Methods for Extending High-Performance Automated Test Equipment (ATE) using Multi-Gigahertz FPGA Technologies

A Dissertation
Presented to
The Academic Faculty

by

Ashraf M. Majid

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy in the
School of Electrical and Computer Engineering

Georgia Institute of Technology
May 2011

Methods for Extending High-Performance Automated Test Equipment (ATE) using Multi-Gigahertz FPGA Technologies

Approved by:

Dr. David C. Keezer, Advisor
School of Electrical and Computer Engineering
Georgia Institute of Technology

Dr. James O. Hamblen
School of Electrical and Computer Engineering
Georgia Institute of Technology

Dr. Abhijit Chatterjee
School of Electrical and Computer Engineering
Georgia Institute of Technology

Dr. Shijie Deng
School of Industrial and Systems Engineering
Georgia Institute of Technology

Dr. Linda S. Milor
School of Electrical and Computer Engineering
Georgia Institute of Technology

Date Approved: March 29th, 2011

As no man is born an artist, so no man is born an angler.
-Izaak Walton – The Compleat Angler

To my mother, Shahida Majid,
and my family,
for their endless love, encouragement and support,
without which, this work would not have been possible.

ACKNOWLEDGEMENTS

My career at Georgia Tech started as I first stepped through the halls of this great institution in 1998. As an eager undergraduate student during the ramp-up to the dot-com era, my only plan was to graduate and enter the high-tech industry as quickly as possible. I had no clue that this institution had a lot more planned for me over the next 13 years. Fortunately, along the way, I was able to meet wonderful people whose guidance, support, and above all friendship have enabled this achievement.

First of all, I would like to express my deepest gratitude to my advisor Dr. David Keeezer for all his support throughout my graduate career at Georgia Tech. If it were not for his willingness to take a gamble on a young student, with no experience in digital testing, this research would not have been possible. His guidance and wisdom have proven to be invaluable in the development of my career. Also, his encouragement and patience throughout the years have made this entire experience joyful. I would also like to give a special thanks to my lab mate Carl Grey who has been instrumental in much of our work and who has always been willing to provide his technical expertise.

I would like to thank both my reading committee members Dr. Abhijeet Chatterjee and Dr. Linda Milor, who have been highly supportive of this research and available for guidance and advice. Dr. Chatterjee has helped our research group with numerous publications and industry liaisons throughout the years and deserves a special acknowledgement. I would like to express my gratitude to Dr. James Hamblen for serving on my proposal and dissertation defense committees. Furthermore, I would like to

thank Dr. Shije Deng for his assistance and guidance during my time in the quantitative finance program, and also for serving on my dissertation defense committee.

I would also like to acknowledge of my undergraduate professors at Georgia Tech. I was fortunate enough to take courses throughout various schools at Georgia Tech and able to meet many wonderful professors, each of whom contributed to my development. I would especially like to thank Dr. David Anderson who introduced me to this intriguing world of research. Allowing me to join his Digital Signal Processing lab as an undergraduate, he helped sow the seeds for my desire of research and further studies.

Over the last three years, I have been working full time at SunTrust Bank's capital markets division while finishing this research. This would not have been possible without the support and patience of my entire team at SunTrust. I would especially like to thank my manager Al Kolesar, who has proven to be a great mentor over the years and pivotal in developing my newfound career in investment banking.

Finally, I am eternally indebted to my family, without whose support, this work would have been impossible, if not meaningless. This entire journey was not only made bearable, but pleasurable and fun by the never ending love and support of my mother and sister. My father's own academic achievements, and hard work ethic, inspired and drove me to endeavors I attempt today. Their understanding and patience has been vital in this entire process. Furthermore, this section would not be complete without acknowledging the wonderful lifelong friendships I have developed during my tenure at Georgia Tech and in Atlanta. These friendships have proven to be an essential support structure providing the encouragement required for completing this research. I have come to value all these relationships very very dearly.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF SYMBOLS AND ABBREVIATIONS

ATE	Automated Test Equipment
AC	Alternating Current
ASIC	Application-Specific Integrated Circuit
ATPG	Automatic Test Pattern Generator
BCB	Benzocyclobutene
BGA	Ball Grid Array
BIST	Built-In Self-Test
BLVDS	Bus Low-Voltage Differential Signaling
BoN	Bed of Nails
BOST	Built-Off Self-Test
CAD	Computer-Aided Design
CMT	Clock Management Tiles
CPU	Central Processing Unit
CRC	Cyclical Redundancy Check
CSP	Chip Scale Package
CUT	Circuit Under Test
DAC	Digital-to-Analog Converter
DC	Direct Current
DDJ	Data Dependent Jitter
DDR	Double Data Rate
DFT	Design for Testability
DIB	Device Interface Board
DLC	Digital Logic Core

DLL	Delay-Locked Loop
DRAM	Dynamic Random-Access Memory
DSP	Digital Signal Processing
DUT	Device Under Test
FIFO	First In, First Out
GaAs	Gallium Arsenide
GPS	Global Positioning System
GSG	Ground-Signal-Ground
GTL	Gunning Transceiver Logic
GTLP	Gunning Transceiver Logic Plus
HDL	Hardware Description Language
HID	Human Interface Device
HSTL	High-speed Transceiver Logic
I/O	Input and Output
IC	Integrated Circuit
IEEE	Institute of Electrical and Electronics Engineers
InP	Indium Phosphide
IP	Intellectual Property
ISA	Instruction Set Architecture
ITRS	International Technology Roadmap for Semiconductors
JTAG	Joint Test Action Group
LFSR	Linear Feedback Shift Register
LVCMOS	Low-Voltage Complementary Metal Oxide Semiconductor
LVDS	Low-Voltage Differential Signaling
LVTTL	Low-Voltage Transistor-Transitor Logic

MAC	Media Access Control
MCC	Multiple Copper Columns
MCM	Multi-Chip Module
MCP	Multi-Chip Package
MEMS	Microelectromechanical Systems
MUX	Multiplexor
ORA	Output Response Analyzers
PA ATE	Protocol Aware ATE
PC	Personal Computer
PCB	Printed Circuit Board
PCI	Peripheral Component Interconnect
PCIE	PCI Express
PCI-X	PCI eXtended
PEC	Pin Electronic Card
PECL	Positive Emitter-Coupled Logic
PISO	Parallel In, Serial Out
PLL	Phase-Locked Loop
PMU	Parametric Measurement Unit
PRBS	Pseudorandom Binary Sequence
PROM	Programmable Read-Only Memory
RAM	Random Access Memory
RF	Radio Frequency
RIO MGT	RocketIOTM Multi-Gigabit Transceiver
RSDS	Reduced Swing Differential Signaling
RX	Receive

SATA	Serial Advanced Technology Attachment
SB	Solder Bumps
SECT	Standard For Embedded Core Test
SerDes	Serializer/Deserializer
SIE	Serial Interface Engine
SiGe	Silicon-Germanium
SiP	System-in-a-Package
SIPO	Serial In, Parallel Out
SMA	SubMiniature version A
SoC	System-on-a-Chip
SoL	Sea of Leads
SRAM	Static Random-Access Memory
SSC	Stretched Solder Column
SSTL	Stub Series Transceiver Logic
SW	Software
TAM	Test Access Mechanism
TCK	Test Clock
TDI	Test Data In
TDO	Test Data Out
TMS	Test Mode Select
TPG	Test Pattern Generator
TSP	Test Support Processor
TX	Transmit
UBM	Under Bump Metallization
ULVDS	Ultra Low-Voltage Differential Signaling

USB	Universal Serial Bus
UV	Ultra-Violet
VCO	Voltage-Controlled Oscillator
VHDL	VHSIC Hardware Description Language
WLP	Wafer-Level Package

SUMMARY

This thesis presents methods for developing multi-function, multi-GHz, FPGA-based test modules designed to enhance the performance capabilities of automated test equipment (ATE). In recent years technological advancements in semiconductor technology have outpaced advances in ATE testing capabilities, thereby causing significant challenges for new high-speed device testing. The main motivation of this research was to develop solutions that address these challenges.

