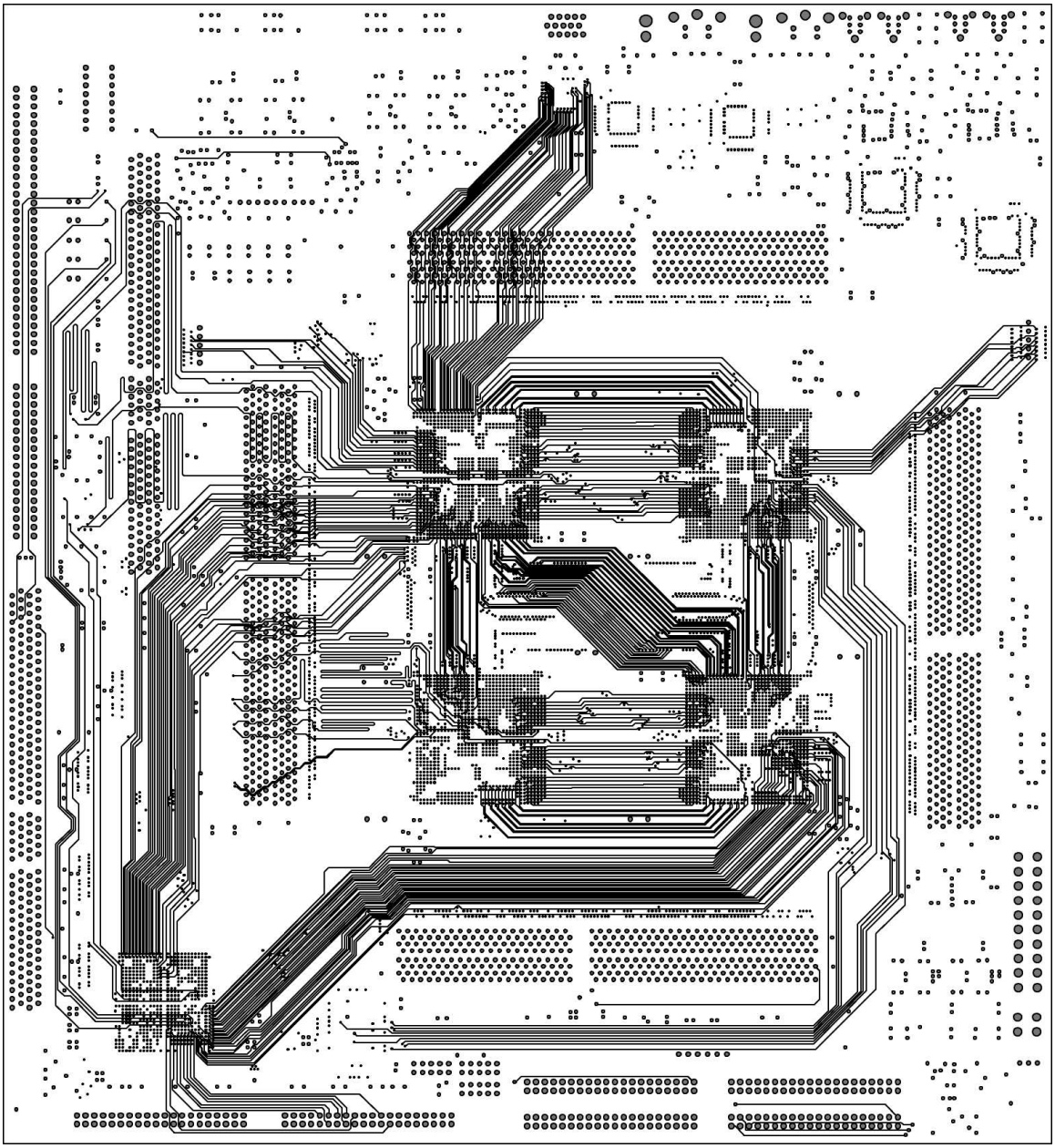
Layer 4



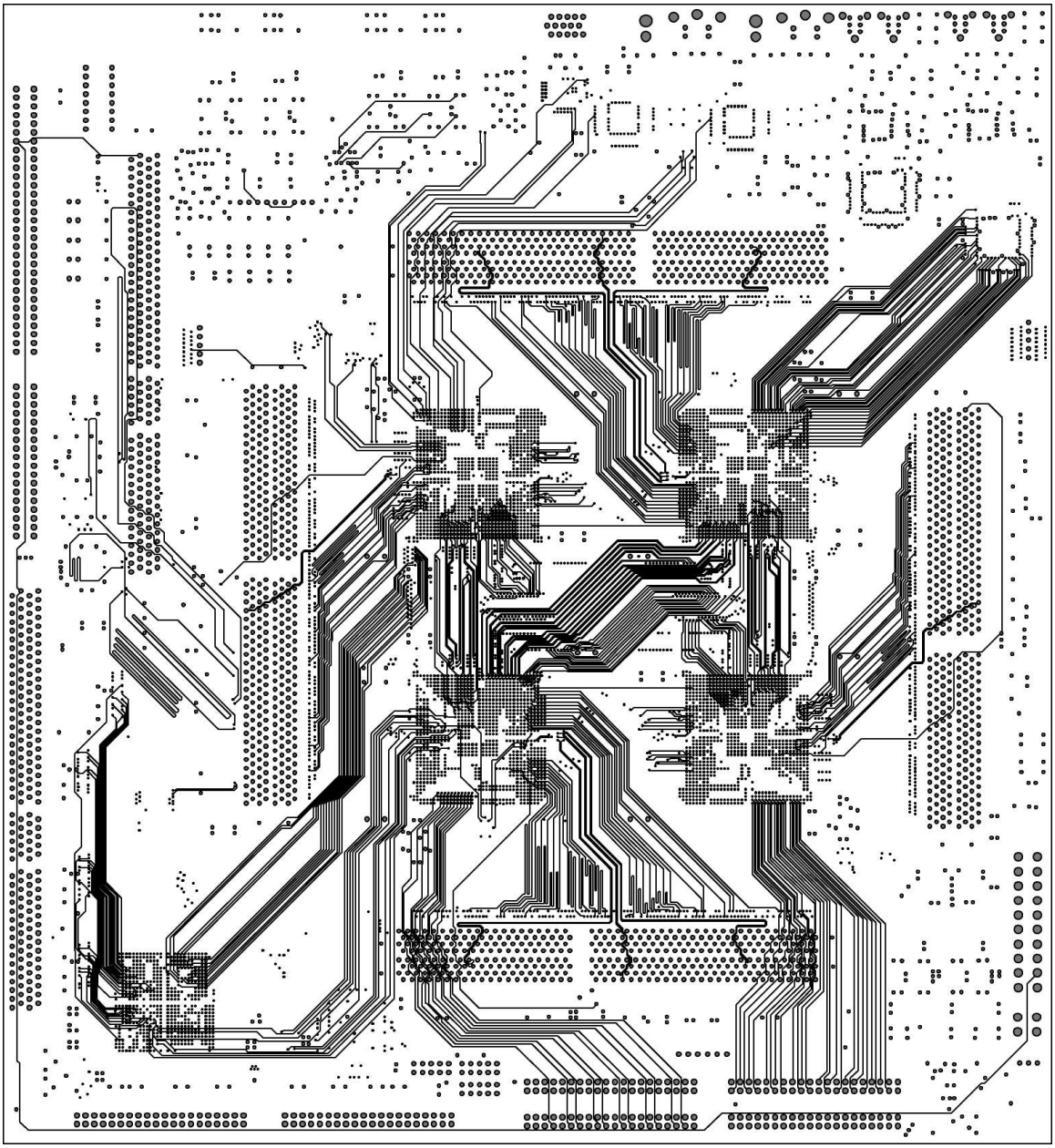
Layer 6



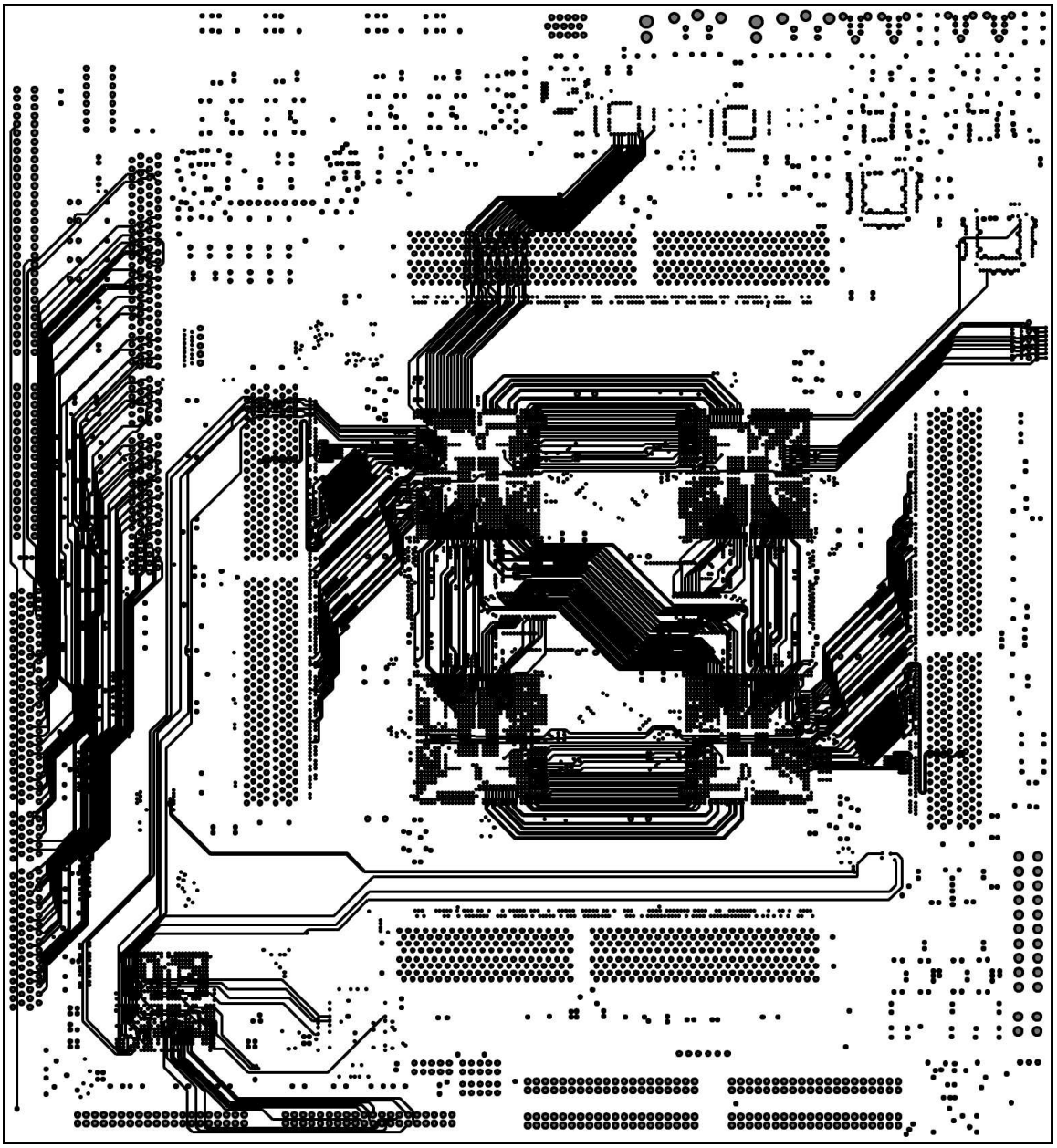
Layer 7



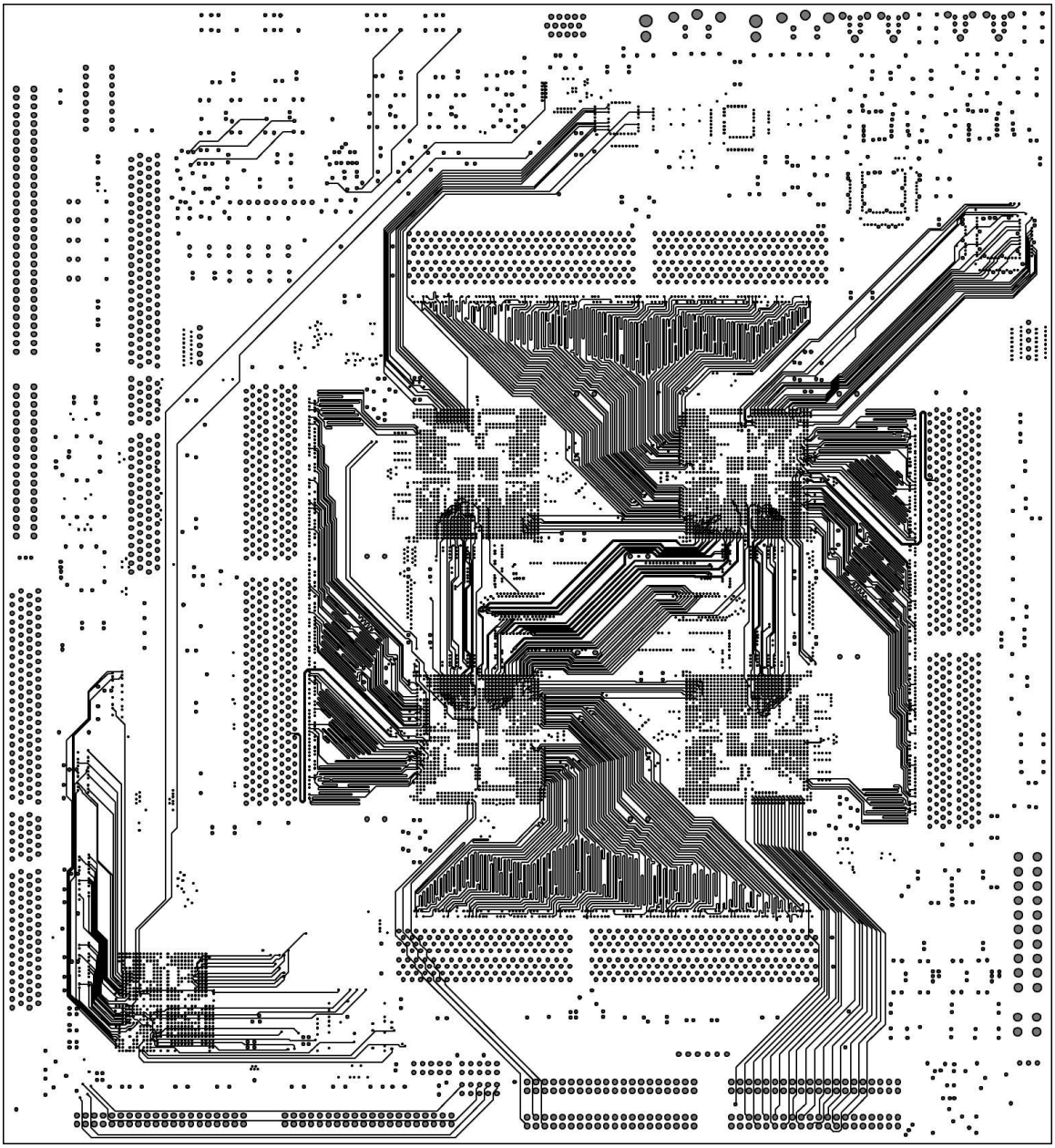
Layer 10



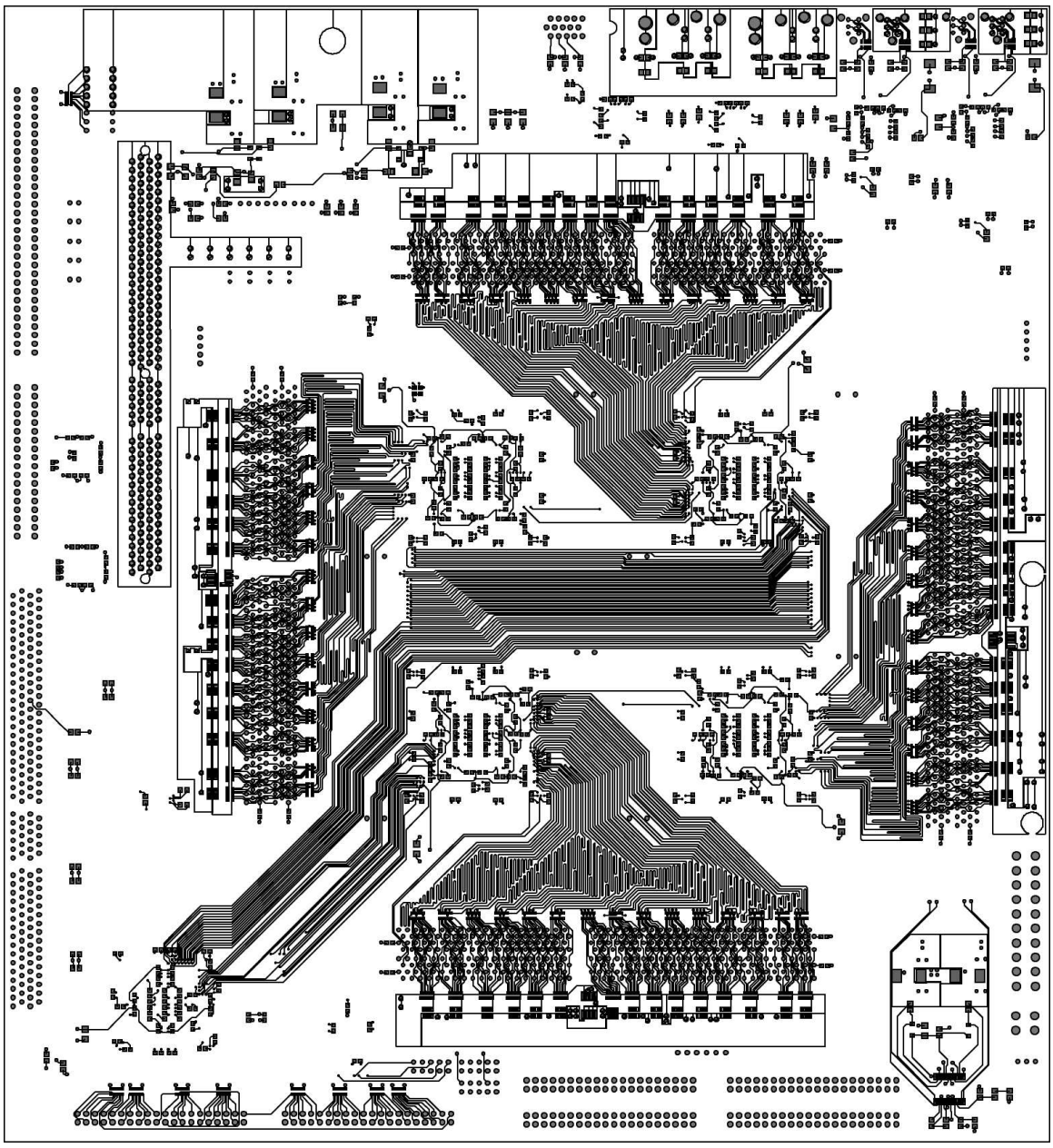
Layer 11



Layer 13



Layer 14



Layer 16

C INTERFACE FPGA VHDL CODE

C.1 TOP.VHD

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_arith.all;
USE ieee.std_logic_unsigned.all;

LIBRARY work;
USE work.complib.all;

ENTITY top IS
  PORT (
    -- Clock Inputs
    hpll_glbclk : IN STD_LOGIC;   -- Board wide phase aligned clocks
    hpll_nibclk : IN STD_LOGIC;
    clk80 : IN STD_LOGIC;         -- 80Mhz Oscillator backup
    spci_clk : IN STD_LOGIC;      -- Secondary PCI clock input
    -- Clock Outputs
    nibclk : OUT STD_LOGIC;
    glbclk : OUT STD_LOGIC;