The methods shown in this thesis are used to develop a design approach that utilizes a test module structure in two blocks. A core logic block is designed using a multi-GHz FPGA that provides control functions. Another block called the "application specific" logic block includes components required for specific test functions. Six test functions are demonstrated in this research: high-speed signal multiplexing, loopback testing, jitter injection, amplitude adjustment, timing adjustment. Furthermore, the test module is designed to be compatible with existing ATE infrastructure, thus retaining full ATE capabilities for standard tests. Experimental results produced by this research provide evidence that the methods are sufficiently capable of enhancing the multi-GHz testing capabilities of ATE and are extendable into future ATE development.

The modular approach employed by the methods in this thesis allow for flexibility and future upgradability to even higher frequencies. The methods allow a next-generation FPGA to be quickly integrated into a test module to increase performance. Similarly, new components can be designed into the "application specific" block for additional test functionality. Therefore the contributions made in this thesis have the potential to be used into the foreseeable future for enhancements to semiconductor test capabilities.

CHAPTER 1

INTRODUCTION

The objective of this research is to develop feasible and economical solutions to testing high-speed digital devices at multi-GHz rates. Testing high-speed digital devices at full speeds is essential to assure that manufactured devices meet design specifications and function properly throughout the entire range of intended operation. Increases in device performance and functionality have resulted in challenging problems pertaining to testing them effectively. Many current methods of testing are either inefficient or prohibitively expensive. This research presents methods to develop an efficient test system to cost-effectively test high-speed semiconductor devices.

Technology roadmaps have accurately predicted rapidly increasing clock and data rates of electronic devices [1]. Current roadmaps indicate this trend will continue into the foreseeable future [2]. Gordon Moore states in his famous 1965 article [3] – the basis for the well-known "Moore's Law"- that the number of components that can be placed into a device nearly doubles every two years. This "law" has been upheld for nearly half a century, despite meeting many physical limitations on the way. Furthermore, this trend is expected to continue without major obstacles until at least the year 2015 or perhaps later [4], thereby producing exponentially complex, higher-performance devices.

Automated Test Equipment (ATE) has generally been used to test high-volume semi-conductor devices over the last four decades. Over this period, ATE performance has improved and new capabilities have been added. However, the advances in many ATE performance measures and capabilities have not kept up with the advances in semi-

conductor technology [1][2]. One commercially available system can run up to 12.8Gbps after installing add-on instrumentation [5]. Typical base systems are limited to lower speeds [6]. This situation has resulted in the production of advanced, complex devices, but not a feasible way to test their complete functionality. This problem has become so critical to the development of higher performance devices, that the paradigm in which design is associated with research and development and testing is associated with manufacturing no longer stands [7]. Design and test can no longer be dealt with as separate issues – they must be approached hand in hand in order to efficiently develop next generation high-performance devices.

Design for Testability (DFT) incorporates certain testability features into the design stage of semiconductor devices. These improvements typically include design modifications and enhancements such as serial scan, boundary scan, and built-in self-test (BIST) [7]. Most DFT and BIST methods can be used to verify internal logic and structural connectivity, thereby simplifying external testing required.

As the complexity of devices increase, testing the entire device globally becomes inefficient and complex. Incorporating DFT techniques into various components of devices produces substantial benefits [8][9]. This has caused most semiconductor manufacturers to incorporate DFT features into complex device design. Assuming there is a practical degree of DFT and BIST methods on the device, its testing complexity can be reduced to a subset of traditional testing [10]. However, functional testing at full clock speeds remains as one of the most critical tests required as DFT and BIST generally do not operate at these speeds. Furthermore, environmental and parametric testing is still required, as these tests are not covered by DFT and BIST methods. Therefore, although

the required external tests are reduced by incorporating DFT, high-performance devices still require high-speed functional tests and traditional testing such as environmental and parametric testing.

ATE have traditionally been able to perform most semiconductor testing requirements mentioned above. Parametric tests are performed by elaborate parametric measurement units (PMUs) within an ATE. Reliability tests are performed by an ATE's full suite of sophisticated instrumentation and software. These tests do not fundamentally change with increasing clock and data rates. Functional testing, on the other hand, is limited by an ATE's performance capabilities. Purchasing new ATE systems when higher-performance testing is required can be cost prohibitive (historical ATE buy rates have been reduced by a factor of two since 1981 [11]). Upgrading existing ATE performance with additional pin electronics cards (PECs) from its manufacturer is a cheaper option when available. Therefore there is a need for a test system that can perform functional tests beyond the performance capabilities of available ATE. Based on technology road maps, test systems will be required to test devices running at speeds in excess of 10Gbps.

To approach the problem of increasing test performance requirements, a modular test system is presented in this thesis. The test system is used to enhance specific ATE performance criteria such as high-speed signal generation, high-speed loop-back testing, etc. The test system is designed with a core component that aims to exploit state-of-the-art field programmable gate array (FPGA) technologies. Current FPGA performance exceeds many performance criteria of ATE [6][12]. Furthermore the use of an FPGA allows the test system to operate independently of the ATE and without any of its

resources which tend to be relatively expensive. With the FPGA, the core component controls and generates many test functions itself. However, for comprehensive testing, test functions such as parametric measurement and reliability testing are required. As discussed above, these traditional, lower-speed parametric tests are better handled by and ATE. Hence, to retain full testing functionality, the designed solutions must be compatible with existing ATE infrastructure.

In addition to a core component, the test system can be designed with logic to enhance ATE capabilities based upon specific test application requirements. For instance, typical ATE are limited in signal speeds up to 3.2Gbps. If higher speed test signals are required for the test application, logic can be added to the test module to increase signal speeds to above 3.2Gbps. Similarly, if other test capabilities are required such as jitter injection, additional logic can be added to the test module.

Therefore the objective of the research presented in this thesis is to develop feasible methodologies for extending ATE performance capabilities using multi-GHz FPGAs. The research develops an approach that can be adapted as new functionalities are required and further technological advances take place. The approach consists of the design of a test system in separate blocks, specifically a core logic block and an application specific block. When higher performance is required of the test system, the core logic block can be redesigned independently of the application specific logic. Similarly, the application specific logic can be redesigned or additional logic added to accommodate new or improved test applications. By limiting the solution to focus on specific enhancements, the methods presented in this thesis allow ATE performance to be extended in a feasible, timely and cost-effective manner.