    -- Development JTAG Chain
    devcfg_TCK : INOUT STD_LOGIC;
    devcfg_TDO : IN STD_LOGIC;
    devcfg_TDI : INOUT STD_LOGIC;
    devcfg_TMS : INOUT STD_LOGIC;
    -- Dipswitchs/LEDS
    dip : IN STD_LOGIC_VECTOR(7 downto 0);
    led : OUT STD_LOGIC_VECTOR(3 downto 0);
    panled : OUT STD_LOGIC_VECTOR(3 downto 0);
    -- Nibble/Development Bus Signals
    nib0 : INOUT STD_LOGIC_VECTOR(49 downto 0);
    nib1 : INOUT STD_LOGIC_VECTOR(49 downto 0);
    nib2 : INOUT STD_LOGIC_VECTOR(49 downto 0);
    nib3 : INOUT STD_LOGIC_VECTOR(49 downto 0);
    -- Backup IO Headers
    bkupio : OUT STD_LOGIC_VECTOR(63 downto 0);
    -- Temp Monitors
    smb_clk : INOUT STD_LOGIC;
    smb_data : INOUT STD_LOGIC;
    smb_alertn : IN STD_LOGIC;
    -- Video Enables
    vid_ceA : OUT STD_LOGIC;
    vid_ceB : OUT STD_LOGIC;
    vid_psaven : OUT STD_LOGIC;
    -- Development FPGA 0 FPP programming signals
    fpga0_data : OUT STD_LOGIC_VECTOR(7 downto 0);
    fpga0_nconfig : OUT STD_LOGIC;
    fpga0_dclk : OUT STD_LOGIC;
    fpga0_conf_done : IN STD_LOGIC;
    fpga0_nstatus : IN STD_LOGIC;
    -- Development FPGA 1 FPP programming signals
    fpga1_data : OUT STD_LOGIC_VECTOR(7 downto 0);
    fpga1_nconfig : OUT STD_LOGIC;
    fpga1_dclk : OUT STD_LOGIC;
    fpga1_conf_done : IN STD_LOGIC;
    fpga1_nstatus : IN STD_LOGIC;
    -- Development FPGA 2 FPP programming signals
    fpga2_data : BUFFER STD_LOGIC_VECTOR(7 downto 0);
    fpga2_nconfig : OUT STD_LOGIC;
    fpga2_dclk : BUFFER STD_LOGIC;
    fpga2_conf_done : IN STD_LOGIC;
    fpga2_nstatus : IN STD_LOGIC;
    -- Development FPGA 3 FPP programming signals
    fpga3_data : OUT STD_LOGIC_VECTOR(7 downto 0);
    fpga3_nconfig : OUT STD_LOGIC;
    fpga3_dclk : OUT STD_LOGIC;
    fpga3_conf_done : IN STD_LOGIC;
    fpga3_nstatus : IN STD_LOGIC;

    -- PCI Bus Signals
    clk : IN STD_LOGIC;
    gntn : IN STD_LOGIC;
```

```
    rstn : IN STD_LOGIC;
    idsel : IN STD_LOGIC;
    framen : INOUT STD_LOGIC;
    irdyn : INOUT STD_LOGIC;
    devseln : INOUT STD_LOGIC;
    trdyn : INOUT STD_LOGIC;
    stopn : INOUT STD_LOGIC;
    req64n : INOUT STD_LOGIC;
    ack64n : INOUT STD_LOGIC;
    intan : OUT STD_LOGIC;
    intbn : OUT STD_LOGIC;
    intcn : OUT STD_LOGIC;
    intdn : OUT STD_LOGIC;
    lockn : OUT STD_LOGIC;
    reqn : OUT STD_LOGIC;
    serrn : OUT STD_LOGIC;
    ad : INOUT STD_LOGIC_VECTOR (63 DOWNTO 0);
    cben : INOUT STD_LOGIC_VECTOR (7 DOWNTO 0);
    par : INOUT STD_LOGIC;
    par64 : INOUT STD_LOGIC;
    perrn : INOUT STD_LOGIC;
    m66en : INOUT STD_LOGIC;
    pmen : INOUT STD_LOGIC
  );
END;

ARCHITECTURE rtl of top IS
  -- Local side PCI core signal declarations
  SIGNAL l_cbeni : STD_LOGIC_VECTOR (7 DOWNTO 0);
  SIGNAL l_adi : STD_LOGIC_VECTOR (63 DOWNTO 0);
  SIGNAL lm_req32n, lm_req64n, lm_lastn, lm_rdyn, lt_rdyn : STD_LOGIC;
  SIGNAL lt_abortn, lt_discn, lirqn : STD_LOGIC;
  SIGNAL l_adro, l_dato : STD_LOGIC_VECTOR (63 DOWNTO 0);
  SIGNAL l_beno : STD_LOGIC_VECTOR (7 DOWNTO 0);
  SIGNAL l_cmdo : STD_LOGIC_VECTOR (3 DOWNTO 0);
  SIGNAL l_ldat_ackn, l_hdat_ackn, lm_adr_ackn, lm_ackn :  STD_LOGIC;
  SIGNAL lm_dxfrn : STD_LOGIC;
  SIGNAL lm_tsr : STD_LOGIC_VECTOR (9 DOWNTO 0);
  SIGNAL lt_framen, lt_ackn, lt_dxfrn : STD_LOGIC;
  SIGNAL lt_tsr : STD_LOGIC_VECTOR (11 DOWNTO 0);
  SIGNAL cmd_reg, stat_reg : STD_LOGIC_VECTOR (6 DOWNTO 0);
  SIGNAL cache : STD_LOGIC_VECTOR (7 DOWNTO 0);
  SIGNAL master_l_adi, target_l_adi : STD_LOGIC_VECTOR(63 downto 0);
  SIGNAL master_l_adi_enable : STD_LOGIC;
  -- PLL signals
  SIGNAL pll_reconfig_reset : STD_LOGIC;
  SIGNAL nib_locked, glb_locked : STD_LOGIC;
  SIGNAL nib_pci_data_out, glb_pci_data_out : STD_LOGIC_VECTOR(31 downto
0);
  SIGNAL nib_pci_we, glb_pci_we : STD_LOGIC;
  -- Write Fifo signals
  SIGNAL writeFIFOdata : STD_LOGIC_VECTOR(31 downto 0);
  SIGNAL writeFIFOdatain : STD_LOGIC_VECTOR(63 downto 0);
  SIGNAL writeFIFOrdreq, writeFIFOempty : STD_LOGIC;
  SIGNAL writeFIFOnearlyfull, writeFIFOwrhigh, writeFIFOwrlow :
STD_LOGIC;
  SIGNAL writeFIFO_wrused : STD_LOGIC_VECTOR(12 downto 0);
  SIGNAL target_writeFIFOwrlow, target_writeFIFOwrhigh : STD_LOGIC;
  -- Read FIFO signals
  SIGNAL readFIFO_level : STD_LOGIC_VECTOR(12 downto 0);
  SIGNAL readFIFOdata : STD_LOGIC_VECTOR(31 downto 0);
  SIGNAL readFIFOwrreq : STD_LOGIC;
  SIGNAL readFIFO_get32bits, readFIFO_get64bits : STD_LOGIC;
  SIGNAL readFIFO_data32ready, readFIFO_data64ready : STD_LOGIC;
  SIGNAL readFIFO_nearlyempty : STD_LOGIC;
  SIGNAL readFIFO_Dataout : STD_LOGIC_VECTOR(63 downto 0);
  -- Development Bus Interface Signals
  SIGNAL devbus_enable : STD_LOGIC;
  SIGNAL devbus_writeFIFOrdreq : STD_LOGIC;
  SIGNAL devbus_peekstate : STD_LOGIC_VECTOR(2 downto 0);
  -- Dev Configure signals
  SIGNAL devcfg_enable : STD_LOGIC;
  SIGNAL devcfg_writeFIFOrdreq : STD_LOGIC;
  -- Double speed clocks
  SIGNAL clk133 : STD_LOGIC;
  -- 22Mhz PCI derived clock
  SIGNAL clk22 : STD_LOGIC;
  -- Control register file signals
  SIGNAL reg_data_in, reg_data_out : STD_LOGIC_VECTOR(31 downto 0);
  SIGNAL reg_data_addr : STD_LOGIC_VECTOR(7 downto 0);
  SIGNAL reg_read_ack, reg_wrreq : STD_LOGIC;
  -- Reset Signals
  SIGNAL force_resetn : STD_LOGIC;
  SIGNAL local_resetn : STD_LOGIC;
  -- Temperature Monitor Signals
  SIGNAL tempmc_pci_data_out : STD_LOGIC_VECTOR(31 downto 0);
  SIGNAL tempmc_pci_we : STD_LOGIC;
  SIGNAL alert_override : STD_LOGIC;
  -- Dev FPGA nConfig register signals
  SIGNAL fpga0_nconfig_reg, fpga1_nconfig_reg : STD_LOGIC;
  SIGNAL fpga2_nconfig_reg, fpga3_nconfig_reg : STD_LOGIC;
  -- Development JTAG controller signals
  SIGNAL jtag_enable : std_logic;
  SIGNAL jtag_dataout : STD_LOGIC_VECTOR(31 downto 0);
  SIGNAL jtag_we : STD_LOGIC;
  -- Nibble bus reset signals
  SIGNAL nib0_resetn, nib1_resetn : STD_LOGIC;
  SIGNAL nib2_resetn, nib3_resetn : STD_LOGIC;
  -- Error catchers
  SIGNAL fifo_overflow_error : STD_LOGIC;
  SIGNAL m_fifo_overflow_error : STD_LOGIC;
  SIGNAL t_fifo_overflow_error : STD_LOGIC;
  -- Flashing LED counter
  SIGNAL flash_counter : STD_LOGIC_VECTOR(24 downto 0);
  -- DMA Control Signals
  SIGNAL master_control_data : STD_LOGIC_VECTOR(31 downto 0);
  SIGNAL master_control_addr : STD_LOGIC;
  SIGNAL master_control_we : STD_LOGIC;
```

```
  SIGNAL writeFIFOwrite32, writeFIFOwrite64 : STD_LOGIC;
  SIGNAL readFIFOread32 : STD_LOGIC;
  SIGNAL target_readFIFO_get32bits : STD_LOGIC;
  SIGNAL master_length_counter : STD_LOGIC_VECTOR(15 downto 0);
  SIGNAL intn : STD_LOGIC;
  -- DEBUG
  SIGNAL masterstatepeek : STD_LOGIC_VECTOR(1 downto 0);
  SIGNAL peakwritelowbuffer : STD_LOGIC;
  SIGNAL peakwritehighbuffer : STD_LOGIC;
  SIGNAL peakwritereq : STD_LOGIC;
    -- PCI Transaction Counter Signals
  SIGNAL pciread32, pciread64 : STD_LOGIC;
  SIGNAL pciwrite32, pciwrite64 : STD_LOGIC;
  SIGNAL write_fifo_has_space : STD_LOGIC;
  SIGNAL read_FIFO_has2 : STD_LOGIC;
  SIGNAL debug_delayed_read : STD_LOGIC;
BEGIN
  PROCESS (hpll_nibclk)
  BEGIN
    IF rising_edge(hpll_nibclk) THEN
      flash_counter <= flash_counter + 1;
    END IF;
  END PROCESS;
  -- Assign LEDS
  led(0) <= not fifo_overflow_error;
  led(1) <= not (nib_locked AND glb_locked);
  led(2) <= flash_counter(24);
  led(3) <= '0';

  -- Place holder PCI local side signals

  intan <= '1';
  intbn <= intn;
  intcn <= '1';
  intdn <= '1';

  -- Instantiate Target Controller
  target : targetcontroller
    PORT MAP( lt_abortn, lt_discn, lt_rdyn, lt_framen, lt_ackn,
lt_dxfrn,
              lt_tsr, target_l_adi, l_adro, l_dato, l_beno,
              l_cmdo, l_ldat_ackn, l_hdat_ackn, reg_data_in,
reg_data_out,
              reg_data_addr, reg_read_ack, reg_wrreq,
              writeFIFOdatain, target_writeFIFOwrlow,
target_writeFIFOwrhigh,
              writeFIFOnearlyfull, readFIFO_dataout,
target_readFIFO_get32bits,
              readFIFO_get64bits, readFIFO_nearlyempty,
readFIFO_data32ready,
              readFIFO_data64ready, t_fifo_overflow_error, local_resetn,
rstn, clk);

  -- Instantiate Master Controller
  master : mastercontroller
    PORT MAP ( lm_req64n, lm_req32n, lm_adr_ackn, lm_tsr, lm_dxfrn,
lm_lastn,
               lm_rdyn, l_hdat_ackn,l_ldat_ackn,
               master_l_adi, master_l_adi_enable, l_cbeni, lirqn,
               master_control_data, master_control_addr,
master_control_we,
               writeFIFOnearlyfull, write_fifo_has_space,
               writeFIFOwrite64, writeFIFOwrite32,
               readFIFO_data32ready, readFIFO_dataout, readFIFOread32,
               read_FIFO_has2,
               masterstatepeek, master_length_counter,
debug_delayed_read,
               m_fifo_overflow_error, local_resetn, rstn, clk);

  fifo_overflow_error <= m_fifo_overflow_error OR t_fifo_overflow_error;

  master_control_data <= reg_data_out;

  l_adi <= master_l_adi when master_l_adi_enable = '1' else
target_l_adi;
  writeFIFOwrlow <= target_writeFIFOwrlow OR writeFIFOwrite32 OR
                                            writeFIFOwrite64;
  writeFIFOwrhigh <= target_writeFIFOwrhigh OR writeFIFOwrite64;

  readFIFO_get32bits <= target_readFIFO_get32bits OR readFIFOread32;

  -- Instantiate PCI Core
  core : mega_pci
    PORT MAP (clk, rstn, gntn, l_cbeni,idsel,l_adi,lm_req32n,
              lm_req64n, lm_lastn, lm_rdyn, lt_rdyn, lt_abortn,
lt_discn,
              lirqn, framen, irdyn, devseln, trdyn, stopn,
              req64n, ack64n, intn, reqn, serrn, l_adro,
              l_dato, l_beno, l_cmdo, l_ldat_ackn, l_hdat_ackn,
lm_adr_ackn,
              lm_ackn, lm_dxfrn, lm_tsr, lt_framen, lt_ackn, lt_dxfrn,
lt_tsr,
              cmd_reg, stat_reg, cache, ad, cben, par, par64,
              perrn);

  -- Instantiate PLL / reconfiguration circuitry / pci interface
  nibPLL : pll_reconfig_interface
    PORT MAP(clk, nibclk, l_dato, nib_pci_data_out, nib_pci_we,
             clk, nib_locked, pll_reconfig_reset,clk22);
-- DEBUG NOTE:  check if l_dato should really be reg_data_out
  glbPLL : pll_reconfig_interface
    PORT MAP(spci_clk, glbclk, l_dato, glb_pci_data_out, glb_pci_we,
             clk, glb_locked, pll_reconfig_reset,clk22);

  pll_reconfig_reset <= NOT rstn;

  -- Instantiate development JTAG interface
```

```
  dev_jtag_inst : dev_jtag
    PORT MAP(devcfg_TCK, devcfg_TDO, devcfg_TDI, devcfg_TMS,
             jtag_enable, reg_data_out, jtag_dataout, jtag_we,
             local_resetn, clk);

  -- Instantiate temperature monitor SMB circuitry
  tempmc_interface_inst : tempmc_interface
    PORT MAP( reg_data_out, tempmc_pci_data_out, tempmc_pci_we,
              smb_clk, smb_data, smb_alertn, clk, local_resetn);

  -- Instantiate clock doubling fast PLL for 64 <-> 32 conversions
  clk133pll : fastpll133 PORT MAP(clk, clk133,clk22);

  -- Instantiate read FIFO and 32 -> 64 convertor
  readFIFOinst : readFIFO
    PORT MAP ( readFIFOdata, readFIFOwrreq, hpll_nibclk,
               readFIFO_get32bits, readFIFO_get64bits,
               readFIFO_data32ready, readFIFO_data64ready,
               readFIFO_nearlyempty, readFIFO_dataout, readFIFO_level,
               read_FIFO_has2,clk133, clk, local_resetn);

  -- Instantiate write FIFO and 64 -> 32 convertor
  writeFIFOinst : writeFIFO
    PORT MAP (writeFIFOdata, writeFIFOrdreq, writeFIFOempty,
hpll_nibclk,
              writeFIFOnearlyfull, write_fifo_has_space,
writeFIFOdatain,
              writeFIFOwrhigh, writeFIFOwrlow,
              clk, clk133, local_resetn, writeFIFO_wrused,
              peakwritelowbuffer, peakwritehighbuffer,peakwritereq);

  writeFIFOrdreq <= devcfg_writeFIFOrdreq OR devbus_writeFIFOrdreq;

  -- Instantiate the development FPGA programmer
  devcfg : devconfigure
    PORT MAP (devcfg_enable, writeFIFOdata, writeFIFOempty,
              devcfg_writeFIFOrdreq, hpll_nibclk, fpga0_data,
              fpga0_dclk, fpga1_data, fpga1_dclk, fpga2_data,
fpga2_dclk,
              fpga3_data, fpga3_dclk, local_resetn);

  -- Disable the development FPGAs in an overheat situation
  fpga0_nconfig <= fpga0_nconfig_reg AND (smb_alertn OR alert_override);
  fpga1_nconfig <= fpga1_nconfig_reg AND (smb_alertn OR alert_override);
  fpga2_nconfig <= fpga2_nconfig_reg AND (smb_alertn OR alert_override);
  fpga3_nconfig <= fpga3_nconfig_reg AND (smb_alertn OR alert_override);

  -- Instantiate Development Bus Interface
  devbusinter : DevBusInterface
    Port Map (devbus_enable, readFIFOdata, readFIFOwrreq, writeFIFOdata,
              writeFIFOempty, devbus_writeFIFOrdreq, hpll_nibclk,
              local_resetn, nib0, nib1, nib2, nib3, devbus_peekstate,
              nib0_resetn, nib1_resetn, nib2_resetn, nib3_resetn);

  -- Instantiate the command register file
  cmdreg : commandRegisters
    Port Map (fpga0_nconfig_reg, fpga0_conf_done, fpga0_nstatus,
              fpga1_nconfig_reg, fpga1_conf_done, fpga1_nstatus,
              fpga2_nconfig_reg, fpga2_conf_done, fpga2_nstatus,
              fpga3_nconfig_reg, fpga3_conf_done, fpga3_nstatus,
              devcfg_enable, devbus_enable, force_resetn,
              vid_ceA, vid_ceB, vid_psaven, jtag_enable,
              nib0_resetn, nib1_resetn, nib2_resetn, nib3_resetn,
              jtag_dataout, jtag_we,
              nib_pci_we, glb_pci_we, nib_pci_data_out,
glb_pci_data_out,
              tempmc_pci_data_out, tempmc_pci_we, alert_override,
              writeFIFO_wrused, readFIFO_level,
              fifo_overflow_error, smb_alertn, nib_locked, glb_locked,
              master_control_addr, master_control_we,
              pciread32, pciread64, pciwrite32, pciwrite64, lirqn,
              reg_data_in, reg_data_out, reg_data_addr, reg_read_ack,
              reg_wrreq, clk, rstn);

  pciwrite32 <= writeFIFOwrhigh XOR writeFIFOwrlow;
  pciwrite64 <= writeFIFOwrhigh AND writeFIFOwrlow;
  pciread32 <= readFIFO_get32bits;
  pciread64 <= readFIFO_get64bits;

  local_resetn <= rstn AND force_resetn;

bkupio <= (others => '1');

END rtl;
```

C.2 COMMANDREGISTERS.VHD

```
-- Writable Register Map
--  Reg   Bit    Description
--   #   Field
--   0     0     Dev FPGA 0 nConfig signal
--   0     1     Dev FPGA 1 nConfig signal
--   0     2     Dev FPGA 2 nConfig signal
--   0     3     Dev FPGA 3 nConfig signal
--
--   1     0     Development bus interface enable
--   1     1     Development configuration mode
--   1     2     NTSC video in A chip enable
--   1     3     NTSC video in B chip enable
--   1     4     RGB video out chip enablen
--   1     5     JTAG controller enable
--   1     6     Temperature alert override
--   1     31    Force resetn
--
--   2    0      Nibble Bus 0 ResetN signal
```

```
--   2    1      Nibble Bus 1 ResetN signal
--   2    2      Nibble Bus 2 ResetN signal
--   2    3      Nibble Bus 3 ResetN signal
--
--   fc   0      Development JTAG controller TDI
--   fc   1      Development JTAG controller TMS
--
--   fd   7-0    Temp Monitors SMB: command
--   fd   15-8   Temp Monitors SMB: writedata
--   fd   16     Temp Monitors SMB: rd_req
--   fd   17     Temp Monitors SMB: wr_req
--   fd   18     Temp Monitors SMB: clr_alert_req
--   fd   31     Temp Monitors SMB: chip_sel
--
--   fe    0     Global clock PLL: reconfig
--   fe    1     Global clock PLL: read_en
--   fe    2     Global clock PLL: write_en
--   fe   7-4    Global clock PLL: counter_type
--   fe   16-8   Global clock PLL: data_in
--   fe   26-24  Global clock PLL: counter_param
--
--   ff    0     Nibble clock PLL: reconfig
--   ff    1     Nibble clock PLL: read_en
--   ff    2     Nibble clock PLL: write_en
--   ff   7-4    Nibble clock PLL: counter_type
--   ff   16-8   Nibble clock PLL: data_in
--   ff   26-24  Nibble clock PLL: counter_param