The organization of this thesis is as follows. In Chapter 2, technology roadmaps are presented and discussed. Also the history of ATE is given and the evolution of testing with concepts such as DFT and BIST is described. Two different methods of previous work done in the area to enhance high-speed digital testing capabilities are presented in Chapters 3 & 4. Chapter 3 presents modular extension techniques of ATE to accommodate high-speed digital testing. This method of test enhancement extends resources from the ATE to produce higher performance test capabilities. Chapter 4 presents a FPGA-based stand-alone test system that is capable of operating independently of an ATE and providing high-performance test capabilities.

The reaming research presented in this thesis uses the lessons learned in both Chapters 3 and 4 and develops an FPGA-based modular extension test system designed to work within existing ATE infrastructure, but not to use ATE resources. The test system can be operated with the FPGA, thus allowing operation independent of an ATE. Chapter 5 discusses in detail the concept and design of the test system. As the FPGA in the test system is a critical component of it, the selection, design, and use of the FPGA is discussed in Chapter 6. Designing high-speed digital systems requires additional considerations versus designing at-speed systems. At higher speeds, passive system elements can cause undesirable effects, essentially altering system performance. These additional considerations are discussed in Chapter 7 and the physical design of the test system is presented. In Chapter 8, the experimental results of this thesis are presented and the developed test system is characterized. Finally in Chapter 9, conclusions of this work are presented and future work discussed.

CHAPTER 2

BACKGROUND AND HISTORY

In the semiconductor industry, specialized computers are used to test devices. Over the past 50 years or so, these specialized computers, or ATE as they are formally known as, have become bigger, faster and offer more functionality than ever before. This chapter aims to present a background on testing and the history of ATE evolution.

Semiconductor testing is largely driven by the technological trends of the semiconductor industry. Various technologies, such as digital, optical, MEMS, RF, etc. require different testing methodologies. Furthermore, technological advancements in these fields merit different methodologies of test techniques due to the complexities imposed by these advancements. Therefore it is relevant to discuss these technological advancements to fully understand the nature of modern day semiconductor testing. Due to this reason, in this chapter, technology roadmaps are discussed first. ATE are the primary tools used in testing semiconductor devices, thus after presenting technology roadmaps, the history and development of ATE is discussed. The last section of this chapter discusses general testing methods and how they have been developed along with ever increasing semiconductor device functionality.

2.1 Technology Roadmaps

Over the past few decades, the ever increasing performance and transistor count of semiconductor devices has caused a phenomenal change in semiconductor test requirements. Consumer demand during the 80s and 90s focused mainly on increased performance of electronics. However the last decade witnessed a fundamental shift in market demand, as technological advancements allowed the integration of various semiconductor technologies. These technological advancements include technologies such as Multi-chip modules (MCM), Multi-chip package (MCP), System-on-a-Chip (SoC), System-in-a-Package (SiP) and 3D packaging, which have allowed the integration of various semiconductor technologies such as digital, RF, optical, MEMS, etc. devices into one system [13]. Based on these capabilites, consumer demand has shifted from not only higher performance from electronic devices, but more functionality as well.

The demand growth of simple one function devices has been steadily declining. In many cases, such as the point-and-shoot digital camera, sales have actually been declining since 2008 [14]. Current market demand indicates that consumers prefer to use their smartphone devices, and demand that a smartphone also function as a high-resolution camera, GPS device, motion sensor, etc. This level of integration has caused significant challenges in testing as it is ultimately the application requirements or specifications that determine test requirements. Furthermore, each technology requires different test methodologies. Along with testing requirements for various technologies, increased device complexity has increased test complexity, which further translates to increased test cost. The cost of testing a device is now a large part of its total manufacturing cost [15]. Fabrication costs per function have generally decreased 25-30%

annually, while test cost per function has decreased only 5-10% annually [16]. Given that test costs are already a large part of the total manufacturing cost; the statistics indicate that testing costs will soon be the majority of a device's final cost. Figure 2.1 shows a plot of manufacturing and test costs per function over the past 40 years and predicted for the next 20 years. It is expected that the cost of testing a device will become the majority of a device's cost shortly within this decade [17].

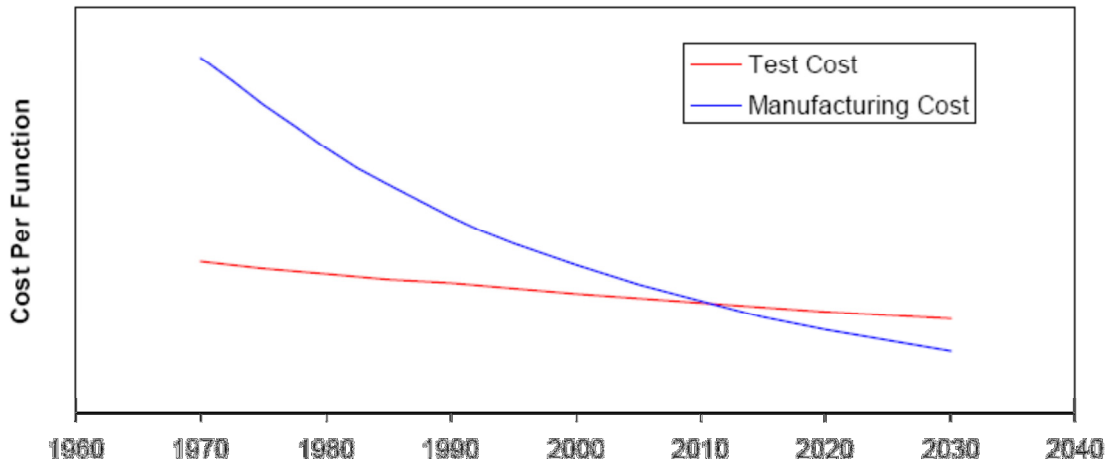


Figure 2.1 Semiconductor test cost vs. manufacturing cost [17].

According to the International Technology Roadmap for Semiconductors (ITRS), the capital cost for a test cell is the following:

$$C_{CELL} = C_{BASE} + C_{INTERFACE} + C_{POWER\text{-}SUPPLIES} + C_{TEST\text{-}CHANNELS} + C_{OTHER}$$

Where C_{BASE} is the cost of the base system (channels not included), $C_{INTERFACE}$ is the cost of interfacing devices, $C_{POWER\text{-}SUPPLIES}$ is the cost of power supplies, $C_{TEST\text{-}CHANNELS}$ is the cost of channels, and C_{OTHER} is the remaining costs such as floor space, cooling systems, etc.[1]. The total cost of a test cell can be broken down into more meaningful figures by dividing costs by testing throughput, to result in a per device test

cost. The current top drivers of test cost and future test cost drivers according to the ITRS are shown in Table 2.1.

Table 2.1 Test Cost Drivers [1]

Current Top Test Cost Drivers	Future Test Cost Drivers
ATE Capital Expenditures	Device Performance Metrics
ATE Interface Expenditures	New Defects and Reliability Problems
Cost of Test Program Development	Known Good Die Requirement
Test Time an Test Coverage	Test Requirements of Packaging

Device performance metrics will be one of the main future test cost drivers. The cost of testing high-speed I/O has become very significant [1]. High frequency I/O technology continues to show significant growth in speed and port count. Trends such as these are expected to continue as technology roadmaps clearly predict rapidly increasing clock and data rates into the foreseeable future. Table 2.2 shows the ITRS Test and Test Technology Roadmap from 2005. It can be noted from the table that not only are the data rates for future semiconductors expected to increase, but their feature sizes are expected to decrease.