-- Readable Register Map
--  Reg   Bit    Description
--   #   Field
--   0     0     DEV FPGA 0 Conf_Done signal
--   0     1     DEV FPGA 0 nStatus signal
--   0     2     DEV FPGA 1 Conf_Done signal
--   0     3     DEV FPGA 1 nStatus signal
--   0     4     DEV FPGA 2 Conf_Done signal
--   0     5     DEV FPGA 2 nStatus signal
--   0     6     DEV FPGA 3 Conf_Done signal
--   0     7     DEV FPGA 3 nStatus signal
--
--   1    8-0    Nibble clock PLL: data_out
--   1    31     Nibble clock PLL: busy
--
--   2    8-0    Global clock PLL: data_out
--   2    31     Global clock PLL: busy
--
--   3    7-0    Temp Monitors SMB: readdata
--   3    8      Temp Monitors SMB: rd_ack
--   3    9      Temp Monitors SMB: wr_ack
--   3    10     Temp Monitors SMB: clr_alert_ack
--
--   4    31-0   TM-4 Identification
--
--   5    12-0   Write FIFO level
--
--   6    0      Fifo overflow timeout error
--   6    1      Temperature Alarm
--   6    2      Development clock pll lock
--   6    3      Global clock pll lock
--
--   7    0      Development JTAG TDO
--   7    31     Development JTAG controller busy
--
--   8           Readable copy of reg 0
--   8     0     Development bus interface enable
--   8     1     Development configuration mode
--   8     2     NTSC video in A chip enable
--   8     3     NTSC video in B chip enable
--   8     4     RGB video out chip enablen
--   8     5     JTAG controller enable
--   8     6     Temperature alert override
--   8     31    Force resetn
--
--   9     12-0  Read FIFO Level
--
--   10    31-0  PCI 32bit Read Counter
--
--   11    31-0  PCI 64bit Read Counter
--
--   12    31-0  PCI 32bit Write Counter
--
--   13    31-0  PCI 64bit Write Counter
--
--   14    0     IRQ status (0 = asserted interrupt)
--

LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_unsigned.all;

LIBRARY work;
USE work.complib.all;

ENTITY commandregisters IS
  PORT (
    -- Development configuration signals
    fpga0_nconfig : OUT STD_LOGIC;
    fpga0_conf_done : IN STD_LOGIC;
    fpga0_nstatus : IN STD_LOGIC;
    fpga1_nconfig : OUT STD_LOGIC;
    fpga1_conf_done : IN STD_LOGIC;
    fpga1_nstatus : IN STD_LOGIC;
    fpga2_nconfig : OUT STD_LOGIC;
    fpga2_conf_done : IN STD_LOGIC;
    fpga2_nstatus : IN STD_LOGIC;
    fpga3_nconfig : OUT STD_LOGIC;
    fpga3_conf_done : IN STD_LOGIC;
    fpga3_nstatus : IN STD_LOGIC;
```

```
    -- TM-4 Ctrl Register signals
    devcfg_enable : OUT STD_LOGIC;
    devbus_enable : OUT STD_LOGIC;
    force_resetn   : OUT STD_LOGIC;
    vid_ceA : OUT STD_LOGIC;
    vid_ceB : OUT STD_LOGIC;
    vid_psaven : OUT STD_LOGIC;
    jtag_enable : OUT STD_LOGIC;
    -- Nibble Bus Resetn signals
    nib0_resetn : OUT STD_LOGIC;
    nib1_resetn : OUT STD_LOGIC;
    nib2_resetn : OUT STD_LOGIC;
    nib3_resetn : OUT STD_LOGIC;
    -- JTAG controller signals
    jtag_datain : IN STD_LOGIC_VECTOR(31 downto 0);
    jtag_we : OUT STD_LOGIC;
    -- PLL reconfiguration interface signals
    nib_pci_we : OUT STD_LOGIC;
    glb_pci_we : OUT STD_LOGIC;
    nib_data_in : IN STD_LOGIC_VECTOR(31 downto 0);
    glb_data_in : IN STD_LOGIC_VECTOR(31 downto 0);
    -- Temperature monitor signals
    tempmc_data_in : IN STD_LOGIC_VECTOR(31 downto 0);
    tempmc_pci_we : OUT STD_LOGIC;
    alert_override : OUT STD_LOGIC;
    -- Write FIFO status
    writeFIFO_wrused : IN STD_LOGIC_VECTOR(12 downto 0);
    readFIFO_level : IN STD_LOGIC_VECTOR(12 downto 0);
    -- FIFO error
    fifo_overflow_error : IN STD_LOGIC;
    temp_alertn : IN STD_LOGIC;
    devpll_locked : IN STD_LOGIC;
    glbpll_locked : IN STD_LOGIC;
    -- DMA control signals
    master_control_addr : OUT STD_LOGIC;
    master_control_we : OUT STD_LOGIC;
    -- PCI Transaction Counter Signals
    pciread32 : IN STD_LOGIC;
    pciread64 : IN STD_LOGIC;
    pciwrite32 : IN STD_LOGIC;
    pciwrite64 : IN STD_LOGIC;
    -- Other signals
    lirqn : IN STD_LOGIC;
    -- PCI Interface signals
    reg_data_out : OUT STD_LOGIC_VECTOR(31 downto 0);
    reg_data_in : IN STD_LOGIC_VECTOR(31 downto 0);
    reg_data_addr : IN STD_LOGIC_VECTOR(7 downto 0);
    reg_read_ack : IN STD_LOGIC;
    reg_wrreq : IN STD_LOGIC;
    clk : IN STD_LOGIC;
    rstn : IN STD_LOGIC);
END;

ARCHITECTURE rtl OF commandRegisters IS

  TYPE writeregarray is array(255 downto 0) of STD_LOGIC_VECTOR(31
downto 0);
  TYPE readregarray is array(14 downto 0) of STD_LOGIC_VECTOR(31 downto
0);

  SIGNAL writeregisters : writeregarray;
  SIGNAL readregisters : readregarray;
BEGIN
  ----------------------------------------
  -- Assign readable register connections --
  ----------------------------------------

  -- Connect the development configuration signals to registers
  readregisters(0)(0) <= fpga0_conf_done;
  readregisters(0)(1) <= fpga0_nstatus;
  readregisters(0)(2) <= fpga1_conf_done;
  readregisters(0)(3) <= fpga1_nstatus;
  readregisters(0)(4) <= fpga2_conf_done;
  readregisters(0)(5) <= fpga2_nstatus;
  readregisters(0)(6) <= fpga3_conf_done;
  readregisters(0)(7) <= fpga3_nstatus;
  readregisters(0)(31 downto 8) <= (others => '0');

  readregisters(1) <= nib_data_in;
  readregisters(2) <= glb_data_in;
  readregisters(3) <= tempmc_data_in;
  readregisters(4) <= "01010100010011100010110100110100";
  readregisters(5)(12 downto 0) <= writeFIFO_wrused;
  readregisters(5)(31 downto 13) <= (others => '0');
  readregisters(6)(0) <= fifo_overflow_error;
  readregisters(6)(1) <= NOT temp_alertn;
  readregisters(6)(2) <= devpll_locked;
  readregisters(6)(3) <= glbpll_locked;

  readregisters(6)(31 downto 4) <= (others => '0');
  readregisters(7) <= jtag_datain;
  readregisters(8) <= writeregisters(1);

  readregisters(9)(12 downto 0) <= readFIFO_level;
  readregisters(9)(31 downto 13) <= (others => '0');

  readregisters(14)(0) <= lirqn;
  readregisters(14)(31 downto 1) <= (others => '0');
  ----------------------------------------
  -- Assign writeable register connections --
  ----------------------------------------

  -- Connect the development configuration signals to registers
  fpga0_nconfig <= writeregisters(0)(0);
  fpga1_nconfig <= writeregisters(0)(1);
  fpga2_nconfig <= writeregisters(0)(2);
  fpga3_nconfig <= writeregisters(0)(3);

  -- Connect TM-4 control register signals
```

```
  devbus_enable <= writeregisters(1)(0);
  devcfg_enable <= writeregisters(1)(1);
  vid_ceA <= writeregisters(1)(2);
  vid_ceB <= writeregisters(1)(3);
  vid_psaven <= writeregisters(1)(4);
  jtag_enable <= writeregisters(1)(5);
  alert_override <= writeregisters(1)(6);
  force_resetn <= writeregisters(1)(31);

  nib0_resetn <= writeregisters(2)(0);
  nib1_resetn <= writeregisters(2)(1);
  nib2_resetn <= writeregisters(2)(2);
  nib3_resetn <= writeregisters(2)(3);

  -- PLL reconfiguration interface signals
  master_control_addr <= reg_data_addr(0);
  master_control_we <= '1' WHEN reg_data_addr(7 downto 1) = "1111101"
AND
                              reg_wrreq = '1' ELSE '0';
  jtag_we <= '1'       WHEN reg_data_addr = "11111100" and reg_wrreq =
'1' else '0';
  tempmc_pci_we <= '1' WHEN reg_data_addr = "11111101" and reg_wrreq =
'1' else '0';
  glb_pci_we <= '1'    WHEN reg_data_addr = "11111110" and reg_wrreq =
'1' else '0';
  nib_pci_we <= '1'    WHEN reg_data_addr = "11111111" and reg_wrreq =
'1' else '0';
  -- Handle updating the register array
  reg_data_out <= readregisters(conv_integer(reg_data_addr));
  PROCESS (clk,rstn)
  BEGIN
    IF rstn = '0' THEN
       writeregisters <= (others => (others => '0'));
       readregisters(10) <= (others => '0');
       readregisters(11) <= (others => '0');
       readregisters(12) <= (others => '0');
       readregisters(13) <= (others => '0');
    ELSIF rising_edge(clk) THEN
      IF pciread32 = '1' THEN
        readregisters(10) <= readregisters(10) + 1;
      END IF;
      IF pciread64 = '1' THEN
        readregisters(11) <= readregisters(11) + 1;
      END IF;
      IF pciwrite32 = '1' THEN
        readregisters(12) <= readregisters(12) + 1;
      END IF;
      IF pciwrite64 = '1' THEN
        readregisters(13) <= readregisters(13) + 1;
      END IF;

      IF (reg_wrreq = '1') THEN
         writeregisters(conv_integer(reg_data_addr)) <= reg_data_in;
      END IF;
    END IF;
  END PROCESS;
END rtl;
```

C.3 DEV_JTAG.VHD

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.std_logic_unsigned.all;

ENTITY dev_jtag IS
  PORT (
    -- JTAG signals
    TCK : INOUT std_logic;
    TDO : IN std_logic;
    TDI : INOUT std_logic;
    TMS : INOUT std_logic;
    -- Control signals
    enable : IN std_logic;
    -- Data interface
    datain : IN std_logic_vector(31 downto 0);
    dataout : OUT std_logic_vector(31 downto 0);
    datawe : IN std_logic;

    rstn : IN std_logic;
    clk  : IN std_logic);
END dev_jtag;

ARCHITECTURE rtl OF dev_jtag IS
  SIGNAL ltck, ltdo, ltdi, ltms, busy : std_logic;
  SIGNAL count : std_logic_vector(2 downto 0);
BEGIN
  tck <= ltck when enable = '1' else 'Z';
  tdi <= ltdi when enable = '1' else 'Z';
  tms <= ltms when enable = '1' else 'Z';

  dataout(0) <= ltdo;
  dataout(30 downto 1) <= (others => '0');
  dataout(31) <= busy;

  PROCESS (clk,rstn)
  BEGIN
    IF rstn = '0' THEN
      ltdo <= '0';
      ltdi <= '0';
      ltms <= '0';
      busy <= '0';
      ltck <= '0';
      count <= "000";
    ELSIF rising_edge(clk) THEN
      ltdo <= tdo;
```

```
      IF datawe = '1' then
        busy <= '1';
        ltdi <= datain(0);
        ltms <= datain(1);
        count <= "000";
      ELSIF busy = '1' THEN
        IF count = "101" THEN
          count <= "000";
          IF ltck = '1' THEN
            busy <= '0';
          END IF;
          ltck <= not ltck;
        ELSE
          count <= count + 1;
        END IF;
      END IF;
    END IF;
  END PROCESS;
END rtl;
```

C.4 DEVBUSINTERFACE.VHD

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_arith.all;
USE ieee.std_logic_unsigned.all;

LIBRARY work;
USE work.complib.all;

ENTITY DevBusInterface IS
  PORT (
    enable : IN STD_LOGIC;
    -- PCIreadFIFO
    readFIFOdata : OUT STD_LOGIC_VECTOR(31 downto 0);
    readFIFOwrreq : OUT STD_LOGIC;
    -- PCIwriteFIFO
    writeFIFOdata : IN STD_LOGIC_VECTOR(31 downto 0);
    writeFIFOempty : IN STD_LOGIC;
    writeFIFOrdreq : OUT STD_LOGIC;
    clk : IN STD_LOGIC;
    rstn : IN STD_LOGIC;
    -- Development bus signals
    nib0 : INOUT STD_LOGIC_VECTOR(49 downto 0);
    nib1 : INOUT STD_LOGIC_VECTOR(49 downto 0);
    nib2 : INOUT STD_LOGIC_VECTOR(49 downto 0);
    nib3 : INOUT STD_LOGIC_VECTOR(49 downto 0);
    -- Current State Peek
    peek_state : OUT STD_LOGIC_VECTOR(2 downto 0);
    -- Nibble Bus Resetn signals
    nib0_resetn : IN STD_LOGIC;
    nib1_resetn : IN STD_LOGIC;
    nib2_resetn : IN STD_LOGIC;
    nib3_resetn : IN STD_LOGIC);
END;

ARCHITECTURE rtl OF DevBusInterface IS
  FUNCTION is_zero(SIGNAL input : IN STD_LOGIC) RETURN boolean IS
  BEGIN
    IF input = '0' THEN
      RETURN TRUE;
    ELSE
      RETURN FALSE;
    END IF;
  END is_zero;

  FUNCTION getTackn ( signal tackn0, tackn1 : IN STD_LOGIC;
                      signal tackn2, tackn3 : IN STD_LOGIC;
                      signal fpganum : IN STD_LOGIC_VECTOR(1 downto 0))
                      return boolean IS
  BEGIN
    return is_zero(tackn0) or is_zero(tackn1) or
           is_zero(tackn2) or is_zero(tackn3);
--    CASE fpganum IS
--      WHEN "00" => return is_zero(tackn0);
--      WHEN "01" => return is_zero(tackn1);
--      WHEN "10" => return is_zero(tackn2);
--      WHEN "11" => return is_zero(tackn3);
--    END CASE;
  END getTackn;

  FUNCTION getDataIn ( signal data0in,data1in : IN STD_LOGIC_VECTOR;
                       signal data2in,data3in : IN STD_LOGIC_VECTOR;
                       signal fpganum : IN STD_LOGIC_VECTOR(1 downto 0))
                       return std_logic_vector IS
  BEGIN
    CASE fpganum IS
      WHEN "00" => return data0in;
      WHEN "01" => return data1in;
      WHEN "10" => return data2in;
      WHEN "11" => return data3in;
    END CASE;
  END getDataIn;

  PROCEDURE AssertFrame( signal framen0, framen1 : OUT std_logic;
                         signal framen2, framen3 : OUT std_logic;
                         signal fpganum : IN std_logic_vector(1 downto
0)) IS
  BEGIN
    CASE fpganum IS
      WHEN "00" => framen0 <= '0';
      WHEN "01" => framen1 <= '0';
      WHEN "10" => framen2 <= '0';
      WHEN "11" => framen3 <= '0';
```

```
    END CASE;
  END PROCEDURE;

  PROCEDURE WriteData( signal data0, data1 : OUT std_logic_vector(31
downto 0);
                       signal data2, data3 : OUT std_logic_vector(31
downto 0);
                       signal data : IN std_logic_vector(31 downto 0);
                       signal fpganum : IN std_logic_vector(1 downto 0))
IS
  BEGIN
    data0 <= data;
    data1 <= data;
    data2 <= data;
    data3 <= data;
--    CASE fpganum IS
--      WHEN "00" => data0 <= data;
--      WHEN "01" => data1 <= data;
--      WHEN "10" => data2 <= data;
--      WHEN "11" => data3 <= data;
--    END CASE;
  END PROCEDURE;

  TYPE states IS (S_IDLE, S_READ, S_READWAIT, S_WRITE, S_ACKWAIT,
S_WRITEWAIT);
  SIGNAL curr_state : states;
  SIGNAL address : STD_LOGIC_VECTOR(7 downto 0);
  SIGNAL address0, address1 : STD_LOGIC_VECTOR(7 downto 0);
  SIGNAL address2, address3 : STD_LOGIC_VECTOR(7 downto 0);
  SIGNAL count : STD_LOGIC_VECTOR(15 downto 0);
  SIGNAL ackrequired : STD_LOGIC;
  SIGNAL framen0, framen1, framen2, framen3 : STD_LOGIC;
  SIGNAL data0, data1, data2, data3 : STD_LOGIC_VECTOR(31 downto 0);
  SIGNAL tackn0, tackn1, tackn2, tackn3 : STD_LOGIC;
  SIGNAL rst, counterload : STD_LOGIC;
  SIGNAL count_enable : STD_LOGIC;
  SIGNAL data0in, data1in, data2in, data3in : STD_LOGIC_VECTOR(31 downto
0);
  SIGNAL transactionCFG : STD_LOGIC_VECTOR(31 downto 0);
  SIGNAL widthCFG, widthCounter : STD_LOGIC_VECTOR(5 downto 0);
BEGIN
  PROCESS (curr_state)
  BEGIN
    CASE (curr_state) IS
      WHEN S_IDLE => peek_state <= "000";
      WHEN S_READ => peek_state <= "001";
      WHEN S_WRITE => peek_state <= "010";
      WHEN S_ACKWAIT => peek_state <= "011";
      WHEN S_READWAIT => peek_state <= "100";
      WHEN S_WRITEWAIT => peek_state <= "101";
    END CASE;
  END PROCESS;

  -- Assign development bus signals mappings
  -- Address1..address3 are all identical registers
  -- Four seperate registers are used as a Quartus bug work around
  PROCESS (CLK)
  BEGIN
    IF rising_edge(clk) THEN
      nib0(37 downto 32) <= address0(5 downto 0);
      nib1(37 downto 32) <= address1(5 downto 0);
      nib2(37 downto 32) <= address2(5 downto 0);
      nib3(37 downto 32) <= address3(5 downto 0);

      nib0(38) <= framen0;
      nib1(38) <= framen1;
      nib2(38) <= framen2;
      nib3(38) <= framen3;
      IF curr_state /= S_READ THEN
        nib0(31 downto 0) <= data0;
        nib1(31 downto 0) <= data1;
        nib2(31 downto 0) <= data2;
        nib3(31 downto 0) <= data3;
      ELSE
        nib0(31 downto 0) <= (others => 'Z');
        nib1(31 downto 0) <= (others => 'Z');
        nib2(31 downto 0) <= (others => 'Z');
        nib3(31 downto 0) <= (others => 'Z');
      END IF;
    END IF;
  END PROCESS;

  -- Tristate the TACKn lines
  nib0(39) <= 'Z';   nib1(39) <= 'Z';
  nib2(39) <= 'Z';   nib3(39) <= 'Z';

  nib0(40) <= nib0_resetn;
  nib1(40) <= nib1_resetn;
  nib2(40) <= nib2_resetn;
  nib3(40) <= nib3_resetn;

  nib0(49 downto 41) <= (others => '0');
  nib1(49 downto 41) <= (others => '0');
  nib2(49 downto 41) <= (others => '0');
  nib3(49 downto 41) <= (others => '0');

  -- Latch the bus signals as they come into the FPGA
  PROCESS (CLK)
  BEGIN
    IF rising_edge(clk) THEN
      tackn0 <= nib0(39);
      tackn1 <= nib1(39);
      tackn2 <= nib2(39);
      tackn3 <= nib3(39);
      data0in <= nib0(31 downto 0);
      data1in <= nib1(31 downto 0);
      data2in <= nib2(31 downto 0);
      data3in <= nib3(31 downto 0);
```

```
    END IF;
  END PROCESS;

  -- Acknowledge the data from the fifo when we read it
  writeFIFOrdreq <= '1' when (enable = '1' and writeFIFOempty = '0' and
                              (curr_state = S_IDLE or curr_state =
S_WRITE))
                         else '0';

  -- Connect to the PCIreadFIFO
  readFIFOdata <= getDataIn( data0in, data1in, data2in, data3in,
                             address(7 downto 6));
  readFIFOwrreq <= '1' when curr_state = S_READ AND
      getTackn(tackn0, tackn1, tackn2, tackn3,address(7 downto 6)) else
'0';

  -- Instantiate word counter
  counter : counter16
    PORT MAP (clk,count_enable, counterload, rst, writeFIFOdata(23
downto 8), count);
  rst <= NOT rstn;
  counterload <= '1' when (curr_state = S_IDLE) else '0';
  count_enable <= '1' when (curr_state = S_WRITE AND writeFIFOempty =
'0') or
     (curr_state = S_READ AND
      getTackn(tackn0, tackn1, tackn2, tackn3,address(7 downto 6))) else
'0';

  -- Setup stuff
  transactionCFG <= "0000000000000000" & writeFIFOdata(23 downto 8);

  PROCESS(clk,rstn,enable)
  BEGIN
    IF rstn = '0' THEN
      curr_state <= S_IDLE;
      framen0 <= '1'; framen1 <= '1';
      framen2 <= '1'; framen3 <= '1';
      widthCFG <= (others => '0');
      widthcounter <= (others => '0');
    ELSIF rising_edge(clk) THEN
      IF enable = '1' THEN
        -- Default to deasserted frame
        framen0 <= '1'; framen1 <= '1';
        framen2 <= '1'; framen3 <= '1';
        CASE curr_state IS
          WHEN S_IDLE =>
            -- Constantly latch the new address and ack request
            address <= writeFIFOdata(7 downto 0);
            address0 <= writeFIFOdata(7 downto 0);
            address1 <= writeFIFOdata(7 downto 0);
            address2 <= writeFIFOdata(7 downto 0);
            address3 <= writeFIFOdata(7 downto 0);

            ackrequired <= writeFIFOdata(30);
            widthCFG <= writeFIFOdata(29 downto 24);
            widthcounter <= "000001";

            IF (writeFIFOempty = '0') THEN
              IF (writeFIFOdata(31) = '0') THEN
                curr_state <= S_WRITE;
              ELSE
                curr_state <= S_READWAIT;
                AssertFrame(framen0, framen1, framen2, framen3,
                            writeFIFOdata(7 downto 6));
                writeData(data0,data1,data2,data3, transactionCFG,
                          writeFIFOdata(7 downto 6));
              END IF;
            END IF;
          WHEN S_READWAIT =>
             curr_state <= S_READ;
          WHEN S_WRITE =>
            IF (writeFIFOempty = '0') THEN
              widthcounter <= widthcounter + 1;
              writedata(data0,data1,data2,data3,writeFIFOdata,
                        address(7 downto 6));
              assertFrame(framen0, framen1, framen2, framen3,
                          address(7 downto 6));
              IF (widthcounter = widthCFG) THEN
                IF (ackrequired = '1') THEN
                  curr_state <= S_ACKWAIT;
                ELSE
                  IF count = CONV_STD_LOGIC_VECTOR(1,16) THEN
                    curr_state <= S_IDLE;
                  ELSE
                    widthcounter <= "000001";
                  END IF;
                END IF;
              END IF;
            END IF;
          WHEN S_ACKWAIT =>
            widthcounter <= "000001";
            IF getTackn(tackn0, tackn1, tackn2, tackn3,address(7 downto
6)) THEN
              IF count = CONV_STD_LOGIC_VECTOR(0,16) THEN
                curr_state <= S_IDLE;
              ELSE -- Need to deassert frame for one cycle
                curr_state <= S_WRITEWAIT;
              END IF;
            ELSE
              assertFrame(framen0, framen1, framen2, framen3,
                          address(7 downto 6));
            END IF;
          WHEN S_WRITEWAIT =>
            curr_state <= S_WRITE;
          WHEN S_READ =>
            IF getTackn(tackn0, tackn1, tackn2, tackn3,address(7 downto
6)) THEN
              IF count = CONV_STD_LOGIC_VECTOR(1,16) THEN
```

```
              curr_state <= S_IDLE;
            END IF;
          END IF;
        END CASE;
      END IF;
    END IF;
  END PROCESS;

END rtl;
```

C.5 DEVCONFIGURE.VHD

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

LIBRARY work;
USE work.complib.all;

ENTITY devConfigure IS
  PORT (
    enable : IN STD_LOGIC;
    -- FIFO Input Signals
    FIFOdata : IN STD_LOGIC_VECTOR(31 downto 0);
    FIFOempty : IN STD_LOGIC;
    FIFOrdreq : OUT STD_LOGIC;
    clk : IN STD_LOGIC;
    -- FPGA Configuration Outputs
    fpga0_data : OUT STD_LOGIC_VECTOR(7 downto 0);
    fpga0_dclk : OUT STD_LOGIC;
    fpga1_data : OUT STD_LOGIC_VECTOR(7 downto 0);
    fpga1_dclk : OUT STD_LOGIC;
    fpga2_data : OUT STD_LOGIC_VECTOR(7 downto 0);
    fpga2_dclk : OUT STD_LOGIC;
    fpga3_data : OUT STD_LOGIC_VECTOR(7 downto 0);
    fpga3_dclk : OUT STD_LOGIC;
    rstn : IN STD_LOGIC );
END;

ARCHITECTURE rtl OF devConfigure IS
  SIGNAL dclk : STD_LOGIC;
  SIGNAL dclkgo : STD_LOGIC;
BEGIN
  -- Using look ahead fifo so ack when we have used the data
  FIFOrdreq <= '1' WHEN (enable = '1') AND (FIFOempty = '0') else '0';

  PROCESS (clk,enable,FIFOempty)
  BEGIN
    IF rising_edge(clk) THEN
      IF (enable = '1') AND (FIFOempty = '0') THEN
        fpga0_data <= FIFOdata(7 downto 0);
        fpga1_data <= FIFOdata(15 downto 8);
        fpga2_data <= FIFOdata(23 downto 16);
        fpga3_data <= FIFOdata(31 downto 24);
        dclkgo <= '1';
      ELSE
        dclkgo <= '0';
      END IF;
    END IF;
  END PROCESS;

  fpga0_dclk <= dclk;
  fpga1_dclk <= dclk;
  fpga2_dclk <= dclk;
  fpga3_dclk <= dclk;

  dclk <= (NOT clk) AND dclkgo;

END rtl;
```

C.6 MASTERCONTROLLER.VHD

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_arith.all;
USE ieee.std_logic_unsigned.all;

LIBRARY work;
USE work.complib.all;

ENTITY mastercontroller IS
  PORT (
    -- PCI Controller interface signals
    lm_req64n : BUFFER STD_LOGIC;
    lm_req32n : OUT STD_LOGIC;
    lm_adr_ackn : IN STD_LOGIC;
    lm_tsr : IN STD_LOGIC_VECTOR(9 downto 0);
    lm_dxfrn : IN STD_LOGIC;
    lm_lastn : BUFFER STD_LOGIC;
    lm_rdyn : OUT STD_LOGIC;
    l_hdat_ackn   : in  std_logic;          -- local high data
acknowledge
    l_ldat_ackn   : in  std_logic;          -- local low data
acknowledge
    l_adi : OUT STD_LOGIC_VECTOR(63 downto 0);
    l_adi_enable : OUT STD_LOGIC;
    l_cbeni : OUT STD_LOGIC_VECTOR(7 downto 0);
    lirqn : OUT STD_LOGIC;

    -- Control Register Interface
    control_data : IN STD_LOGIC_VECTOR(31 downto 0);
    control_addr : IN STD_LOGIC;
    control_we   : IN STD_LOGIC;
```

```
    -- Write FIFO signals
    write_FIFO_nearly_full : IN STD_LOGIC;
    write_fifo_has_space : IN STD_LOGIC;
    write_FIFO_wr64 : BUFFER STD_LOGIC;
    write_FIFO_wr32 : BUFFER STD_LOGIC;

    -- Read FIFO Signals
    read_FIFO_dataready : IN STD_LOGIC;
    read_FIFO_data : IN STD_LOGIC_VECTOR(63 downto 0);
    read_FIFO_readack : BUFFER STD_LOGIC;
    read_FIFO_has2 : IN STD_LOGIC;

    -- Debug Signal
    debug_statepeek : OUT STD_LOGIC_VECTOR(1 downto 0);
    debug_length_counter : OUT STD_LOGIC_VECTOR(15 downto 0);
    debug_delayed_read : OUT STD_LOGIC;

    fifo_overflow_error : OUT STD_LOGIC;
    local_rstn : IN STD_LOGIC;
    rstn : IN STD_LOGIC;
    clk : IN STD_LOGIC);
END;

ARCHITECTURE rtl OF mastercontroller IS
  -- State Machine Variables
  TYPE states IS (S_IDLE, S_REQ, S_ADDR, S_ACTIVE);

  SIGNAL curr_state, next_state : states;

  SIGNAL target_address : STD_LOGIC_VECTOR(31 downto 2);
  SIGNAL read : STD_LOGIC;   -- Read / Not Write
  SIGNAL pci_command : STD_LOGIC_VECTOR(3 downto 0);

  -- Transaction Length Counter
  SIGNAL clear_lirqn, trans_load, count_1, count_2 : BOOLEAN;
  SIGNAL trans_length_counter : STD_LOGIC_VECTOR(15 downto 0);
  SIGNAL trans_eq_0, trans_eq_1, trans_eq_2, trans_eq_4, trans_eq_6 :
boolean;

  SIGNAL transaction_pending : BOOLEAN;
  -- PCI Controller status register bits
  SIGNAL addr_phase, data_phase, trans64, requestgnt_phase : BOOLEAN;
  SIGNAL read_from_tm4, write_to_tm4 : BOOLEAN;
  SIGNAL last_override : BOOLEAN;
  -- Fifo over/under flow counter signals
  SIGNAL disc_counter : INTEGER RANGE 0 TO 16777215;
  SIGNAL disc_count_enable : STD_LOGIC;
  SIGNAL disc_count_reset : STD_LOGIC;
  SIGNAL fifo_override : STD_LOGIC;
  SIGNAL delayed_read : boolean;
BEGIN
  -- Debugging Output Connections
  PROCESS(curr_state)
  BEGIN
    CASE(curr_state) IS
      WHEN S_IDLE   => debug_statepeek <= "00";
      WHEN S_REQ    => debug_statepeek <= "01";
      WHEN S_ADDR   => debug_statepeek <= "10";
      WHEN S_ACTIVE => debug_statepeek <= "11";
    END CASE;
  END PROCESS;
  debug_length_counter <= trans_length_counter;
  debug_delayed_read <= '1' WHEN delayed_read else '0';

  -- Actual Module Code Starts Here

  -- FIFO over/under flow error catcher
  disc_count_enable <= '1' WHEN NOT transaction_pending AND
(trans_length_counter /= 0)
                       ELSE '0';

  disc_count_reset  <= '1' WHEN curr_state = S_IDLE ELSE '0';
  fifo_overflow_error <= fifo_override;

  PROCESS (clk,local_rstn)
  BEGIN
    IF (local_rstn = '0') THEN
      disc_counter <= 0;
      fifo_override <= '0';
    ELSIF rising_edge(clk) THEN
      IF disc_count_reset = '1' THEN
         disc_counter <= 0;
      ELSIF (disc_count_enable = '1') THEN
         disc_counter <= disc_counter + 1;
      END IF;

      IF (disc_counter = 16777215) THEN
        fifo_override <= '1';
      END IF;
    END IF;
  END PROCESS;

  -- Assign PCI Core signals
  lm_req64n <= '0' WHEN curr_state = S_REQ AND write_to_tm4 ELSE '1';
  lm_req32n <= '0' WHEN curr_state = S_REQ AND read_from_tm4 ELSE '1';

  l_adi(31 downto 0) <= target_address & "00" WHEN lm_adr_ackn = '0'
                        ELSE read_FIFO_data(31 downto 0);

  l_adi(63 downto 32) <= read_FIFO_data(63 downto 32);

  l_adi_enable <= '1' WHEN (curr_state /= S_IDLE) ELSE '0';

  l_cbeni(3 downto 0) <= pci_command WHEN (lm_adr_ackn = '0') ELSE
"0000";
  l_cbeni(7 downto 4) <= "0000";
```

```
  -- Decode Read / Not Write into a equivalent PCI command nibble
  -- then 32bytes before disconnect

  pci_command <= "1100" WHEN write_to_tm4 ELSE "0111";

  -- Provide easier to read names for status bits

  write_to_tm4 <= (read = '0');
  read_from_tm4 <= (read = '1');

  requestgnt_phase <= (lm_tsr(0) = '1' OR lm_tsr(1) = '1');
  addr_phase <= (lm_tsr(2) = '1');
  data_phase <= (lm_tsr(3) = '1');
  trans64 <= (lm_tsr(9) = '1');

  transaction_pending <= (trans_length_counter /= 0) AND (
                         ((write_to_tm4 AND write_fifo_has_space = '1')
OR
                          (read_from_tm4 AND read_fifo_dataready = '1'))
OR
                           fifo_override = '1');

  -- Configuration register loading
  PROCESS (clk, rstn)
  BEGIN
    IF rstn = '0' THEN
      curr_state <= S_IDLE;

    ELSIF rising_edge(clk) THEN
      curr_state <= next_state;
      IF control_we = '1' THEN
        IF control_addr = '0' THEN
          target_address <= control_data(31 downto 3) & "0";
        ELSE
          read <= control_data(31);
        END IF;
      ELSIF count_1 THEN
        target_address <= target_address + 1;
      ELSIF count_2 THEN
        target_address <= target_address + 2;
      END IF;
    END IF;
  END PROCESS;
  trans_load <= (control_we = '1' and control_addr = '1');
  clear_lirqn <= (control_we = '1' and control_addr = '0');

  -- Transaction Length Counter
  PROCESS (clk, rstn)
  BEGIN
    IF rstn = '0' THEN
      trans_length_counter <= (others => '0');
      lirqn <= '1';
    ELSIF rising_edge(clk) THEN
      IF clear_lirqn THEN
        lirqn <= '1';
      ELSIF trans_load THEN
        trans_length_counter <= control_data(15 downto 0);
      ELSIF count_1 THEN
        trans_length_counter <= trans_length_counter - 1;
        IF trans_eq_1 THEN
          lirqn <= '0';
        END IF;
      ELSIF count_2 THEN
        trans_length_counter <= trans_length_counter - 2;
        IF trans_eq_2 THEN
          lirqn <= '0';
        END IF;
      END IF;
    END IF;
  END PROCESS;

  trans_eq_0 <= (trans_length_counter = 0);
  trans_eq_1 <= (trans_length_counter = 1);
  trans_eq_2 <= (trans_length_counter = 2);
  trans_eq_4 <= (trans_length_counter = 4);
  trans_eq_6 <= (trans_length_counter = 6);

  -- Note: Currently performing 32bit Master writes only
  count_1 <= (write_FIFO_wr32 = '1') OR (read_FIFO_readack = '1');
  count_2 <= (write_FIFO_wr64 = '1');

  -- Data FIFO Transfer Logic
  write_FIFO_wr32 <= '1' WHEN write_to_tm4 AND (lm_dxfrn = '0') AND NOT
trans64
                         ELSE '0';
  write_FIFO_wr64 <= '1' WHEN write_to_tm4 AND (lm_dxfrn = '0') AND
trans64
                         ELSE '0';
  read_FIFO_readack <= '1' WHEN read_from_tm4 AND (data_phase OR
addr_phase) AND (
                              ((lm_dxfrn = '0') AND ((read_fifo_has2 =
'1') OR (fifo_override = '1'))) OR
                               (delayed_read AND ((read_fifo_has2 = '1')
or (fifo_override = '1'))) OR
                               (lm_dxfrn = '0' AND trans_eq_1)) ELSE '0';

  PROCESS (clk,rstn)
  BEGIN
    if rstn = '0' THEN
      delayed_read <= false;
    ELSIF rising_edge(clk) THEN
      IF delayed_read AND (read_fifo_has2 = '1') THEN
        delayed_read <= false;
      ELSIf lm_dxfrn = '0' AND read_from_tm4 AND (read_fifo_has2 = '0')
THEN
        delayed_read <= true;
      END IF;
    END IF;
```

```
  END PROCESS;

  -- Last Rdyn Override
  PROCESS(clk,rstn)
  BEGIN
    IF rstn = '0' THEN
      last_override <= FALSE;
    ELSIF rising_edge(clk) THEN
      IF lm_lastn = '0' THEN
        last_override <= TRUE;
      ELSIF NOT data_phase THEN
        last_override <= FALSE;
      END IF;
    END IF;
  END PROCESS;

  -- Local side wait/ data ready logic
  lm_rdyn <= '0' WHEN (write_to_tm4 AND curr_state = S_ACTIVE AND
                       (write_FIFO_nearly_full = '0' OR last_override or
fifo_override = '1')) OR
                      (read_from_tm4 AND
                        (lm_adr_ackn = '0' OR
                         (data_phase AND NOT trans_eq_1 AND NOT
trans_eq_0 AND
                          ((read_fifo_has2 = '1') OR fifo_override =
'1'))
                         )
                       )
                 ELSE '1';

--  lm_rdyn <= '0' WHEN (write_to_tm4 AND curr_state = S_ACTIVE AND
--                       (write_FIFO_nearly_full = '0' OR last_override
or fifo_override = '1')) OR
--                      (read_from_tm4 AND
--                        (curr_state = S_ACTIVE OR curr_state = S_ADDR)
AND
--                         ((read_fifo_dataready = '1' AND readFIFOlast =
'0')
--                            OR fifo_override = '1') AND
--                          NOT last_override)
--                 ELSE '1';  -- AND NOT trans_eq_2


  -- Local side transaction termination logic
--  last : last_gen PORT MAP
--    (lm_lastn => lm_lastn,
--     clk => clk,
--     rstn => rstn,
--     wr_rdn => read,  -- Make sure this is correct
--     lm_req64n => lm_req64n,
--     lm_dxfrn => lm_dxfrn,
--     l_hdat_ackn => l_hdat_ackn,
--     l_ldat_ackn => l_ldat_ackn,
--     lm_tsr => lm_tsr,
--     xfr_length => trans_length_counter);

  lm_lastn <= '0' WHEN (write_to_tm4 AND
                         ( ((lm_adr_ackn  = '0') AND (trans_eq_2 OR
trans_eq_4)) OR
                           (curr_state = S_ACTIVE AND (lm_dxfrn = '0')
AND
                            ((trans_eq_6 AND trans64) OR (trans_eq_4 AND
NOT trans64))
                           )
                         )OR
                        (read_from_tm4 AND (lm_dxfrn = '0') AND
trans_eq_1)
                  ELSE '1';

  -- Control Statemachine
  PROCESS(curr_state, transaction_pending, requestgnt_phase, addr_phase,
          data_phase)
  BEGIN
    CASE (curr_state) IS
      WHEN S_IDLE =>
        IF transaction_pending THEN
          next_state <= S_REQ;
        ELSE
          next_state <= S_IDLE;
        END IF;
      WHEN S_REQ =>
        IF requestgnt_phase THEN
          next_state <= S_ADDR;
        ELSE
          next_state <= S_REQ;
        END IF;
      WHEN S_ADDR =>
        IF addr_phase THEN
          next_state <= S_ACTIVE;
        ELSE
          next_state <= S_ADDR;
        END IF;
      WHEN S_ACTIVE =>
        IF NOT data_phase THEN
          next_state <= S_IDLE;
        ELSE
          next_state <= S_ACTIVE;
        END IF;
    END CASE;
  END PROCESS;

END rtl;
```

C.7 PLL_RECONFIG_INTERFACE.VHD

```
--  TM4: Bridge FPGA - Reconfigurable PLLs & PCI interface circuit
--
--  Author: Josh Fender
--
--
--
--  Description:
--
--    Instantiates both an enhanced PLL and PLL_RECONFIG megafunction
--
--    and provides wrapper circuitry to interface with the PCI core
--
--
--
--  TODO:
--
--    - Finish documentation about signal ports
--
--
--

LIBRARY ieee;
USE ieee.std_logic_1164.all;

LIBRARY work;
USE work.complib.all;

ENTITY pll_reconfig_interface IS
  PORT (
    -- PLL clocks
    clk_in : IN STD_LOGIC;
    clk_out : OUT STD_LOGIC;
    -- Bus interface signals
    pci_data_in : IN STD_LOGIC_VECTOR(63 downto 0);
    pci_data_out : OUT STD_LOGIC_VECTOR(31 downto 0);
    pci_we : IN STD_LOGIC;
    pci_clk : IN STD_LOGIC;
    -- Misc signals
    locked : OUT STD_LOGIC;
    reset : IN STD_LOGIC;
    clk22 : IN STD_LOGIC);
END;