Table 2.2 ITRS Test and Test Technology Roadmap 2005 [2]

Year	2005	2006	2007	2008	2009	2010	2011	2012	2013
Wafer Test - number of Sites	8	8	8	16	16	16	16	32	32
Half Pitch size (nm)	90	78	68	59	52	45	40	36	32
Chip-to-board (off-chip) speed (high-performance, for peripheral buses) (MHz)	3125	3906	4883	6103	7629	9536	TBD	14901	18626
I/O data rate (GT/s)	0.1 - 3	0.1 - 6	0.1 - 6	0.1 - 6	0.2 - 12	0.2 - 12	0.2 - 12	0.2 - 15	0.2 - 15

Based on this roadmap and technology trends, there are a few important areas of concern regarding test. The first is the rampant data rate increase. Due to higher data rates, test systems must be capable of such high-speed data rates. Second, test systems must be able to accommodate the increase in pin counts enabled by the lower feature sizes. In 2009, up to 250 pairs of 7Gbps backplane style SerDes channels were found in some applications [1], however most applications will have 32 channels or less. The third area of concern is test system bandwidth. At 10Gbps, the bandwidth requirement becomes 20GHz or higher and providing such an interface including connections through PCB, cables, connectors, etc. becomes a monumental engineering task. The fourth area of concern is jitter, and this becomes a significant concern at multi-GHz data rates. At a 10Gbps speed, bit periods are 100ps and only 50ps of jitter can render half that bit period useless. Dealing with such high-speed signals and introducing low amounts of jitter is not a trivial task. The final area of concern is the cost factor. Traditionally high-speed devices were designed as high-performance high-priced devices. This has been rapidly changing as the thirst for high-speed data in commoditized hand-held devices has been insatiable. Furthermore, with accelerating technology improvements, product lifecycles have become shortened, which has made cost savings from "mature" technologies very difficult to achieve.

The concerns discussed above indicate a need for a test system that:

i) is capable of multi-GHz I/O

ii) is capable of accommodating multiple data channels

iii) is capable of high bandwidth through all components

iv) adds low jitter to the test signal

v) feasibly and economically tests devices

In addition to the new requirements for a test system discussed above, a comprehensive test system should always be capable of standard tests functions such as timing adjustment, voltage adjustment, parametric tests, etc. Taking all these concerns into consideration, multi-dimensional challenges arise when developing new test instrumentation. Semiconductor testing is generally done using ATE. The next section discusses the history and development of ATE systems.

2.2 Automated Test Equipment

The roots of modern ATE were developed in the 1960s by Nick DeWolf, a well-known test engineer during that era, and often referred to as the father of ATE [18]. DeWolf and Alex d'Arbeloff together founded a company called Teradyne whose business plan was to manufacture and design semiconductor test equipment. In 1966, Teradyne introduced an integrated circuit tester, the J259, which was the first tester to use a minicomputer for control, thus launching the first ATE [19].

The first generation of ATE was comparatively primitive to modern day ATE. It wasn't until the late 70s and early 80s that ATE started to achieve complex functionality and the use of automatic handlers and probers. During this time multi-function ATE were being introduced that could test mixed-signal, RF, optical as well as digital components.

In the mid-to late 80s digital devices were rapidly evolving into the GHz range. This required test systems capable of testing at GHz rates. Many manufacturers developed systems using various techniques to increase speeds. For instance, the Megatest MegaOne and the Tektronix S-3295 used multiplexing concepts on adjacent

pins to double output frequencies [20]. However these systems could operate at only a few hundred MHz, with timing accuracy of only 0.5-1ns, and above all, cost nearly $3M for a 256-pin system. Therefore around this time, ATE functionality could not keep pace with the rapid advancements of semiconductor technologies. In order to achieve GHz speeds and keep up with technological trends, test engineers either developed custom test heads for existing ATE or developed extension systems to produce GHz signals. These enhancements are the topic of the next chapter.

Despite the lack of performance during this period, a significant shift did occur in ATE architecture. Up to this point, most ATE systems used a "shared resource" architecture, where a few dedicated "per-pin" electronics, such as formatter or comparator served as an interface to the device under test (DUT). When lower pin counts were required, this allowed a cost advantage. However as more and more switching and multiplexing techniques were employed, issues such as test calibration and programming became very tedious due to signal routing. Also, at higher speeds bandwidth limitations produced undesirable results. To overcome these issues, manufacturers introduced "per-pin" architecture [20]. Under this architecture, each tester pin included its own dedicated pin electronics such as pattern generation, timing control, etc. This eliminated the complex switching algorithms and timing issue between pins, and created a more simplistic, efficient and easy to use system as each channel could be controlled independently. This architecture in turn reduced the cost of the base system, and allowed users to purchase additional channels when required. An overview of the shift from shared resource architecture to per-pin architecture is shown in Figure 2.2.

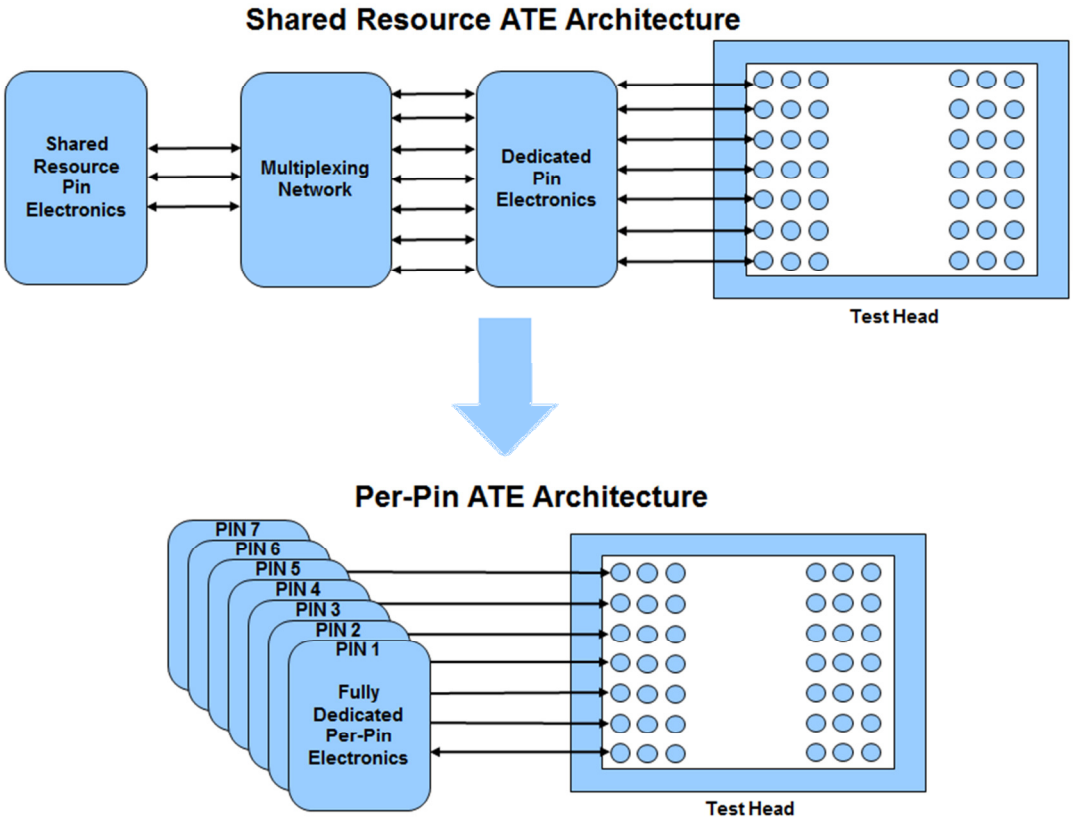


Figure 2.2 ATE Architecture Change

Hewlett Packard released its first series of testers based on this new "per-pin" architecture in the 90s. The 83000-F660 was the state-of-the-art tester at the time. However, it could not reach 1GHz; its highest speed was 666MHz. The cost of the system ranged between $1-3M and despite the cost and performance limitations, these systems were the industry leader and quite popular.

Other players in the market included Teradyne, Advantest and Credence systems. All had their own custom systems, but none running above 1GHz. However around this time, a new standard for modern ATE architecture did evolve which included a test mainframe/test head, a development workstation, and power supplies. Much of the testing done on ATE was power intensive and generated much heat. Thus most ATE were

equipped with efficient cooling systems, for example the HP8300 was equipped with water cooling features.

Despite the fact that tester performance could not keep pace with device performance, many additional functions were added and improved. These included elaborate parametric measurement units, sophisticated test pattern generation algorithms, large amounts of memory, etc. Adding such additional functions to ATE have kept their costs around the same level. ATE cost per pin for high-end systems still hovers around $3,000-$10,000 per channel. Therefore a 256 channel ATE can easily range from $1M to above $5M.

In order to lower ATE costs, manufacturers pushed the concept of open-architecture test platforms in the early 2000s [13]. The concept of open-architecture testers is similar to that of building a custom PC, in which one can purchase a motherboard, a processor, a video card, etc. separately in order to build one system. Similarly ATE manufacturers developed base test systems to which PECs could be added to. The idea was that consumers could purchase a base system and only the cards they required, thus keeping costs down [21]. Furthermore this allowed the design of higher end cards that could be sold to consumers that required high-performance testing. For example Verigy (formerly HP's Test Systems division) develops a base system called V93000 SOC (the latest system evolved from the HP83000). The system can accept cards with functions such as digital nano-electronics testing, high-speed digital testing, high-end consumer mixed-signal testing, etc. [5]. Despite this new model, the demand for higher-performance, greater accuracy and increased vector memory have offset most cost savings achieved. The base system for the V93000 costs well over $1M, and adding

PECs for required high-speed testing can create a total price tag well into the $3-5M range. Furthermore, there haven't been significant third-party efforts to develop PECs for ATE due to their complexity and performance requirements. Independent PEC development by third-parties could have lowered costs. Therefore the ATE industry is yet to experience significant cost reductions.

The latest development in the ATE industry is what has been termed Protocol Aware ATE (PA ATE). PA ATE came into prominence in 2007 and was formally debuted in [22]. The concept for PA ATE arose as higher levels of integration allowed the manufacture of diverse devices using technologies such as SoC. Device manufacturers with substantial IP libraries could develop an entire true system with diverse IP blocks in a single process on a single die with minimal development time. Each IP block could have different protocols such as JTAG, PCI, PCI-E, USB, SATA, serial flash, SRAM/DRAM, etc. And each protocol would require its own test strategy, thus testing the entire device on a global level would be high inefficient if not impossible. PA ATE aims to solve this problem by essentially natively emulating, in real time, chip I/O at the protocol level. Programmable interfaces in PA ATE are used to perform real-time state detection to handshake with a device using its native protocol. Test strategies are developed for each protocol, and once a communication link to the device is established, testing is performed in a more efficient manner.

PA ATE enables cooperative test between an ATE and a DUT with realistic device activity, thus improving the quality of test [23]. The ATE provides a suitable test environment with infrastructure such as power supplies, cooling systems, DUT

interfaces, probe cards, etc. Furthermore, the ATE controls overall test flow, I/O levels, I/O timing, etc. and manages results.

PA ATE is most effective when used with DFT techniques. DFT is used to provide access to internal chip resources and allows the ATE to perform low-level structural tests using the required protocol. Low-level structural tests require less ATE complexity and simpler use, which results in lower cost and higher reliability [24]. In fact much of the cost savings in testing have been afforded by DFT techniques. Therefore the next section discusses the evolution of DFT standards and usage in detail.

2.3 Design for testability (DFT)

In the semiconductor industry, testing is the process by which stimuli is applied to a circuit in order to demonstrate its correct operation [25]. Traditional testing of semiconductor devices can generally be divided into three broad categories [26]:

i) Functional tests: these tests comprise of testing input/output pins on the device, its timing characteristics, proper logic handling, behavior, etc.

ii) Environmental testing: tests include operation characteristics under various power and temperature conditions.

iii) Reliability testing: these tests characterize a device's quality, reliability, life expectancy, failure rates, etc.

Out of the above three categories, functional tests are the most involved as they are required to test the proper logic functioning of the device. Comprehensive functional tests

should be able to verify the proper function of all components in a device. A typical modern semiconductor device can contain hundreds of millions of transistors and wires. Every one of these transistors and wires can contain defects that manifest themselves as incorrect opens or shorts. These defects can only be tested via device I/O pins, which there are only a few hundred of. Therefore there is a gap between the large number of on-chip components that need to be tested and the relatively small number of pins through which these test can be performed. This gap necessitates design modifications to improve accessibility from external pins to all internal logic. DFT addresses this problem and is the design of additional on-chip hardware to improve accessibility to internal logic. Improving accessibility is used as a generic term here, as DFT techniques can provide accessibility in a variety of ways from physically providing access to internal logic to generating tests on internal logic and providing access to results.

Modern trends in DFT techniques can be distinguished into three general roles [27]. The first role of DFT is to enable high-quality testing, through the use of additional internal modules designed on-chip. The second role is to provide access to embedded modules within a DUT. The third role is the on-chip generation of stimuli and/or the evaluation of test responses. These roles are discussed further below.

2.3.1 Additional Module based DFT

This technique of DFT requires the inclusion of additional modules into devices to enable external testing. The most common methods of this style of DFT are scan-based designs which allow the access and control of device components connected to internal registers through scan chains. Various types of scans exist based on the depth of the

testing required such as full serial scan, partial serial scan, non-serial san, and system level scan [28]. The most popular scan-based design is the Joint Test Action Group (JTAG) Boundary Scan Standard (IEEE Standard 1149.1) [29]. A serial scan architecture is shown in Figure 2.3.