ARCHITECTURE rtl OF pll_reconfig_interface IS
  -- PLL signals
  SIGNAL scanaclr, scandata, scanclk : STD_LOGIC;
  SIGNAL c0_clk, scandataout : STD_LOGIC;
  -- PLL_reconfig signals
  SIGNAL reconfig, read_en : STD_LOGIC;
  SIGNAL write_en, busy : STD_LOGIC;
  SIGNAL counter_type : STD_LOGIC_VECTOR (3 downto 0);
  SIGNAL data_in, data_out : STD_LOGIC_VECTOR (8 downto 0);
  SIGNAL counter_param : STD_LOGIC_VECTOR (2 downto 0);
  SIGNAL command_wait : STD_LOGIC_VECTOR(1 downto 0);
BEGIN
  PLL : enhancedpll
    PORT MAP (clk_in, scanaclr, scandata, scanclk, c0_clk,
              scandataout, locked, clk_out);

  reconfig_inst : pll_reconfig
    PORT MAP (reconfig, counter_type, scandataout, read_en,
              reset, data_in, clk22, counter_param, write_en,
              scanclk, scanaclr, busy, data_out, scandata);

  -- Combine the necessary signals into a single PCI read
  -- only register
  pci_data_out(8 downto 0) <= data_out;
  pci_data_out(30 downto 9) <= (others => '0');
  pci_data_out(31) <= busy;

  -- Extract the control signals from a PCI write register
  -- request and register results.  The registering is redundant
  -- but helps to meet timing
  PROCESS (pci_clk)
  BEGIN
    IF rising_edge(pci_clk) THEN
      IF (pci_we = '1') THEN
        reconfig <= pci_data_in(0);
        read_en <= pci_data_in(1);
        write_en <= pci_data_in(2);
        counter_type <= pci_data_in(7 downto 4);
        data_in <= pci_data_in(16 downto 8);
        counter_param <= pci_data_in(26 downto 24);
        command_wait <= "11";
      ELSIF (command_wait = "00") THEN
        -- We need to clear the enable signals once the configuration
        -- circuitry sees it to insure the command only executes once
        read_en <= '0';
        write_en <= '0';
        reconfig <= '0';
      ELSE
        command_wait(1) <= command_wait(0);
        command_wait(0) <= '0';
      END IF;
    END IF;
  END PROCESS;

END rtl;
```

C.8 READFIFO.VHD

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

LIBRARY work;
USE work.complib.all;

ENTITY readfifo IS
  PORT (
    -- Input side signals (nib domain)
    dataIn : IN STD_LOGIC_VECTOR(31 downto 0);
    wrreq : IN STD_LOGIC;
    nibclk : IN STD_LOGIC;
    -- Output side signals (PCI domain)
    get32bits : IN STD_LOGIC;
    get64bits : IN STD_LOGIC;
    data32ready : OUT STD_LOGIC;
    data64ready : OUT STD_LOGIC;
    nearlyempty : OUT STD_LOGIC;
    dataOut : OUT STD_LOGIC_VECTOR(63 downto 0);
    fifolevel : OUT STD_LOGIC_VECTOR(12 downto 0);
    readFIFOhas2 : OUT STD_LOGIC;

    pciclk2x : IN STD_LOGIC;
    pciclk : IN STD_LOGIC;
    rstn : IN STD_LOGIC);
END;

ARCHITECTURE rtl OF readfifo IS
  -- FIFO Signals
  SIGNAL fifo_data_out : STD_LOGIC_VECTOR(31 downto 0);
  SIGNAL rdreq, rdempty, wrfull : STD_LOGIC;
  SIGNAL aclr : STD_LOGIC;
  -- State Signals
  SIGNAL data_lo_valid, data_hi_valid : STD_LOGIC;
  SIGNAL pause : STD_LOGIC;
  -- Data signals
  SIGNAL data_lo, data_hi : STD_LOGIC_VECTOR(31 downto 0);
  SIGNAL l_fifolevel : STD_LOGIC_VECTOR(12 downto 0);
BEGIN
  process(pciclk)
  BEGIN
    IF rising_edge(pciclk) THEN
      fifolevel <= l_fifolevel;
    END IF;
  END PROCESS;

  -- Instantiate fifo
  fifo : lpm_read_fifo
    PORT MAP( datain, wrreq, rdreq, pciclk, nibclk, aclr,
              fifo_data_out, rdempty, l_fifolevel,wrfull);
  aclr <= NOT rstn;

  nearlyempty <= rdempty;
--  rdreq <= '1' WHEN (rdempty = '0') AND (get32bits = '1') else '0';
  rdreq <= '1' WHEN (get32bits = '1') else '0';
  readFIFOhas2 <= '1' when l_fifolevel /= "0000000000001" AND
                           l_fifolevel /= "0000000000000" AND
                           rdempty = '0' ELSE '0';

  dataout(63 downto 32) <= fifo_data_out;
  dataout(31 downto 0) <= fifo_data_out;
  data32ready <= NOT rdempty;
  data64ready <= '0';

END rtl;
```

C.9 TARGECONTROLLER.VHD

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

LIBRARY work;
USE work.complib.all;

ENTITY targetcontroller IS
  PORT (
    -- Altera PCI Core Local Side Target Signals
    lt_abortn : OUT STD_LOGIC;
    lt_discn : OUT STD_LOGIC;
    lt_rdyn : OUT STD_LOGIC;
    lt_framen : IN STD_LOGIC;
    lt_ackn : IN STD_LOGIC;
    lt_dxfrn : IN STD_LOGIC;
    lt_tsr : IN STD_LOGIC_VECTOR(11 downto 0);

    -- Altera PCI Core Local Side Addr/Data Signals
    l_adi : OUT STD_LOGIC_VECTOR(63 downto 0);
    l_adro : IN STD_LOGIC_VECTOR(63 downto 0);
    l_dato : IN STD_LOGIC_VECTOR(63 downto 0);
    l_beno : IN STD_LOGIC_VECTOR(7 downto 0);
    l_cmdo : IN STD_LOGIC_VECTOR(3 downto 0);
    l_ldat_ackn : IN STD_LOGIC;
    l_hdat_ackn : IN STD_LOGIC;

    -- Control register interface
    reg_data_in : IN STD_LOGIC_VECTOR(31 downto 0);
    reg_data_out : OUT STD_LOGIC_VECTOR(31 downto 0);
    reg_data_addr : OUT STD_LOGIC_VECTOR(7 downto 0);
    reg_read_ack : OUT STD_LOGIC;
    reg_wrreq : OUT STD_LOGIC;
```

```
    -- Bar1 FIFO interface signals
    writeFIFOdata : OUT STD_LOGIC_VECTOR(63 downto 0);
    writeFIFOlow : OUT STD_LOGIC;
    writeFIFOhigh : OUT STD_LOGIC;
    writeFIFOnearlyfull : IN STD_LOGIC;

    readFIFOdata : IN STD_LOGIC_VECTOR(63 downto 0);
    readFIFO_get32 : OUT STD_LOGIC;
    readFIFO_get64 : OUT STD_LOGIC;
    readFIFOnearlyempty : IN STD_LOGIC;
    readFIFO_data32ready : IN STD_LOGIC;
    readFIFO_data64ready : IN STD_LOGIC;

    -- Other signals
    fifo_overflow_error : OUT STD_LOGIC;
    local_rstn : IN STD_LOGIC;
    rstn : IN STD_LOGIC;
    clock : IN STD_LOGIC);
END;

ARCHITECTURE rtl OF targetcontroller IS
  TYPE states IS (S_IDLE, S_ASSERTREADY,S_ACTIVE, S_DISCONNECT);

  SIGNAL curr_state, next_state : states;
  SIGNAL memread, memwrite, bar0, bar1,  burst,trans64 : STD_LOGIC;
  SIGNAL bar0write,bar0read : STD_LOGIC;
  SIGNAL bar1write,bar1read : STD_LOGIC;
  SIGNAL readFIFOdataready : STD_LOGIC;
  SIGNAL highTransfer : STD_LOGIC;
  SIGNAL temp : STD_LOGIC_VECTOR(63 downto 0);
  -- Fifo over/under flow counter signals
  SIGNAL disc_counter : INTEGER RANGE 0 TO 16777215;
  SIGNAL disc_count_enable : STD_LOGIC;
  SIGNAL disc_count_reset : STD_LOGIC;
  SIGNAL fifo_override : STD_LOGIC;
BEGIN
  -- Setup the BAR1 fifo writes
  writeFIFOdata <= l_dato(63 downto 0);
  writeFIFOlow <= bar1write AND (NOT l_beno(0));
  writeFIFOhigh <= bar1write AND trans64 AND (NOT l_beno(4));

  -- Setup the register write/read signals
  highTransfer <= (NOT l_beno(4) AND trans64) OR l_adro(2);

  reg_data_out <= l_dato(63 downto 32) when highTransfer = '1' else
                  l_dato(31 downto 0);
  reg_data_addr <= l_adro(9 downto 3) & (highTransfer or l_adro(2));
  reg_wrreq <= bar0write;

  -- Setup the read connections
  PROCESS (Clock)
  BEGIN
    IF rising_edge(clock) THEN
      IF bar0 = '1' THEN
        l_adi(31 downto 0) <= reg_data_in;
        l_adi(63 downto 32) <= reg_data_in;
      ELSE -- IF bar1 = '1' THEN
        l_adi <= readFIFOdata;
      END IF;

      -- Setup the register read ack signal
      reg_read_ack <= bar0read;
      readFIFO_get32 <= bar1read AND (NOT l_beno(0));
      readFIFO_get64 <= bar1read AND trans64 AND (NOT l_beno(4));
    END IF;
  END PROCESS;

  -- Command decoder
  memread <= '1' when (l_cmdo = "0110") or (l_cmdo = "1100") or (l_cmdo
= "1110") else '0';
  memwrite <= '1' when (l_cmdo = "0111") or (l_cmdo = "1111") else '0';
  burst <= lt_tsr(9);  -- Need to know so we can issue a disconnect
  trans64 <= lt_tsr(7);

  -- Decode addressing phase
  bar0 <= '1' when (lt_tsr(0) = '1') and (lt_framen = '0') else '0';
  bar1 <= '1' when (lt_tsr(1) = '1') and (lt_framen = '0') else '0';

  -- Drive local side control signals
  lt_rdyn <= '0' WHEN (curr_state = S_ACTIVE OR
                       curr_state = S_ASSERTREADY) ELSE '1';
  lt_abortn <= '1';

  -- Backend interface signals
  bar0write <= bar0 AND memwrite AND NOT lt_dxfrn;
  bar0read <= bar0 AND memread AND NOT lt_dxfrn;
  bar1write <= bar1 AND memwrite AND NOT lt_dxfrn;
  bar1read <= bar1 AND memread AND NOT lt_dxfrn;

  -- FIFO over/under flow error catcher
  disc_count_enable <= '1' WHEN curr_state = S_DISCONNECT ELSE '0';
  disc_count_reset  <= '1' WHEN curr_state = S_ASSERTREADY ELSE '0';
  fifo_overflow_error <= fifo_override;

  PROCESS (clock,local_rstn)
  BEGIN
    IF (local_rstn = '0') THEN
      disc_counter <= 0;
      fifo_override <= '0';
    ELSIF rising_edge(clock) THEN
      IF disc_count_reset = '1' THEN
         disc_counter <= 0;
      ELSIF (disc_count_enable = '1') THEN
         disc_counter <= disc_counter + 1;
      END IF;

      IF (disc_counter = 16777215) THEN
```

```
        fifo_override <= '1';
      END IF;
    END IF;
  END PROCESS;

  -- BAR0 Statemachine
  PROCESS (clock,rstn)
  BEGIN
    IF (rstn = '0') THEN
      curr_state <= S_IDLE;
    ELSIF rising_edge(clock) THEN
      curr_state <= next_state;
    END IF;
  END PROCESS;

  -- Some simplification signals
  readFIFOdataready <= readFIFO_data32ready AND (readFIFO_data64ready OR
NOT trans64);

  PROCESS (curr_state, lt_ackn, burst, bar0,bar1,memread,
           readfifodataready,memwrite,writefifonearlyfull,
           readfifonearlyempty, lt_framen, fifo_override)
  BEGIN
    lt_discn <= '1';
    CASE (curr_state) IS
      WHEN S_IDLE =>
        -- Check if we have fifo data/space for a bar1 access
        IF (bar1 = '1') AND
            ((memread = '1' AND readFIFOdataready = '0') OR
             (memwrite = '1' AND writeFIFOnearlyfull = '1')) AND
             (fifo_override = '0') THEN
          -- Issue a PCI retry
          lt_discn <= '0';
          next_state <= S_DISCONNECT;
        ELSIF (bar0 = '1' OR bar1 = '1') THEN
          next_state <= S_ASSERTREADY;
        ELSE
          next_state <= S_IDLE;
        END IF;
      WHEN S_ASSERTREADY =>
        IF (bar0 = '1') AND (burst = '1') THEN
          next_state <= S_DISCONNECT;
          lt_discn <= '0';
        ELSE
          next_state <= S_ACTIVE;
        END IF;
      WHEN S_ACTIVE =>
        IF (bar1 = '1') AND burst = '1' and
           ((memread = '1' AND readFIFOnearlyempty = '1') OR
            (memwrite = '1' AND writeFIFOnearlyfull = '1')) AND
            (fifo_override = '0') THEN
          next_state <= S_DISCONNECT;
          lt_discn <= '0';
        ELSIF (lt_framen = '1') THEN
          next_state <= S_IDLE;
        ELSE
          next_state <= S_ACTIVE;
        END IF;
      WHEN S_DISCONNECT =>
        IF (lt_framen = '1') THEN
          next_state <= S_IDLE;
        ELSE
          next_state <= S_DISCONNECT;
        END IF;
    END CASE;
  END PROCESS;

END rtl;
```

C.10 TEMPMC_INTERFACE.VHD

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

LIBRARY work;
USE work.complib.all;

ENTITY tempmc_interface IS
  PORT (
    -- PCI interface signals
    pci_data_in : IN STD_LOGIC_VECTOR(31 downto 0);
    pci_data_out : OUT STD_LOGIC_VECTOR(31 downto 0);
    pci_we : STD_LOGIC;
    -- Temp Monitors Signals
    smb_clk : INOUT STD_LOGIC;
    smb_data : INOUT STD_LOGIC;
    smb_alertn : IN STD_LOGIC;

    clk : IN STD_LOGIC;
    rstn : IN STD_LOGIC);
END tempmc_interface;

ARCHITECTURE rtl OF tempmc_interface IS
  SIGNAL command, writedata, readdata : std_logic_vector(7 downto 0);
  SIGNAL rd_req, rd_ack, wr_req, wr_ack : STD_LOGIC;
  SIGNAL clr_alert_req, clr_alert_ack : STD_LOGIC;
  SIGNAL chip_sel : STD_LOGIC;
BEGIN
  tempmc_inst : tempmc
     PORT MAP( rstn, smb_clk, smb_data, command, readdata, writedata,
rd_req,
               rd_ack, wr_req, wr_ack, clr_alert_req, clr_alert_ack,
               clk, chip_sel, smb_alertn );
  -- Combine outputs to pci_data_out
```

```
  pci_data_out(7 downto 0) <= readdata;
  pci_data_out(8) <= rd_ack;
  pci_data_out(9) <= wr_ack;
  pci_data_out(10) <= clr_alert_ack;
  pci_data_out(31 downto 11) <= (others => '0');

  -- Process the data writes
  PROCESS (clk)
  BEGIN
    IF rising_edge(clk) THEN
       IF pci_we = '1' THEN
         command <= pci_data_in(7 downto 0);
         writedata <= pci_data_in(15 downto 8);
         rd_req <= pci_data_in(16);
         wr_req <= pci_data_in(17);
         clr_alert_req <= pci_data_in(18);
         chip_sel <= pci_data_in(31);
       END IF;
    END IF;
  END PROCESS;
END rtl;
```

C.11 WRITEFIFO.VHD

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

LIBRARY work;
USE work.complib.all;

ENTITY writefifo IS
  PORT (
    -- Output side signals (nib domain)
    dataout : OUT STD_LOGIC_VECTOR(31 downto 0);
    rdreq : IN STD_LOGIC;
    rdempty : OUT STD_LOGIC;
    nibclk : IN STD_LOGIC;
    -- Input side signals (PCI domain)
    wrnearlyfull : OUT STD_LOGIC;
    write_fifo_has_space : OUT STD_LOGIC;
    datain : IN STD_LOGIC_VECTOR(63 downto 0);
    writehigh : IN STD_LOGIC;
    writelow : IN STD_LOGIC;
    pciclk : IN STD_LOGIC;
    pciclk2x : IN STD_LOGIC;
    rstn : IN STD_LOGIC;
    wrused : BUFFER STD_LOGIC_VECTOR(12 downto 0);
    -- DEBUG Signals
    peakwritelowbuffer : OUT STD_LOGIC;
    peakwritehighbuffer : OUT STD_LOGIC;
    peakwritereq : OUT STD_LOGIC);
END;

ARCHITECTURE rtl OF writefifo IS
  -- Fifo signals
  SIGNAL fifo_data_in : std_logic_vector(31 downto 0);
  SIGNAL wrreq, aclr : std_logic;
  -- Other signals
  SIGNAL databuffer : STD_LOGIC_VECTOR(63 downto 0);
  SIGNAL writehighbuffer : STD_LOGIC;
  SIGNAL writelowbuffer : STD_LOGIC;
BEGIN
  peakwritelowbuffer <= writelowbuffer;
  peakwritehighbuffer <= writehighbuffer;
  peakwritereq <= wrreq;

  -- Instantiate fifo
  fifo : lpm_write_fifo
    PORT MAP( fifo_data_in, wrreq, rdreq, nibclk, pciclk2x, aclr,
dataout,
              rdempty, wrused);
  aclr <= NOT rstn;

  -- FIFO control signals
--  fifo_data_in <= databuffer(31 downto 0) WHEN (writelowbuffer = '1')
ELSE
--                  databuffer(63 downto 32);
  fifo_data_in <= databuffer(31 downto 0) WHEN (pciclk = '1') ELSE
                  databuffer(63 downto 32);

  write_fifo_has_space <= '1' when wrused(12) = '0' else '0';

  wrreq <= '1' WHEN (writelowbuffer = '1') OR (writehighbuffer = '1' AND
pciclk = '0') ELSE '0';
  wrnearlyfull <= '1' WHEN (wrused(12 downto 2) = "11111111111") else
'0';

  -- Buffer the input signal using the slow clock
  PROCESS(pciclk)
  BEGIN
    IF rising_edge(pciclk) THEN
      databuffer <= datain;
    END IF;
  END PROCESS;

  PROCESS(pciclk2x,rstn)
  BEGIN
    IF (rstn = '0') THEN
      writelowbuffer <= '0';
      writehighbuffer <= '0';
    ELSIF rising_edge(pciclk2x) THEN
      IF (writelowbuffer = '1') AND (pciclk = '1') THEN
        writelowbuffer <= '0';
      ELSIF pciclk = '0' THEN
        writelowbuffer <= writelow;
```

```
      writehighbuffer <= writehigh;
    END IF;

  END IF;
END PROCESS;
```

```
END rtl;
```

D DEVELOPMENT BUS VHDL CODE

D.1 DEVREAD.VHD

```
--  TM4: Development Bus Parameterized Bus Abstraction Module
--  Author: Gary Pong / Josh Fender
--
--  Command: Direct Read (no handshake)
--
--  Description:
--    Provides a parameterized width register on a development FPGA
--    that user circuits can use as a handshake free output
--
--  User Signals:
--     data : std_logic_vector(dataWidth - 1 downto 0)
--        - parameterized width register's output
--
--  Module Parameters:
--     dataWidth    Width, in bits, of desired register
--     readCycles  Number of 32bit development bus transactions
--                  necessary for a dataWidth read
--                  =  Ceiling(dataWidth/32)
--     readPow2    A flag indicated if readCycles is a power of 2
--                    - 1 if power of 2
--                    - 0 if not a power of 2
--     portAddr     An 6 bit std_logic_vector that indicates the
--                  address of the abstracted register port
--
--  Bus Protocol
--    The direct read transaction consists of [readCycles] different
--    32bit read cycles.  Reads are ordered from LSB to MSB.  The bus
--    master drives the address lines and asserts the FRAMEn signal for
--    one cycle.  After a variable number of idle bus cycles, possibly
--    zero, the target device assert TACKn, latches the data input port
--    into an internal buffer and transmits the values 32 bits at a time
--    across the development bus.