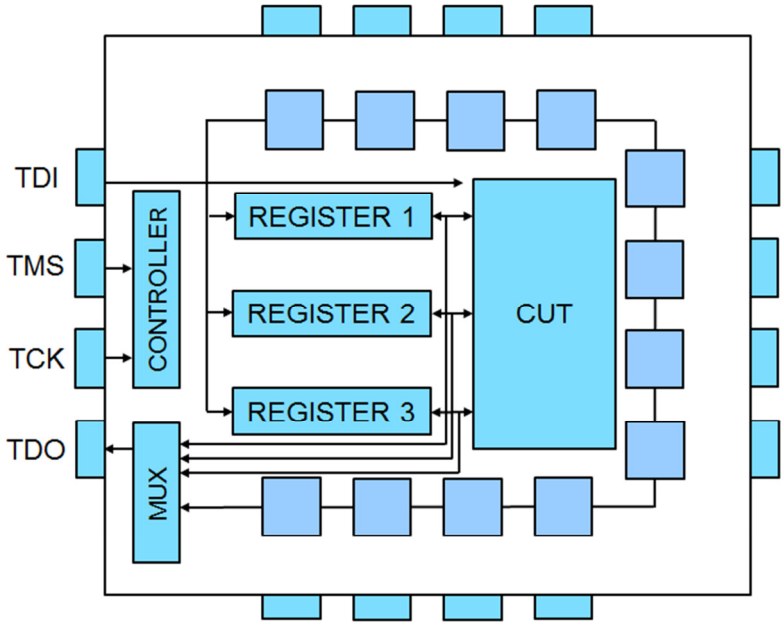


Figure 2.3 Serial Scan Architecture

As shown in the figure, the architecture requires four additional pins. In the JTAG standard these signals are defined as Test Data In (TDI), Test Data Out (TDO), Test Clock (TCK) and Test Mode Select (TMS). Test data and instructions are sent through the TDI pin to internal registers in the device. Each register has certain test functions implemented in it, and the TMS pin is used to select these registers. Once tests are performed, results are sent back over the TDO pin. A clock input is required for the internal logic and provided through the TCK pin. The advantage of using the JTAG

standard is that is allows the scan of multiple circuits. For example a flash memory device connected to eight FPGAs can be scanned at once using this standard.

2.3.2 Embedded Module Access based DFT

Complex devices can be composed of multiple logic modules. Testing these modules in a similar fashion, i.e. modular testing, is an efficient approach to testing as it reduces test generation time through reuse and concurrent engineering [27]. Modular testing requires an on-chip test infrastructure in the form of test wrappers and Test Access Mechanisms (TAMs) [30]. The test wrappers are used to provide access to the targeted module and the TAMs are used to transport data to and from external pins. Therefore, a small number of external pins can access a much larger number of internal signals, thus allowing testing to be performed using external test equipment. The IEEE describes a standard for test wrappers called the Standard for Embedded Core Test (SECT – IEEE Std. 1500) [31]. The standard is similar to the JTAG standard, however slightly different and allows for some customization. This makes SECT a better standard for use with various TAMs.

2.3.3 Self-Test

Test stimuli and response evaluation has traditionally been performed by an external tester. As devices became more complex and access to embedded components became more difficult, external testers could not perform comprehensive testing. To solve this problem, internal test modules are designed onto the device itself in a technique called

BIST. This technique was first applied to memory devices in the late 80s [32]. Additional on-chip circuitry is added on to devices that generate test stimuli and evaluate responses to verify operation.

BIST is generally of two types, online and offline. Online BIST is designed to function when a device is in normal operation, thus being able to detect errors in real-time. Offline BIST is designed to perform testing functions when the device is not in operation mode, such as on power up [28]. Both types of BIST require similar architecture. The main components of BIST are test pattern generators (TPGs) and output response analyzers (ORAs). A BIST controller component controls operation of the TPGs and reads output from the ORA to determine test results. These components can be built using registers and finite state machines.

The complexity of BIST depends on the amount of test patterns required and the number of circuits under test (CUTs). BIST has been an efficient method of DFT for the last 20 years [27]. Miniaturization of transistor features has allowed additional circuitry to be built into devices without much imposition on the devices performance. However as device complexity increased exponentially, employing BIST has become more challenging. Therefore in 1999, Credence Systems Corp. introduced the concept of built-off self-test (BOST) [33]. In this DFT technique, additional self-test circuitry is added to a device, but built off of the device such as on a load board, test fixture, etc. Typically a FPGA is used to control the test circuitry and generate test patterns. This technique permits tremendous flexibility, as it removes physical limitations of BIST. Furthermore, the use of high-performance FPGAs can develop elaborate, high-performance testing solutions.

The three techniques of DFT can be mixed according to testing needs to design optimal results. Using any form of DFT techniques does add additional costs to a semiconductor device. However there are definite economic benefits that justify the use of DFT [8]-[10]. Therefore most semiconductor manufactures employ some level of DFT. While DFT can do a variety of testing, it cannot replace all traditional required testing functions. Environmental and reliability testing still needs to be handled by an external tester. External functional testing can be reduced, but not eliminated by using DFT. For example at-speed tests and characterization may not be handled by DFT. Therefore DFT is often designed to complement the use of an ATE. This has led to the development of low-cost testers by ATE manufacturers [34]. These low-cost testers work adequately provided one is willing to rely on DFT to compensate for reduced tester capability. Although, this solution addresses the concerns of cost, it does not address the concerns of higher performance test requirements. By relying on DFT, a similar argument can be made to develop higher-performance test systems. The development of higher performance test systems is presented in the following chapters.

CHAPTER 3

MODULAR EXTENSION OF ATE TO MULTI-GHZ SPEEDS

Testing electronic devices has been challenging due to their exponential growth in complexity [3]. When next generation devices are developed, a test engineer may only have current generation technologies available for devising test strategies. For instance, when the first generation of GHz capable devices are developed, the only resources available to test may be limited to MHz speeds. In terms of clock and data speeds, this issue was first addressed in the 80s. During that era, standard ATE had testing capabilities up to 200 Mbps [35], however many GaAs devices and high-end application-specific integrated circuits (ASIC) were developed to operate above 1000 Mbps, thus creating a need for multi-GHz testing equipment. Therefore, the first substantial attempts to enable testing capabilities into the Gbps range were seen in the late 80s [35]-[43].

The need for high-speed testing has typically been limited to newer devices and ASICs. Furthermore, these devices may require high-speed testing limited to only a few pins. These new devices are often manufactured in smaller quantities. In the fast paced technology industry, new devices have a limited a “window of opportunity” in which to make an impact, thus requiring a shorter testing turnaround time. As such, purchasing an entire new test system is difficult to justify from a business point of view. Also developing a custom platform to test these devices is not feasible in order to take advantage of the window of opportunity. Taking these concerns into consideration, test engineers have developed solutions extending ATE resources. These solutions involved

developing test electronics that used ATE resources to produce higher-performance tests. In [36]-[40], high-speed ATE extension is demonstrated using custom design test heads. Although this approach is certainly a possibility, designing custom test heads for today's complex ATE systems is a more challenging task. Furthermore, developing custom test heads for specific ATE does not allow portability of the solution to other ATE systems. Building on this approach, [35] & [41]-[43] demonstrate extension of ATE through the development of modular electronic cards using similar principles. These modules are compatible with the existing ATE test head - through a device interface board (DIB) or similar interface; thus allowing portability to other ATE systems. Initial modular extensions developed for GHz speeds included drivers/receivers [35][41], pattern generators [42], clock distribution networks [43], and others. Essentially, these works laid the foundations of using modular electronics card to extend ATE performance.

In this chapter, a number of selected modular ATE extension methods pertinent to this research are presented. In the first section high-speed signal generation using driver and receiver modules is presented. The next section discusses loopback modules, followed by jitter manipulation modules. Modules capable of timing and skew control of high-speed signals are presented and finally high-speed signal switching modules presented.

3.1 High speed signal generation

ATE systems may not have a single signal running in the multi-GHz range, however many have multiple signals running in the GHz range. It is not uncommon for an ATE to have 64 channels or more, each capable of producing up to 1Gbps. The sheer

aggregate bandwidth of these signals can be awesome, however available only in parallel format. Harnessing a few of these signals into one high-speed signal using careful techniques is an approach to generating high-speed data. In this approach a large number of low-speed signals are taken from the ATE and interleaved or multiplexed into a smaller number of high-speed signals. Although not a trivial task by any means, this becomes especially challenging when attempting to generate multi-GHz speeds. For instance, the bit period of a 5Gbps is 200ps. This requires the ATE to maintain very tight timing accuracies across very long test sequences and over multiple channels. Further the electronics used to multiplex these signals must be capable of producing high-speeds without significantly distorting the signal by adding unwanted elements such as jitter, noise, etc.

In [44] and [45], Keezer et al demonstrate the extension of ATE using test-support modules. The modules are designed to work with a DIB. The DIB is an interface between the test-support module and the ATE. DIBs are designed to fit a specific ATE test head. Since DIBs do not have any active components, they can be quickly designed for another test system, thus allowing a test extension module to be ported to another test system. Figure 3.1 illustrates how these test-support modules are used with the DIB. In this research, an Agilent 93000-P1000 ATE with about 900 channels was used.

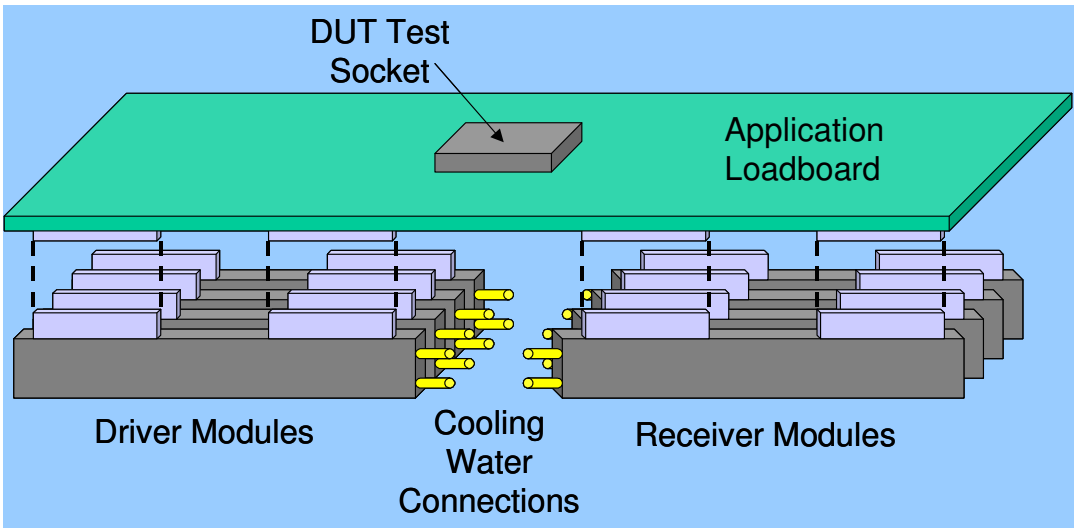


Figure 3.1. Top-level system view of modular extension approach to ATE[44]

To generate and receive high-speed signals from existing ATE signals, typically a driver and receiver module is required. The driver module synthesizes low-speed signals from the ATE into high-speed signals through multiplexing using various logic and control functions from the ATE. These signals are delivered to the DUT. The receiver card performs the opposite function; it takes high-speed signals from the DUT and de-multiplexes them into a larger number of slow speed signals which the ATE can handle. Depending on the application requirements, numerous driver and receiver modules can be used. However care must be taken to order to maintain tight timing accuracy across the ATE and the modules. For temperature stability, the driver and receiver modules have water-cooled plates sandwiched around them as shown in the figure.

A typical application test configuration is shown in Figure 3.2. A DUT may require several multi-GHz test signals. To produce these signals, several high-speed signals are synthesized by the driver modules from incoming low-speed ATE signals and delivered to the DUT. The DUT performs its designed function with the signals and

produces output signals, which are also running at multi-GHz. These high-speed outputs from the DUT are then input into the receiver modules. Since the ATE is also not capable of receiving high-speed signals, the DUT output is de-multiplexed by the receiver module in multiple lower speed signals and relayed back to the ATE. This approach provides a low-cost solution to obtaining high-speed signals from an ATE using little or no external instrumentation. The benefits of this approach include flexibility, customization, and compatibility [44].

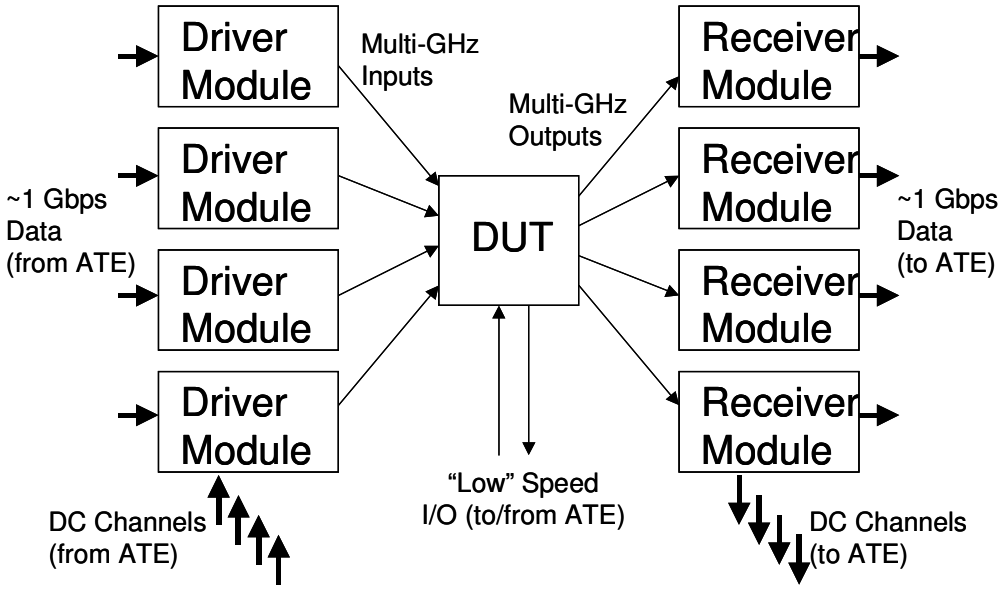


Figure 3.2 Typical application test configuration [44].

Basic driver and receiver modules are designed using multiplexing logic with additional ancillary logic devices. Figure 3.3 shows a logical overview of a basic driver. Multiple data channels from the ATE are supplied to multiplexing logic in the driver module. A multi-GHz clock is used as the select signal on the MUX and multiplexes the input signals into a higher-speed signal. The signal then passes through a buffer where it

is re-clocked; and also allows for amplitude control. Relays are present on this driver module which allow for switching between high-speed and low speed signals (switching is discussed further in Section 3.5). Figure 3.4 shows a basic receiver module designed in a similar fashion. High-speed data from the DUT passes through relays and a buffer after which it enters de-multiplexing logic. Similar, but opposite to the driver module, a multi-GHz clock is used to de-multiplex the signals into multiple lower speed signals which can then be received by the ATE.