LIBRARY ieee;
USE ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;

ENTITY DevRead IS
  GENERIC (
     dataWidth : natural;
--     readCycles : natural;   -- = CEIL(Datawidth/32)
--     readPow2 : natural;     -- = 1 if readCycles in a power of 2
--     portAddr : natural
  );
  PORT (
     -- Development Bus Signals
--     resetn : in std_logic;
--     address : in std_logic_vector(5 downto 0);
--     datain : in std_logic_vector(31 downto 0);
--     dataout : out std_logic_vector(31 downto 0);
--     oe : out std_logic;
--     framen : in std_logic;
--     tackn : out std_logic;
--     devclk : in std_logic;
--     -- User interface signals
--     data : in std_logic_vector(datawidth - 1 downto 0)
  );
END;

ARCHITECTURE rtl of DevRead IS
  SIGNAL databuffer : std_logic_vector( (readCycles-1)*32 downto 0);
  SIGNAL readcount : natural range 0 to readCycles-1;
  SIGNAL readEn : std_logic;
BEGIN
  -- Control Logic
  readEn <= '1' WHEN (framen = '0') AND (address = portaddr) ELSE '0';

  PROCESS (resetn,devclk,readEn)
  BEGIN
    IF (resetn = '0') THEN
      readcount <= 0;
      databuffer <= (others => '0');
      tackn <= '1';
      oe <= '0';
```

```
      dataout <= (others => '0');
    ELSIF (rising_edge(devclk)) THEN
      IF (readEn = '1' or readcount /= 0) THEN
        tackn <= '0';
        oe <= '1';

        -- Handle data transmission and buffering
        IF (readcount = 0) THEN
          IF ( readCycles > 1) THEN
            -- if multiple cycles required, buffer data
            databuffer(datawidth - 1 - 32 downto 0) <= data(datawidth - 1
downto 32);
            dataout <= data(31 downto 0);
          ELSE
            -- directly output all data
            dataout(datawidth - 1 downto 0) <= data;
          END IF;
        ELSE
          -- map databuffer contents to dataout, 32 bits at a time
          readloop : FOR k IN 1 TO readCycles-1 LOOP
            IF (readcount = k) THEN
              dataout <= databuffer(k*32-1 downto (k-1)*32);
            END IF;
          END LOOP readloop;
        END IF;

        -- Update the read cycle counters
        IF (readcount = readCycles-1) THEN
          -- If readCycles is power of 2 the counter will auto wrap
          IF (readPow2 = 1) THEN
            readcount <= readcount + 1;
          ELSE
            readcount <= 0;
          END IF;
        ELSE
          readcount <= readcount + 1;
        END IF;
      ELSE
        tackn <= '1';
        oe <= '0';
        dataout <= (others => '0');
      END IF;
    END IF;
  END PROCESS;

END rtl;
```

D.2 DEVREADBURST.VHD

```
--  TM4: Development Bus Parameterized Bus Abstraction Module           --
--  Author: Gary Pong / Josh Fender                                     --
--                                                                      --
--  Command: Direct Read (with handshake)                                 --
--                                                                        --
--  Description:                                                          --
--    Provides a parameterized width port on a development FPGA           --
--    that user circuits can use as an output with flow control.          --
--                                                                        --
--  User Signals:                                                         --
--     data : std_logic_vector(dataWidth - 1 downto 0);                   --
--        - parameterized width register's output                         --
--                                                                        --
--  Module Parameters:                                                    --
--     dataWidth   Width, in bits, of desired register                    --
--     readCycles  Number of 32bit development bus transactions           --
--                  necessary for a dataWidth write                       --
--                  =  Ceiling(dataWidth/32)                              --
--     readPow2    A flag indicated if readCycles is a power of 2         --
--                    - 1 if power of 2                                   --
--                    - 0 if not a power of 2                             --
--     portAddr     An 6 bit std_logic_vector that indicates the          --
--                  address of the abstracted register port               --
--                                                                        --
--  Bus Protocol                                                          --
--    The direct read transaction consists of [readCycles] different      --
--    32bit read cycles.  Reads are ordered from LSB to MSB.  The         --
--    direct read transaction continuously                                --

LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_arith.all;
USE ieee.std_logic_unsigned.all;

ENTITY DevReadBurst IS
  GENERIC (
     dataWidth : natural := 64;
     readCycles : natural := 2;   -- = CEIL(Datawidth/32)
     readPow2 : natural := 1;     -- = 1 if readCycles in a power of 2
     portAddr : natural := 0
  );
  PORT (
     -- Development Bus Signals
     resetn : in std_logic;
     address : in std_logic_vector(5 downto 0);
     datain : in std_logic_vector(31 downto 0);
     dataout : out std_logic_vector(31 downto 0);
     oe : out std_logic;
     framen : in std_logic;
     tackn : out std_logic;
     devclk : in std_logic;
     -- User interface signals
     data : in std_logic_vector(datawidth - 1 downto 0);
     dataReq : out std_logic;
     dataReady : in std_logic
  );
```

```
END;

ARCHITECTURE rtl of DevReadBurst IS
  PROCEDURE BufferData(SIGNAL data : IN std_logic_vector(datawidth - 1 downto
0);
                       SIGNAL readcount : INOUT natural;
                       SIGNAL dataout : OUT std_logic_vector(31 downto 0);
                       SIGNAL databuffer : OUT std_logic_vector( (readCycles-
1)*32 downto 0)) IS
  BEGIN
    IF readCycles > 1 THEN
      -- Need to buffer a multicycle transfer
      databuffer(datawidth - 1 - 32 downto 0) <= data(datawidth - 1 downto
32);
      dataout <= data(31 downto 0);
      readcount <= 1;
    ELSE
      dataout(datawidth - 1 downto 0) <= data;
      databuffer(0) <= '-';
      readcount <= 0;
    END IF;
  END BufferData;

  PROCEDURE SendData( SIGNAL databuffer : IN std_logic_vector( (readCycles-
1)*32 downto 0);
                      SIGNAL readcount : INOUT natural;
                      SIGNAL dataout : OUT std_logic_vector(31 downto 0)) IS
  BEGIN
    SendLoop : FOR k IN 1 TO readCycles-1 LOOP
      IF (readcount = k) THEN
        dataout <= databuffer(k*32-1 downto (k-1)*32);
      END IF;
    END LOOP SendLoop;

    IF readcount = (readCycles - 1) THEN
      -- We need to reset the counter to zero
      IF (ReadPow2 = 1) THEN
        readcount <= readcount+1;
      ELSE
        readcount <= 0;
      END IF;
    ELSE
      readcount <= readcount+1;
    END IF;

  END SendData;

  TYPE states IS (S_IDLE, S_FRAME, S_HANDSHAKEWAIT, S_TRANSFER);

  SIGNAL curr_state : states;

  SIGNAL framecycle : BOOLEAN;
  SIGNAL cyclelength : STD_LOGIC_VECTOR(15 downto 0);
  SIGNAL databuffer : std_logic_vector( (readCycles-1)*32 downto 0);
  SIGNAL readcount : natural range 0 to readCycles-1;
BEGIN
  framecycle <= (framen = '0') AND (address = portaddr);

  PROCESS (resetn,devclk)
  BEGIN
    IF (resetn = '0') THEN
      curr_state <= S_IDLE;
      cyclelength <= (others => '0');
      dataReq <= '0';
      oe <= '0';
      tackn <= '1';
    ELSIF (rising_edge(devclk)) THEN
      oe <= '0';
      tackn <= '1';
      dataout <= (others => '0');
      CASE (curr_state) IS
        WHEN S_IDLE =>
          IF framecycle THEN
            IF dataReady = '0' THEN
              curr_state <= S_FRAME;
              cyclelength <= datain(15 downto 0);
            ELSE
              cyclelength <= datain(15 downto 0) -
CONV_STD_LOGIC_VECTOR(1,16);
              dataReq <= '1';
              -- Perform transfer
              oe <= '1';
              tackn <= '0';
              BufferData(data,readcount,dataout,databuffer);
              IF readCycles > 1 THEN
                curr_state <= S_TRANSFER;
              ELSE
                curr_state <= S_HANDSHAKEWAIT;
              END IF;
            END IF;
          END IF;
        WHEN S_FRAME =>
          IF dataReady = '1' THEN
            dataReq <= '1';
            cyclelength <= cyclelength - 1;
            -- Perform transfer
            oe <= '1';
            tackn <= '0';
            BufferData(data,readcount,dataout,databuffer);
            IF readCycles > 1 THEN
              curr_state <= S_TRANSFER;
            ELSE
              curr_state <= S_HANDSHAKEWAIT;
            END IF;
          END IF;
        WHEN S_TRANSFER =>
          oe <= '1';
          tackn <= '0';
```

```
          cyclelength <= cyclelength - 1;
          SendData(databuffer,readcount,dataout);
          IF readcount = (readCycles - 1)THEN
            -- We are at the end of a transfer
            IF dataReady = '1' THEN
              curr_state <= S_HANDSHAKEWAIT;
            ELSE
              dataReq <= '0';
              IF cyclelength = CONV_STD_LOGIC_VECTOR(0,16) THEN
                curr_state <= S_IDLE;
              ELSE
                curr_state <= S_FRAME;
              END IF;
            END IF;
          END IF;
        WHEN S_HANDSHAKEWAIT =>
          dataReq <= dataReady;
          IF dataReady = '0' THEN
            IF cyclelength = CONV_STD_LOGIC_VECTOR(0,16) THEN
              curr_state <= S_IDLE;
            ELSE
              curr_state <= S_FRAME;
            END IF;
          END IF;
      END CASE;
    END IF;
  END PROCESS;

END rtl;
```

D.3 DEVWRITE.VHD

```
--  TM4: Development Bus Parameterized Bus Abstraction Module           --
--  Author: Josh Fender                                                 --
--                                                                      --
--  Command: Direct Write (no handshake)                                --
--                                                                      --
--  Description:                                                        --
--    Provides a parameterized width register on a development FPGA     --
--    that user circuits can use as a handshake free input.             --
--                                                                      --
--  User Signals:                                                       --
--     data : out std_logic_vector(dataWidth - 1 downto 0);             --
--        - parameterized width register's output                       --
--     dataNew : out std_logic;                                         --
--        - Asserted by module for one cycle at the same time new data  --
--          is available on the data output lines                       --
--                                                                      --
--  Module Parameters:                                                  --
--     dataWidth    Width, in bits, of desired register                 --
--     writeCycles  Number of 32bit development bus transactions        --
--                  necessary for a dataWidth write                     --
--                  =  Ceiling(dataWidth/32)                            --
--     writePow2    A flag indicated if writeCycles is a power of 2     --
--                    - 1 if power of 2                                 --
--                    - 0 if not a power of 2                           --
--     portAddr     An 6 bit natural that indicates the         --
--                  address of the abstracted register port             --
--                                                                      --
--  Bus Protocol                                                        --
--    The direct write transaction consists of [writeCycles] different  --
--    32bit write cycles.  Writes are ordered from LSB to MSB.  The     --
--    direct write transaction accepts any number of idle bus states    --
--    between different write cycles in a multiwrite transaction.  The  --
--    abstract port register is only updated when all [writeCycles]     --
--    write cycles have been completed.                                 --
--                                                                      --
--    A single write cycle consists of the bridge chip driving the      --
--    address and data lines as well as asserting the framen signal for --
--    one cycle.                                                        --
--                                                                      --
--  Waveform                                                            --
--     CLOCK    ____|````|____|````|____|````|____|````|               --
--     ADDRESS  -[ Address ]--1--[ Address          ]---               --
--     FRAMEn   `|_________|`````|_________________|```                --
--     DATA     -[Data LSB]------[  Data  ][  Data  ]---               --
--     TACKn    ``````````````````````````````````````                 --
--                                                                      --
--     Notes: 1 There can be any number of bus idle states between      --
--              different 32bit write cycles.  An idle state consists   --
--              of a bus cycle where FRAMEn is not asserted.            --

LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_arith.all;
USE ieee.std_logic_unsigned.all;


ENTITY DevWrite IS
  GENERIC (
     dataWidth : natural;
     writeCycles : natural;   -- = CEIL(Datawidth/32)
     writePow2 : natural;     -- = 1 if writecycles in a power of 2
     portAddr : natural );
  PORT (
     -- Development Bus Signals
     resetn : in std_logic;
     address : in std_logic_vector(5 downto 0);
     datain : in std_logic_vector(31 downto 0);
     dataout : out std_logic_vector(31 downto 0);
     oe : out std_logic;
     framen : in std_logic;
     tackn : out std_logic;
     devclk : in std_logic;
     -- User interface signals
```

```
    data : out std_logic_vector(datawidth - 1 downto 0);
    dataNew : out std_logic);
END;

ARCHITECTURE rtl of DevWrite IS
  -- Note: databuffer made one too large to handle case where no data
  --       buffer is required (ie single cycle transactions)
  SIGNAL databuffer : std_logic_vector((writecycles-1)*32 downto 0);
  SIGNAL writecount : natural range 0 to writecycles-1;
  SIGNAL writeEn : std_logic;
BEGIN
  -- Set Dev Bus outputs
  tackn <= '1';  oe <= '0';  dataout <= (others => '0');
  -- Control Logic
  writeEn <= '1' WHEN (framen = '0') AND (address = portaddr) ELSE '0';

  PROCESS (resetn,devclk,writeEn)
  BEGIN
    IF (resetn = '0') THEN
      databuffer <= (others => '0');
      writecount <= 0;   data <= (others => '0');
      dataNew <= '0';
    ELSIF (rising_edge(devclk)) THEN
      dataNew <= '0';
      IF (writeEn = '1') THEN
        -- If multicycle write then buffer the initial cycles data
        IF (writecycles > 1) THEN
          writeloop : FOR k IN 0 TO writecycles-2 LOOP
            IF (writecount = k) THEN
              databuffer((k+1)*32-1 downto k*32) <= datain;
            END IF;
          END LOOP writeloop;
        END IF;

        -- If this is the last cycle of write update data output
        -- with buffered data and current bus transaction data
        IF (writecount = writecycles-1) THEN
          dataNew <= '1';
          data(datawidth-1 downto (writecycles-1)*32) <=
                datain(datawidth - (writecycles-1)*32-1 downto 0);
          IF (writecycles > 1) THEN
            data((writecycles-1)*32-1 downto 0) <=
                databuffer((writecycles-1)*32-1 downto 0);
          END IF;

          -- If writecycles is power of 2 the counter will auto wrap
          IF (writePow2 = 1) THEN
             writecount <= writecount + 1;
          ELSE
            writecount <= 0;
          END IF;
        ELSE
          writecount <= writecount + 1;
        END IF;
      END IF;
    END IF;
  END PROCESS;

END rtl;
```

D.4 DEVWRITEACK.VHD

```
--  TM4: Development Bus Parameterized Bus Abstraction Module           --
--  Author: Josh Fender                                                 --
--                                                                      --
--  Command: Acked Write (with dataNew output)                          --
--                                                                      --
--  Description:                                                        --
--    Provides a parameterized width register on a development FPGA     --
--    that user circuits can use as a flow controlled input port        --
--                                                                      --
--  User Signals:                                                       --
--     data : out std_logic_vector(dataWidth - 1 downto 0);             --
--        - parameterized width register's output                       --
--     dataReq : in std_logic;                                          --
--        - Asserted by user when they want new data                    --
--     dataNew : out std_logic;                                         --
--        - Asserted by module when data is valid and is held asserted  --
--          until the user deasserts dataReq                            --
--                                                                      --
--  Module Parameters:                                                  --
--     dataWidth    Width, in bits, of desired register                 --
--     writeCycles  Number of 32bit development bus transactions        --
--                  necessary for a dataWidth write                     --
--                  =  Ceiling(dataWidth/32)                            --
--     writePow2    Unused legacy parameter                             --
--     portAddr     An 6 bit std_logic_vector that indicates the        --
--                  address of the abstracted register port             --
--                                                                      --
--  Bus Protocol                                                        --
--    The acked write transaction consists of [writeCycles] different   --
--    32bit write cycles followed by an acknowledgement cycle.  Writes  --
--    are ordered from LSB to MSB.  The acked write transaction accepts --
--    any number of idle bus states between different write cycles in   --
--    a multiwrite transaction.  The abstract port register is only     --
--    updated when all [writeCycles] write cycles have been completed   --
--    and the user circuit has asserted dataReq.                        --
--                                                                      --
--    A single write cycle consists of the bridge chip driving the      --
--    address and data lines as well as asserting the framen signal for --
--    one cycle.                                                        --
--                                                                      --
--    An acknowledgement cycle must follow, without any idle cycles,    --
--    immediately after a write cycle.  The cycle consists of the master --
--    device holding the target address, write data and continuing to   --
```

```
--    assert frame until it detects a target acknowledgement.  The
--    master will then deassert frame for one cycle before continuing
--    with further transactions.
--
--  Waveform
--     CLOCK    ____|````|____|````|____|````|____|````|____|````|____|~~~~|
--     ADDRESS  -[ Address ]--1--[ Address                ]--------
--     FRAMEn   `|_________|`````|_____________________|````````
--     DATA     -[Data LSB]------[         Data MSB       ]--------
--     TACKn    ````````````````````````````|________|````````````
--
--     Notes: 1 There can be any number of bus idle states between
--              different 32bit write cycles.  An idle state consists
--              of a bus cycle where FRAMEn is not asserted.
--
--  User Circuit Handshake
--     dataReq [User Driven]  _______`````````````````````_________
--     dataNew [Port Driven]  ___________________`````````````_____
--     dataOut [Port Driven]  -----------------<Data Valid>--------
--
--     - User circuit asserts dataReq
--     - When data is ready the port asserts dataNew and outputs the
--       newly received data
--     - Once the user circuit is finished with the data it deasserts
--       dataReq.  From this time dataOut is considered no longer valid
--     - Upon seeing that the user circuit has deasserted dataReq the
--       port will deassert dataNew

LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_arith.all;
USE ieee.std_logic_unsigned.all;

ENTITY DevWriteAck IS
  GENERIC (
     dataWidth : natural := 53;
     writeCycles : natural := 2;   -- = CEIL(Datawidth/32)
     writePow2 : natural := 1;     -- = 1 if writecycles in a power of 2
     portAddr : natural := 0);
  PORT (
     -- Development Bus Signals
     resetn : in std_logic;
     address : in std_logic_vector(5 downto 0);
     datain : in std_logic_vector(31 downto 0);
     dataout : out std_logic_vector(31 downto 0);
     oe : out std_logic;
     framen : in std_logic;
     tackn : out std_logic;
     devclk : in std_logic;
     -- User interface signals
     data : out std_logic_vector(datawidth - 1 downto 0);
     dataReq : in std_logic;
     dataNew : out std_logic);
END;
--
--  ARCHITECTURE rtl of DevWriteAck IS
--    SIGNAL databuffer : std_logic_vector(writecycles*32-1 downto 0);
--    SIGNAL writecount : natural range 0 to writecycles;
--    SIGNAL writeEn, pause : boolean;
--    SIGNAL dataReady : std_logic;
--  BEGIN
--    data <= databuffer(datawidth - 1 downto 0);
--    dataNew <= dataReady;
--
--    -- Set Dev Bus outputs
--    dataout <= (others => '0');
--    oe <= '0';
--
--    writeEn <= (framen = '0') AND (address = portaddr);
--    tackn <= '0' WHEN (writeCount = writeCycles) AND (dataReq = '0')
--                  AND (dataReady = '1') ELSE '1';
--
--    PROCESS (resetn, devclk)
--    BEGIN
--      IF (resetn = '0') THEN
--        databuffer <= (others => '0');
--        writeCount <= 0;
--        dataReady <= '0';
--        pause <= false;
--      ELSIF rising_edge(devclk) THEN
--        -- Pause until the framen is deasserted
--        IF pause THEN
--          pause <= (framen = '0');
--        -- If we have data ready then handle the handshake
--        ELSIF (writeCount = writeCycles) THEN
--          dataReady <= dataReq;
--          IF (dataReq = '0') AND (dataReady = '1') THEN
--            writeCount <= 0;
--            pause <= true;
--          END IF;
--        -- If the devbus is handling a write then buffer to the shift register
--        ELSIF writeEn THEN
--          writeCount <= writecount + 1;
--          -- Load 32bit parallel shift register
--          databuffer(writecycles*32-1 downto (writecycles -1)*32) <= dataIn;
--          IF (writecycles > 1) THEN
--            writeloop : FOR k IN 1 TO writecycles-1 LOOP
--              databuffer((k*32)-1 downto (k-1)*32) <=
--                 databuffer((k+1)*32-1 downto k*32);
--            END LOOP writeloop;
--          END IF;
--        END IF;
--      END IF;
--    END PROCESS;
--
--  END rtl;
```

E LINUX DEVICE DRIVER

E.1 TM4DRIVER.C

```
#define MODULE
#define __KERNEL__

#include <linux/config.h>
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/pci.h>
#include <linux/sched.h>
#include <asm/uaccess.h>


#include "ioctlcmd.h"
#include "clockparam.h"

#ifndef CONFIG_PCI
#  error "This driver needs PCI support to be available"
#endif

#define TM4_VENDOR 0x1172
#define TM4_DEVICE 0x0004

#define TM4WREG_NCONFIG  0*4
#define TM4WREG_MODE     1*4
#define TM4WREG_NIBRESET 2*4
#define TM4WREG_DMAADDR  0xfa*4
#define TM4WREG_DMACTRL  0xfb*4
#define TM4WREG_JTAG     0xfc*4
#define TM4WREG_TEMPMON  0xfd*4
#define TM4WREG_GLBCLK   0xfe*4
#define TM4WREG_NIBCLK   0xff*4

#define TM4RREG_NSTATUS 0*4
#define TM4RREG_NIBCLK  1*4
#define TM4RREG_GLBCLK  2*4
#define TM4RREG_TEMPMON 3*4
#define TM4RREG_WFIFO   5*4
#define TM4RREG_ERROR   6*4
#define TM4RREG_JTAG    7*4
#define TM4RREG_RFIFO   9*4
#define TM4RREG_LIRQ    14*4

#define TEMPMON_DATAMASK 0xff
#define TEMPMON_RDACK    0x100
#define TEMPMON_WRACK    0x200
#define TEMPMON_CLRACK   0x400

#define TM4_MODEREG_DEVBUS_ENABLE 0x0001
#define TM4_MODEREG_DEVCFG_ENABLE 0x0002
#define TM4_MODEREG_VIDEO_OUT     0x0010
#define TM4_MODEREG_VIDEO_IN_A    0x0004
#define TM4_MODEREG_VIDEO_IN_B    0x0008
#define TM4_MODEREG_JTAG          0x0020
#define TM4_MODEREG_ALERT_OVER    0x0040
#define TM4_MODEREG_FORCE_RESETN  0x80000000

#define JTAG_BUSY 0x80000000

#define PLLCFG_N  0
#define PLLCFG_M  1
#define PLLCFG_G0 4
#define PLLCFG_G1 5
#define PLLCFG_G2 6
#define PLLCFG_G3 7
#define PLLCFG_L0 8
#define PLLCFG_L1 9
#define PLLCFG_E0 0xc
#define PLLCFG_E1 0xd
#define PLLCFG_E2 0xe
#define PLLCFG_E3 0xf

#define PLLCFG_NOMINAL   0
#define PLLCFG_HIGHCOUNT 0
#define PLLCFG_LOWCOUNT  1
#define PLLCFG_COUNTERBYPASS 4
```

```
/* Stratix JTAG Commands */
#define JTAG_EXTEST         0x000
#define JTAG_PULSE_NCONFIG  0x001
#define JTAG_PROGRAM        0x002
#define JTAG_STARTUP        0x003
#define JTAG_CHECK_STATUS   0x004
#define JTAG_SAMPLE         0x005
#define JTAG_IDCODE         0x006
#define JTAG_USERCODE       0x007
#define JTAG_CLAMP          0x00A
#define JTAG_HIGHZ          0x00B
#define JTAG_CONFIG_IO      0x00D
#define JTAG_BYPASS         0x3ff

int tm4_open (struct inode *inode, struct file *filp);
int tm4_release( struct inode *inode, struct file *filp);
int tm4_ioctl( struct inode *inode, struct file *filp,
       unsigned int cmd, unsigned long arg);
ssize_t tm4_read (struct file *filp, char *buff, size_t count,
    loff_t *offp);

ssize_t tm4_write (struct file *filp, const char *buff,
    size_t count, loff_t *offp);

struct file_operations tm4_fops = {
  open    : tm4_open,
  release : tm4_release,
  write   : tm4_write,
  read    : tm4_read,
  ioctl   : tm4_ioctl,
  owner   : THIS_MODULE,
};

int tm4_debug=0;
int major;
u32 tm4_modereg=0;
int devclock=66;
int glbclock=66;

struct pci_dev *tm4pci;
dma_addr_t dma_bus_addr;
void      *dma_virtual_addr;

/*************************************************************/
/* TM-4 PCI Memory space interface routines                  */
/*************************************************************/

void *bar0virtual, *bar1virtual;

inline void bar0writel(u32 value, u32 offset) {
  writel(value,bar0virtual+offset);
}

inline u32 bar0readl(u32 offset) {
  return readl(bar0virtual+offset);
}

inline u32 bar1readl() {
  return readl(bar1virtual);
}

inline void bar1writel(u32 value) {
  writel(value,bar1virtual);
}

/*************************************************************/


void enable_devcfg() {
  tm4_modereg |= TM4_MODEREG_DEVCFG_ENABLE;
  tm4_modereg |= TM4_MODEREG_FORCE_RESETN;
  tm4_modereg &= ~TM4_MODEREG_DEVBUS_ENABLE;

  bar0writel(tm4_modereg,TM4WREG_MODE);
  wmb();
}

int enable_devbus() {
  int i;

  for (i=0; i < 1000000; i++) {
    if (bar0readl(TM4RREG_WFIFO) == 0) break;
  }
  if (bar0readl(TM4RREG_WFIFO) != 0) return -1;

  tm4_modereg |= TM4_MODEREG_DEVBUS_ENABLE;
  tm4_modereg |= TM4_MODEREG_FORCE_RESETN;
  tm4_modereg &= ~TM4_MODEREG_DEVCFG_ENABLE;

  bar0writel(tm4_modereg,TM4WREG_MODE);
  bar0writel(0, TM4WREG_NIBRESET);
  wmb();
  bar0writel(0xf, TM4WREG_NIBRESET);
  return 0;
}

void tm4_alertoverride() {
  tm4_modereg |= TM4_MODEREG_ALERT_OVER;
  bar0writel(tm4_modereg,TM4WREG_MODE);
  wmb();
}

void set_mode_reg(int bit) {
  tm4_modereg |= bit;
  bar0writel(tm4_modereg,TM4WREG_MODE);
  wmb();
}
```

```
void unset_mode_reg(int bit) {
  tm4_modereg &= ~bit;
  bar0writel(tm4_modereg,TM4WREG_MODE);
  wmb();
}

int set_devclk(int rate) {
  if ((rate < 1) || (rate > 100)) return -1;

  devclock = rate;

  if (clkparam[rate-1][0] == 1)
    write_nib_pll_reconfig_word(PLLCFG_N, PLLCFG_COUNTERBYPASS, 1);
  else {
    write_nib_pll_reconfig_word(PLLCFG_N, PLLCFG_NOMINAL, clkparam[rate-
1][0]);
    write_nib_pll_reconfig_word(PLLCFG_N, PLLCFG_COUNTERBYPASS, 0);
  }

  if (clkparam[rate-1][1] == 1)
    write_nib_pll_reconfig_word(PLLCFG_M, PLLCFG_COUNTERBYPASS, 1);
  else {
    write_nib_pll_reconfig_word(PLLCFG_M, PLLCFG_NOMINAL, clkparam[rate-
1][1]);
    write_nib_pll_reconfig_word(PLLCFG_M, PLLCFG_COUNTERBYPASS, 0);
  }

  write_nib_pll_reconfig_word(PLLCFG_E0, PLLCFG_HIGHCOUNT, clkparam[rate-
1][2]/2);
  write_nib_pll_reconfig_word(PLLCFG_E0, PLLCFG_LOWCOUNT, clkparam[rate-
1][2]/2);
  write_nib_pll_reconfig_word(PLLCFG_G0, PLLCFG_HIGHCOUNT, clkparam[rate-
1][2]/2);
  write_nib_pll_reconfig_word(PLLCFG_G0, PLLCFG_LOWCOUNT, clkparam[rate-
1][2]/2);

  return initiate_nib_pll_reconfig();
}

int make_temp_cmd(int command, int data, int rd_req, int wr_req,
                  int clr_alert, int chip_sel) {
  return (command & 0xff) |
         ((data & 0xff) << 8) |
         ((rd_req & 1) << 16) |
         ((wr_req & 1) << 17) |
         ((clr_alert & 1) << 18) |
         ((chip_sel & 1) << 31);
}

int write_temp_smbus(int arg) {
  int command, data, chip_sel;
  int i,value;

  command = (arg & 0xff);
  data = (arg >> 8) & 0xff;
  chip_sel = (arg >> 16) & 0x01;
  bar0writel( make_temp_cmd( command, data,0,1,0, chip_sel),
TM4WREG_TEMPMON );
  wmb();

  for (i = 0; i < 1000000; i++) {
    value = bar0readl(TM4RREG_TEMPMON);
    if ((value & TEMPMON_WRACK) == TEMPMON_WRACK) break;
  }

  if ((value & TEMPMON_WRACK) == 0) return -1;

  bar0writel(0, TM4WREG_TEMPMON);
  for (i = 0; i < 1000000; i++) {
    if ((bar0readl(TM4RREG_TEMPMON) & TEMPMON_WRACK) == 0)
      return 0;
  }
  return -1;
}

int read_temp_smbus(int *arg) {
  int i, value;
  int argin, command, chip_sel;

  // Read arguments from user space
  get_user(argin,arg);
  command = argin & 0xff;
  chip_sel = (argin >> 8) & 0x01;

  bar0writel( make_temp_cmd(command, 0, 1,0,0,chip_sel), TM4WREG_TEMPMON);
  wmb();

  for (i = 0; i < 1000000; i++) {
    value = bar0readl(TM4RREG_TEMPMON);
    if ((value & TEMPMON_RDACK) == TEMPMON_RDACK) break;
  }

  if ((value & TEMPMON_RDACK) == 0) {
    printk( KERN_INFO "TM-4: Error: SMBUS Interface did not assert
Ack\n");
    return -1;
  }

  bar0writel(0, TM4WREG_TEMPMON);
  for (i = 0; i < 1000000; i++) {
    value=bar0readl(TM4RREG_TEMPMON);
    if ((value & TEMPMON_RDACK) == 0) break;
  }

  if ((value & TEMPMON_RDACK) == TEMPMON_RDACK) {
    printk( KERN_INFO "TM-4 Error: SMBUS Interface did not deassert
Ack\n");
    return -1;
```

```
  }

  return put_user(value & TEMPMON_DATAMASK, arg);
}

void tm4_reset() {
  // Unconfigure the chips
  bar0writel(0x00000000,TM4WREG_NCONFIG);
  wmb();

  // Reset the the housekeeping FIFOs and devbus interfaces
  unset_mode_reg(TM4_MODEREG_FORCE_RESETN);
  set_mode_reg(TM4_MODEREG_FORCE_RESETN);

  // Setup the DMA address
  bar0writel(dma_bus_addr,TM4WREG_DMAADDR);
}

int write_jtag(int arg) {
  int i, result;

  bar0writel(arg, TM4WREG_JTAG);
  wmb();
  for (i = 0; i < 1000000; i++) {
    result = bar0readl(TM4RREG_JTAG);
    if ((result & JTAG_BUSY) == 0) break;
  }
  return result;
}

int jtag_issue(int tms, int tdo) {
  return write_jtag( (tms << 1) | tdo);
}

int get_config_status() {
  return bar0readl(TM4RREG_NSTATUS);
}

int read_reg(int *arg) {
  (*arg) = bar0readl((*arg)*4);
  return 0;
}

int get_clock_rate(int arg) {
  switch (arg) {
    case 0: return devclock;
    case 1: return glbclock;
    default: return -1;
  }
}

// Must issue start read and end read commands to issue correct TAP state
int jtag_read32() {
  int i, result;

  result = 0;
  for (i = 0; i < 32; i++) {
    result = result | (jtag_issue(0,0) << i);
  }

  return result;
}

int tm4_ioctl( struct inode *inode, struct file *filp,
       unsigned int cmd, unsigned long arg) {
  int num = MINOR(inode->i_rdev);

  if (num != 1) return -1;

  if (tm4_debug > 0)
    printk(KERN_INFO "IOCTL cmd %08x\n",cmd);


  switch (cmd) {
    case IOCTL_ENABLE_VIDEO_OUT:  set_mode_reg(TM4_MODEREG_VIDEO_OUT);
break;
    case IOCTL_DISABLE_VIDEO_OUT: unset_mode_reg(TM4_MODEREG_VIDEO_OUT);
break;
    case IOCTL_ENABLE_VIDEO_IN_A: set_mode_reg(TM4_MODEREG_VIDEO_IN_A);
break;
    case IOCTL_DISABLE_VIDEO_IN_A: unset_mode_reg(TM4_MODEREG_VIDEO_IN_A);
break;
    case IOCTL_ENABLE_VIDEO_IN_B: set_mode_reg(TM4_MODEREG_VIDEO_IN_B);
break;
    case IOCTL_DISABLE_VIDEO_IN_B: unset_mode_reg(TM4_MODEREG_VIDEO_IN_B);
break;
    case IOCTL_ENABLE_JTAG: set_mode_reg(TM4_MODEREG_JTAG); break;
    case IOCTL_DISABLE_JTAG: unset_mode_reg(TM4_MODEREG_JTAG); break;
    case IOCTL_SET_DEVCLK: return set_devclk(arg);
    case IOCTL_READ_TEMP_SMBUS: return read_temp_smbus((int *)arg);
    case IOCTL_WRITE_TEMP_SMBUS: return write_temp_smbus(arg);
    case IOCTL_RESET: tm4_reset(); break;
    case IOCTL_JTAG: return write_jtag(arg);
    case IOCTL_GET_CONFIG_STATUS: return get_config_status();
    case IOCTL_NIB_RESET: bar0writel(arg, TM4WREG_NIBRESET); break;
    case IOCTL_GET_WRITE_LEVEL: return bar0readl(TM4RREG_WFIFO);
    case IOCTL_GET_ERROR: return bar0readl(TM4RREG_ERROR);
    case IOCTL_SET_NCONFIG : bar0writel(arg,TM4WREG_NCONFIG); break;
    case IOCTL_REGISTER_READ: return read_reg((int *)arg);
    case IOCTL_GET_CLOCKRATE: return get_clock_rate(arg);
    case IOCTL_SET_DEBUG_LEVEL: tm4_debug = arg; break;
    case IOCTL_ALERT_OVERRIDE: tm4_alertoverride(); break;
    default: return -1;
  }
  return 0;
}

u32 make_pll_reconfig_word(u32 reconfig, u32 read_en, u32 write_en,
```

```
     u32 counter_type, u32 data_in, u32 counter_param) {
  return (reconfig & 0x1) |
         ((read_en & 0x1) << 1) |
         ((write_en & 0x1) << 2) |
         ((counter_type & 0xf) << 4) |
         ((data_in & 0x1ff) << 8) |
         ((counter_param & 0x7) << 24);
}

int write_nib_pll_reconfig_word(u32 counter_type, u32 counter_param, u32
data) {
  int i, result;

  bar0writel( make_pll_reconfig_word(0,0,1,counter_type, data,
counter_param),
    TM4WREG_NIBCLK);
  wmb();

  for (i = 0; i < 100000; i++) {
    result = bar0readl(TM4RREG_NIBCLK);
    rmb();
    if ((result & 0x80000000) == 0) return 0;
  }
  return -1;
}

int read_nib_pll_reconfig_word(u32 counter_type, u32 counter_param) {
  int i, result;

  bar0writel( make_pll_reconfig_word(0,1,0,counter_type,0, counter_param),
    TM4WREG_NIBCLK);
  wmb();

  for (i = 0; i < 100000; i++) {
    result = bar0readl(TM4RREG_NIBCLK);
    rmb();
    if ((result & 0x80000000) == 0) break;
  }

  return result;
}

int initiate_nib_pll_reconfig() {
  int i,result;

  bar0writel(1,TM4WREG_NIBCLK);

  for (i = 0; i < 1000000; i++) {
    result = bar0readl(TM4RREG_NIBCLK);
    rmb();
    if ((result & 0x80000000) == 0) return 0;
  }
  return -1;
}
```

```
void dump_nib_pll_configuration() {
  int i, value;

  for (i = 0; i <= 0xf; i++) {
    value = read_nib_pll_reconfig_word(i,0);
    if (value < 0) {
      printk(KERN_INFO "TM-4: Timeout reading register 0x%x\n",i);
    } else {
      printk(KERN_INFO "TM-4: Nib PLL Cfg Reg 0x%x Value 0x%x\n",i,value);
    }
  }

}

int init_config() {
  int i;


  // We need to assert nCONFIG until nSTATUS is asserted
  bar0writel(0x00000000,TM4WREG_NCONFIG);
  wmb();

  // Test nSTATUS
  for (i=0; (i < 1000000) && (bar0readl(TM4RREG_NSTATUS) != 0); i++);

  if (bar0readl(TM4RREG_NSTATUS) != 0) {
    printk(KERN_INFO "TM-4 ERROR: Device did not assert nStatus\n");
    return -1;
  }


  // Now deassert nCONFIG and wait for nSTATUS to release
  bar0writel(0x0000000f,TM4WREG_NCONFIG);
  wmb();

  // Test nSTATUS for release
  for (i=0; (i < 1000000) && (bar0readl(TM4RREG_NSTATUS) != 0xaa); i++);

  if (bar0readl(TM4RREG_NSTATUS) != 0xaa) {
    printk(KERN_INFO "TM-4 ERROR: Device did not release nStatus
0x%08x\n",
       bar0readl(TM4RREG_NSTATUS));
    return -1;
  }


  // Now set the write FIFOs to feed the dev FPGA configuration circuit
  enable_devcfg();
  wmb();

  return 0;
}
```

```
int finish_config() {
  int i;

  for(i=0; i < 136; i++) {
    bar1writel(0);
    wmb();
  }

  return enable_devbus();
}

/* Insure that TAP controller is in state RUN/IDLE */
void jtag_reset() {
  jtag_issue(1,0);
  jtag_issue(1,0);
  jtag_issue(1,0);
  jtag_issue(1,0);
  jtag_issue(1,0);
  jtag_issue(0,0);
}
void jtag_setIR(int inst0, int inst1, int inst2, int inst3) {
  int i,bit;

  jtag_issue(1,0);  // State: SELECT_DR_SCAN
  jtag_issue(1,0);  // State: Select_IR_scan
  jtag_issue(0,0);  // State: capture_IR

  jtag_issue(0,0);  // State: shift_IR

  // Issue command 3
  bit = inst3;
  for (i = 0; i < 10; i++) {
    jtag_issue(0,bit & 1);
    bit = bit >> 1;
  }

  // Issue command 2
  bit = inst2;
  for (i = 0; i < 10; i++) {
    jtag_issue(0,bit & 1);
    bit = bit >> 1;
  }

  // Issue command 1
  bit = inst1;
  for (i = 0; i < 10; i++) {
    jtag_issue(0,bit & 1);
    bit = bit >> 1;
  }

  // Issue command 0
  bit = inst0;
  for (i = 0; i < 9; i++) {
    jtag_issue(0,bit & 1);
    bit = bit >> 1;
  }

  jtag_issue(1,bit & 1);  // State: Exit1_IR
  jtag_issue(1,0);  // State: Update_IR
  jtag_issue(0,0);  // State: Run_test/Idle
}

void jtag_startread() {
  jtag_issue(1,0);  // State: Select_DR_Scan
  jtag_issue(0,0);  // State: Capture DR
}

void jtag_endread() {
  jtag_issue(1,0);  // State: Exit1_Dr
  jtag_issue(1,0);  // State: Update_Dr
  jtag_issue(0,0);  // State: Run_Test/idle
}

ssize_t tm4_jtag_read (struct file *filp, char *buff, size_t count,
    loff_t *offp) {
  int I, value;

  for (i=0; i < count; i+=4) {
    value = jtag_read32();
    __put_user(value,(u32 *)(buff+i));
  }

  return count;
}

int init_jtag(struct file * filp) {
  // Point read procedure to correct function
  filp->f_op->read = &tm4_jtag_read;
  // Enable JTAG
  set_mode_reg(TM4_MODEREG_JTAG);

  // Reset JTAG controller
  jtag_reset();

  // Set Instruction
  jtag_setIR(JTAG_SAMPLE, JTAG_SAMPLE,
             JTAG_SAMPLE, JTAG_SAMPLE);


  jtag_startread();
  return 0;
}

void finish_jtag() {
  jtag_reset();
  unset_mode_reg(TM4_MODEREG_JTAG);
}
```

```
int tm4_open (struct inode *inode, struct file * filp) {
  int num = MINOR(inode->i_rdev);

  filp->f_op->read = &tm4_read;

  switch (num) {
    case 0: return init_config(); break;
    case 1: break;
    case 2: return init_jtag(filp); break;
    default: return -1;
  }
  return 0;
}

void start_dma_write_to_tm4(u32 count) {
  if (tm4_debug > 0)
    printk(KERN_INFO "DMA Write Setup: 0x%08x addr, 0x%08x command\n",
       dma_bus_addr, (count >> 2) & 0xfffe );

  bar0writel((count >> 2) & 0xFFFE, TM4WREG_DMACTRL);
  wmb();
}

void start_dma_read_from_tm4(u32 count) {
  u32 command = ((count >> 2) & 0xFFFE) | 0x80000000;

  if (tm4_debug > 0)
    printk(KERN_INFO "DMA Read Setup: 0x%08x addr, 0x%08x command\n",
       dma_bus_addr, command );

  bar0writel( command, TM4WREG_DMACTRL);
  wmb();
}

wait_queue_head_t tm4_wait_queue;

int interrupt_arrived;

void interrupt_handler(int irq, void *dev_id, struct pt_regs *regs) {
  if (tm4_debug > 0)
    printk(KERN_INFO "Interrupt!\n");

  if (bar0readl(TM4RREG_LIRQ) == 1) return;

  // Clear the interrupt from the TM-4
  bar0writel(dma_bus_addr,TM4WREG_DMAADDR);
  wmb();
  while (bar0readl(TM4RREG_LIRQ) == 0);

  interrupt_arrived = 1;

  // Wake up the blocked task
  wake_up(&tm4_wait_queue);
}
```

```
ssize_t tm4_read (struct file *filp, char *buff, size_t count,
    loff_t *offp) {


  u32 i;
  u32 value;

  // Clear the low bits to insure we only write multiples of 32bits
  count = count & 0xfffffffc;

  // Don't bother with DMA for small transfers
  if (count <= 0xf) {
    for (i=0; i < count; i+=4) {
      value = bar1readl();
      __put_user(value,(u32 *)(buff+i));
    }
  } else {

    if (count > 4) {
      if (tm4_debug > 0)
        printk(KERN_INFO "Adding to wait queue(read)\n");

      interrupt_arrived = 0;
      start_dma_read_from_tm4(count);

      wait_event_interruptible(tm4_wait_queue,(interrupt_arrived==1));

      if (tm4_debug > 0)
        printk(KERN_INFO "We have been awakened\n");
    }

    if ((count & 0x4) == 0x4) {
      if (tm4_debug > 0)
        printk(KERN_INFO "Performing Lone 32bit read\n");

      value = bar1readl();
      __put_user(value,(u32 *)(buff+count-4));
      copy_to_user( (u32 *)buff,dma_virtual_addr, count-4);
    } else {
      copy_to_user( (u32 *)buff,dma_virtual_addr, count);
    }

  }
  return count;

}

int writecounter = 0;

ssize_t tm4_write (struct file *filp, const char *buff,
    size_t count, loff_t *offp) {

  u32 value;
```

```
  if (tm4_debug > 0)
    printk(KERN_INFO "Devbus Write: 0x%08x bytes\n", count);
  if (tm4_debug > 1) {
    get_user(value,(u32 *)buff);
    printk(KERN_INFO "  - First Value 0x%08x\n",value);
  }

  // Clear the low bits to insure we only write multiples of 32bits
  count = count & 0xfffffffc;

  // Don't bother with DMA for small transfers
  if (count <= 0xf) {
    if (count > 0) copy_from_user(bar1virtual, buff, count);
  } else {
    if ((count & 0x4) == 0x4) {
      if (tm4_debug > 0)
        printk(KERN_INFO "Performing Lone 32bit write\n");

      get_user(value, (u32 *)buff);
      bar1writel(value);
      copy_from_user(dma_virtual_addr, (buff+4), count-4);
    } else {
      copy_from_user(dma_virtual_addr, buff, count);
    }

    if (count > 4) {
      if (tm4_debug > 0)
        printk(KERN_INFO "Adding to wait queue %x\n",writecounter++);

      interrupt_arrived = 0;
      start_dma_write_to_tm4(count);

      wait_event_interruptible(tm4_wait_queue,(interrupt_arrived==1));

      if (tm4_debug > 0)
        printk(KERN_INFO "We have been awakened\n");
    }
  }
  return count;
}

int tm4_release( struct inode *inode, struct file *filp) {
  int num = MINOR(inode->i_rdev);

  // Restore the standard write just in case we were in JTAG mode
  filp->f_op->read = &tm4_read;

  switch (num) {
    case 0: return finish_config(); break;
    case 1: break;
    case 2: finish_jtag(); break;
    default: return -1;
  }
  return 0;
}

u8 tm4irq;

int init_module(void) {
  u32 bar0;
  u32 bar1;

  // Register the device driver
  major = register_chrdev(0, "tm4", &tm4_fops);
  if (major < 0) return -1;

  printk(KERN_INFO "TM-4: Device Driver Init\n");
  if (!pci_present()) return -1;

  tm4pci = pci_find_device(TM4_VENDOR, TM4_DEVICE, NULL);
  if (tm4pci != NULL)
    printk(KERN_INFO "TM-4: Found TM4 PCI controller\n");
  else
    return -1;

  pci_enable_device(tm4pci);

  bar0 = pci_resource_start(tm4pci,0);
  bar1 = pci_resource_start(tm4pci,1);

  bar0virtual = ioremap_nocache(bar0,1024);
  bar1virtual = ioremap_nocache(bar1,64*1024);

  if ((bar0virtual == 0) || (bar1virtual == 0)) {
    printk(KERN_INFO "TM-4 ERROR: Unable to allocate BAR memory space\n");
    return -1;
  }

  if(pci_read_config_byte(tm4pci, PCI_INTERRUPT_LINE, &tm4irq)) {
    printk(KERN_INFO "TM-4 ERROR: Unable to determine IRQ number\n");
    return -1;
  }

  printk(KERN_INFO "TM-4: Detected IRQ # %x\n", tm4irq);

  if (request_irq(tm4irq, interrupt_handler, SA_SHIRQ,
"tm4",&dma_virtual_addr)) {
    printk(KERN_INFO "TM-4 ERROR: Unable to install interrupt handler\n");
    return -1;
  }

  printk(KERN_INFO "TM-4: Interrupt Handler Installed\n");

  // Setup the wait queue so we can sleep during blocks
  init_waitqueue_head(&tm4_wait_queue);

  // Allocate a 64K PCI DMA Buffer
```

```
  dma_virtual_addr = pci_alloc_consistent(tm4pci, 0x10000, &dma_bus_addr);

  printk(KERN_INFO "TM-4: DMA buffer allocated\n");

  // Reset TM-4
  tm4_reset();

  printk(KERN_INFO "TM-4: Driver load complete\n");

  return 0;
}

void cleanup_module(void)  {

  free_irq(tm4irq, &dma_virtual_addr);
  pci_free_consistent(tm4pci, 0x10000, dma_virtual_addr, dma_bus_addr);
  iounmap(bar1virtual);
  iounmap(bar0virtual);
  unregister_chrdev(major,"tm4");
  printk(KERN_INFO "TM-4: Driver removed\n");
}

MODULE_AUTHOR("Josh Fender");
MODULE_DESCRIPTION("Transmogrifier-4 Interface Driver");
MODULE_LICENSE("Not Free");
```

E.2 IOCTLCMD.H

```
#define IOCTL_MAGIC '4'
#define IOCTL_ENABLE_VIDEO_OUT   _IO(IOCTL_MAGIC, 0)
#define IOCTL_DISABLE_VIDEO_OUT  _IO(IOCTL_MAGIC, 1)
#define IOCTL_ENABLE_VIDEO_IN_A  _IO(IOCTL_MAGIC, 2)
#define IOCTL_DISABLE_VIDEO_IN_A _IO(IOCTL_MAGIC, 3)
#define IOCTL_ENABLE_VIDEO_IN_B  _IO(IOCTL_MAGIC, 4)
#define IOCTL_DISABLE_VIDEO_IN_B _IO(IOCTL_MAGIC, 5)
#define IOCTL_SET_DEVCLK         _IO(IOCTL_MAGIC, 6)
#define IOCTL_NIB_RESET          _IO(IOCTL_MAGIC, 7)
#define IOCTL_RESET              _IO(IOCTL_MAGIC, 8)
#define IOCTL_READ_TEMP_SMBUS    _IOWR(IOCTL_MAGIC, 9,4)
#define IOCTL_WRITE_TEMP_SMBUS   _IO(IOCTL_MAGIC, 10)
#define IOCTL_ENABLE_JTAG        _IO(IOCTL_MAGIC, 11)
#define IOCTL_DISABLE_JTAG       _IO(IOCTL_MAGIC, 12)
#define IOCTL_JTAG               _IO(IOCTL_MAGIC, 13)
#define IOCTL_GET_CONFIG_STATUS  _IO(IOCTL_MAGIC, 14)
#define IOCTL_GET_WRITE_LEVEL    _IO(IOCTL_MAGIC, 15)
#define IOCTL_GET_ERROR          _IO(IOCTL_MAGIC, 16)
#define IOCTL_SET_NCONFIG        _IO(IOCTL_MAGIC, 17)
#define IOCTL_REGISTER_READ      _IOWR(IOCTL_MAGIC, 18,4)
#define IOCTL_GET_CLOCKRATE      _IO(IOCTL_MAGIC, 19)
#define IOCTL_ALERT_OVERRIDE     _IO(IOCTL_MAGIC, 20)
#define IOCTL_SET_DEBUG_LEVEL    _IO(IOCTL_MAGIC, 21)
#define MAX_IOCTL_COMMAND 21

#ifdef TM4SET

char *ioctl_text[] = {
"enable_video_out", "disable_video_out", "enable_video_in_a",
"disable_video_in_a",
"enable_video_in_b", "disable_video_in_b", "set_devclk", "nib_reset",
"reset",
"read_temp_smbus", "write_temp_smbus", "enable_jtag", "disable_jtag",
"write_jtag",
"get_config_status", "get_write_fifo_level", "get_error", "set_nconfig",
"reg_read",
"get_clockrate", "alert_override", "debug_level"};

int ioctl_mapping[] = {
  IOCTL_ENABLE_VIDEO_OUT,  IOCTL_DISABLE_VIDEO_OUT,
IOCTL_ENABLE_VIDEO_IN_A,
  IOCTL_DISABLE_VIDEO_IN_A,  IOCTL_ENABLE_VIDEO_IN_B,
IOCTL_DISABLE_VIDEO_IN_B,
  IOCTL_SET_DEVCLK,  IOCTL_NIB_RESET,  IOCTL_RESET,
IOCTL_READ_TEMP_SMBUS,
  IOCTL_WRITE_TEMP_SMBUS,  IOCTL_ENABLE_JTAG,  IOCTL_DISABLE_JTAG,
IOCTL_JTAG,
  IOCTL_GET_CONFIG_STATUS,  IOCTL_GET_WRITE_LEVEL,  IOCTL_GET_ERROR,
  IOCTL_SET_NCONFIG,  IOCTL_REGISTER_READ,  IOCTL_GET_CLOCKRATE,
  IOCTL_ALERT_OVERRIDE,  IOCTL_SET_DEBUG_LEVEL};

#endif
```

F DC-DC CONVERTER SPICE MODEL

```
TM4: Power Subsystem: 1.5v

.options NOMOD NOPAGE POST=1 PARHIER=local BRIEF
.options METHOD=GEAR

.include "irf6602.inc"
.include "irf6601.inc"
.include "MBR0520L.mod"
.include "MBRS340T3.mod"

.options BRIEF=0

*** NOTES:
* - Assumes Ideal PCB trace (0uH, 0uF, 0uOhm)
* - Boost Diode and capacitor replaced with Voltage Source


* V1.5v Holdup Capacitors
Cb1 b1 0 470u
Rb1 b1 vload 8m
Cb2 b2 0 470u
Rb2 b2 vload 8m
Cb3 b3 0 470u
Rb3 b3 vload 8m
Cb4 b4 0 470u
Rb4 b4 vload 8m
Cb5 b5 0 470u
Rb5 b5 vload 8m
Cb6 b6 0 470u
Rb6 b6 vload 8m
Cb7 b7 0 470u
Rb7 b7 vload 8m
Cb8 b8 0 470u
Rb8 b8 vload 8m

* Fixed Load 50Amps
*Rload vload 0 0.03

* Fixed Load 2Amps
*Rload vload 0 0.75

* Step Load 15-25Amps
Gr vload 0 VCR rvolt 0 1
Vr rvolt 0 PULSE(0.1v 0.06v 160u 0.1u 0.1u 40u 300u)

* Vin Bulk Bypassing Capacitors
*Cc1 c1 0 82uf
*Rc1 c1 vin 39m
*Cc2 c2 0 82uf
*Rc2 c2 vin 39m
*Cc3 c3 0 82uf
*Rc3 c3 vin 39m
*Cc4 c4 0 82uf
*Rc4 c4 vin 39m
*Cc5 c5 0 82uf
*Rc5 c5 vin 39m

*Vvcc12 vcc12r 0 DC 12v   $ Simulate 12v supply inductance
*Rvcc12 vcc12r vcc12 100m  $ Supply Series Resistance
*Lvcc12 vcc12 vin 0u
Vin vin 0 DC 12v

Vcc vcc 0 DC 5v      $ Ignore MAX4038s Voltage Regulator

X1 clp1 csp1 csn1 dl1 dh1 lx1 bst1
+  clp2 csp2 csn2 dl2 dh2 lx2 bst2
+  vsp vsn diff ean eaout MAX5038EAI15

*C30 clp1 0 470p IC=1v $ Phase 1 Compensation Network
*R6  clp1 1 1k
*C29 1 0 6.8n

*C28 clp2 0 470p IC=1v $ Phase 2 Compensation Network
*R5  clp2 2 1k
*C27 2 0 6.8n


C30 clp1 0 47p IC=1v $ Phase 1 Compensation Network
R6  clp1 1 10k
C29 1 0 0.68n

C28 clp2 0 47p IC=1v $ Phase 2 Compensation Network
```

```
R5  clp2 2 10k
C27 2 0 0.68n

X2 vin dh1 lx1 irf6602  $ Phase 1 Discrete Components
X3 lx1 dl1 0 irf6601
D1 lx1 0 MBRS340T3
L1 lx1 csp1 0.69uH
R2 csp1 csn1 1.58m      $ Inflated to compensate for PCB resistance
V3 csn1 vload DC 0v
*C12 bst1 lx1 0.1u IC=5v
*D3 bst1 vcc MBR0520L
V10 bst1 lx1 DC 5v

X4 vin dh2 lx2 irf6602  $ Phase 2 Discrete Components
X5 lx2 dl2 0 irf6601
D2 lx2 0 MBRS340T3
L2 lx2 csp2 0.69uH
R3 csp2 csn2 1.58m
V4 csn2 vload DC 0v
*C13 bst2 lx2 0.1u IC=5v
*D4 bst2 vcc MBR0520L
V20 bst2 lx2 DC 5v


Rin ean diff 5.11k     $ Feed Back Resistors
Rf eaout ean 51.1k
Rx ean vcc 75k
*Rf eaout ean 100k
*Rx ean vcc 200k

V1 vsn 0 DC 0v        $ Sense Connections
V2 vsp vload DC 0v


*************************
* MAX5038 PWM Controller *
*************************
* NOTES:
*  - Set for 1.5v operation
*  - 250Khz Clock
.subckt MAX5038EAI15
+ clp1 csp1 csn1 dl1 dh1 lx1 bst1
+ clp2 csp2 csn2 dl2 dh2 lx2 bst2
+ vsp vsn diff ean eaout
.param clkfreq=250000
.param clkrise=5n
.param clk2delay='1/(2*clkfreq)'

v1 clk1 0 dc PULSE(0 5v 0 clkrise clkrise 200n '1/clkfreq')
v2 ramp1 0 dc PULSE(0 2v 0 '1/clkfreq-clkrise' clkrise 0 '1/clkfreq')
v3 clk2 0 dc PULSE(0 5v clk2delay clkrise clkrise 200n '1/clkfreq')
v4 ramp2 0 dc PULSE(0 2v clk2delay '1/clkfreq-clkrise' clkrise 0 '1/clkfreq')

Xphase1 clp1 csp1 csn1 eaout ramp1 clk1 dl1 dh1 lx1 bst1 ICLC
Xphase2 clp2 csp2 csn2 eaout ramp2 clk2 dl2 dh2 lx2 bst2 ICLC
Xdiff vsp vsn diff DIFF
Xvea ean eaout VEA
.ends MAX5038EAI15

*********************
* Voltage Error Amp *
*********************
* NOTES:
*  - No current drive limits
*  - Frequency response incorrect (Should be unity Gain freq)
.subckt VEA inn out
.param GAIN='pow(10,70/20)'
.param POLE=3000000
.param VHIGH='900mv+600mv'
.param VLOW=0v
.param VREF='1.5v+0.6v'

E1 1 0 VOL='(VREF-v(inn))*GAIN'
R1 2 1 100
C1 2 0 '1/(6.28*100*POLE)'
E2 out 0 VOL='v(2)' MAX=VHIGH MIN=VLOW
.ends VEA

**************************
* Differential Amplifier *
**************************
* NOTES:
*  - No current drive limits
.subckt DIFF inp inn out
.param GAIN=1
.param POLE=3000000
.param RIN=100k
.param VHIGH=5v
.param VLOW=0v
.param VOFFSET=0.6v

Rin inp inn RIN
E1 1 0 VOL='(v(inp)-v(inn))*GAIN'
R1 2 1 100
C1 2 0 '1/(6.28*100*POLE)'
E2 out 0 VOL='v(2)+VOFFSET' MAX='VHIGH+VOFFSET' MIN='VLOW+VOFFSET'
.ends DIFF

**********************
* Inner Current Loop *
**********************
* NOTES:
*  - No Peak Current Comparator
*  - No Shdn
.subckt ICLC clp csp csn gmin ramp clk dl dh lx bst
X1 csp csn 1 CSA
X2 gmin 1 clp CEA
X3 ramp clp 2 CMP
```

```
X4 2 clk q qn FFLOP
X5 dl qn LOWDRV
X6 dh bst lx q HIGHDRV
.ends ICLC

**********************
* Highside FET Driver *
**********************
.subckt HIGHDRV dh bst lx en
E1 on 0 en 0 1 MAX=1 MIN=0
E2 dhi 0 VOL='v(lx)+(v(bst)-v(lx))*v(on)'
R1 dh dhi 1
.ends HIGHDRV

**********************
* Lowside FET Driver *
**********************
.subckt LOWDRV dl en
E1 dli 0 VOL='v(en)'
R1 dl dli 1
.ends LOWDRV

***************************
* Current Error Amplifier *
***************************
.subckt CEA inp inn out
.param GM=550uS
.param CHIGH=320u
.param CLOW=-320u

G1 0 out inp inn GM MAX=CHIGH MIN=CLOW
.ends CEA

***************************
* Current Sense Amplifier *
***************************
.subckt CSA inp inn out
.param GAIN=18
.param POLE=4000000
.param RIN=4k
.param VHIGH=5v
.param VLOW=0v

Rin inp inn RIN
E1 1 0 VOL='(v(inp)-v(inn))*GAIN'
R1 2 1 100
C1 2 0 '1/(6.28*100*POLE)'
E2 out 0 VOL='v(2)' MAX=VHIGH MIN=VLOW
.ends CSA

***********************
* Nonlinear Comparator *
***********************
.subckt CMP inp inn out
Eopamp out 0 inp inn 100000 MAX=5 MIN=0
.ends CMP

*********************************
* Linear Flipflop Approximation *
*********************************
.subckt FFLOP S R Q QN
X1 S Sn INV
X2 R Rn INV
X3 Rn Qn Q0 NAND2
X4 Sn Q Qn0 NAND2

R1 Q Q0 100
R2 Qn Qn0 100
C1 Q 0 100p  IC=0v
C2 Qn 0 100p IC=5v
.ends FFLOP

*****************************
* Linear Nand Approximation *
*****************************
.subckt NAND2 in1 in2 out
E1 out 0 NAND(2) in1 0 in2 0 0v 5v 5v 0v
.ends NAND2

**************************
* Linear Or Approximation *
**************************
.subckt OR2 in1 in2 out
E1 out 0 OR(2) in1 0 in2 0 0v 0v 5v 5v
.ends OR2

*********************************
* Linear Invertor Approximation *
*********************************
.subckt INV in out
E1 out 0 VOL='5-v(in)'
.ends INV

* Generate transient data
.probe I(Gr)
.tran 1us 250us UIC

.end
```