Several driver and receiver modules were demonstrated in [44] including 4:1, 8:1 and 16:1 multiplexing schemes. Figure 3.5 shows an output signal produced from a driver using a 16:1 multiplexing scheme to generate a 5Gbps signal. The eye opening is about 0.75UI and can be used for certain test applications. The results clearly demonstrate that external electronic modules can be used feasibly to generate high-speed signals from lower speed ATE resources.

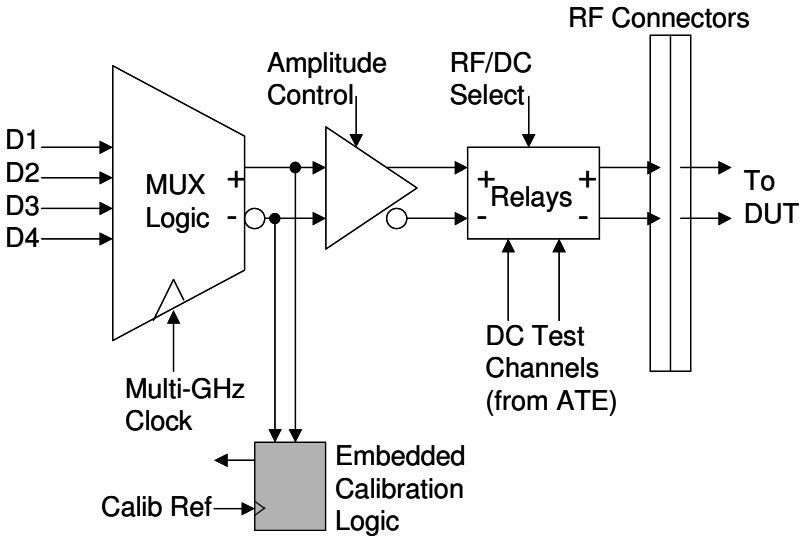


Figure 3.3. Multi-GHz driver module logic [44].

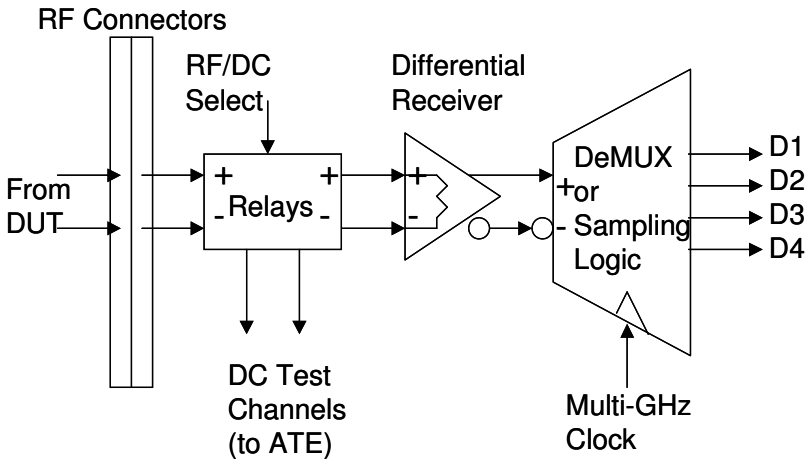


Figure 3.4. Multi-GHz receiver module logic [44].

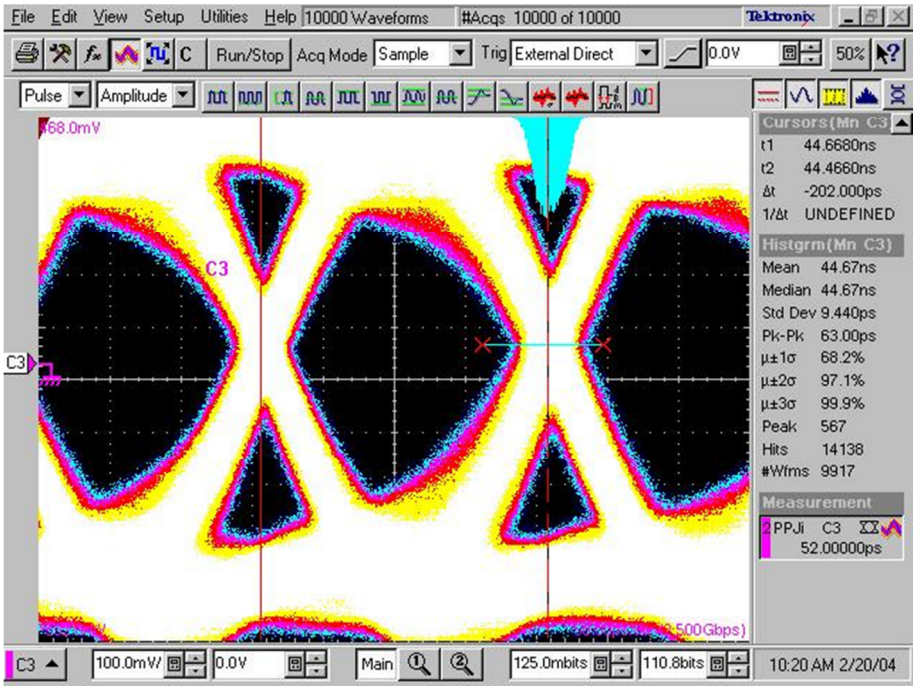


Figure 3.5. High-speed data signal from driver module at 5.0Gbps [44].

3.2 Loopback Testing

As the drive for higher bandwidth has made its way into almost all computing devices, there has been a push for high-speed serial interfaces similar to those found in communication devices [46]. Testing communication devices requires large throughput bandwidth on the order of terabits per second. Generating such tests can involve highly expensive custom ATE equipment and require long test times [47]. Test costs and times for high volume manufactured computing devices have been pressured lower. Therefore using expensive ATE equipment for elaborate tests is not a feasible option. Due to this, DFT methods such as BIST have been pursued. Self-test indeed reduces the full test support required from an ATE. However internal BIST does not accurately reflect a realistic operating environment a device may face [48]-[49]. Furthermore, I/O parametric and signal characterization cannot be validated using BIST. Therefore an external loopback path is ideal in order to test a device under a real world environment.

In 1999, Credence Systems Corp. introduced the concept of BOST [33]. BOST implemented self-test circuitry using FPGAs built off the device and on the test fixture. Test patterns were delivered to the device through load boards, essentially creating an external loop back path. This concept was extended to multi-GHz rates and demonstrated in [50]-[51]. However, multi-GHz devices now produce multi-GHz self-test signals. Therefore when multi-GHz BIST is present in a DUT, its internal BIST circuits generate multi-GHz test patterns that can be used to test itself. Modular test cards can be built to simply loopback the test patterns generated by BIST back into the DUT.

Designing a simple loopback path to handle multi-GHz data rates is not a trivial task. Due to the high-speed nature of the signals, small amounts of jitter and noise from the path can cause undesirable results on the output signal. Furthermore, a loopback path spanning across an ATE test head can cause undesirable attenuation of the signal. In [52] & [53], modular high-speed loopback paths are demonstrated with capabilities up to 12.8Gbps. The modules are designed to connect to an ATE via a DIB similar to the setup shown in Figure 3.1. An overview of a minimal loopback path is shown in Figure 3.6. In addition to loopback testing, low-speed functional tests and parametric measurement may also be required. Therefore relay switches are implemented on these modules to select between loopback signals and ATE signals (switches further discussed in Section 3.5). High performance connectors and relays are utilized in order to support high bandwidth required for multi-GHz signals.

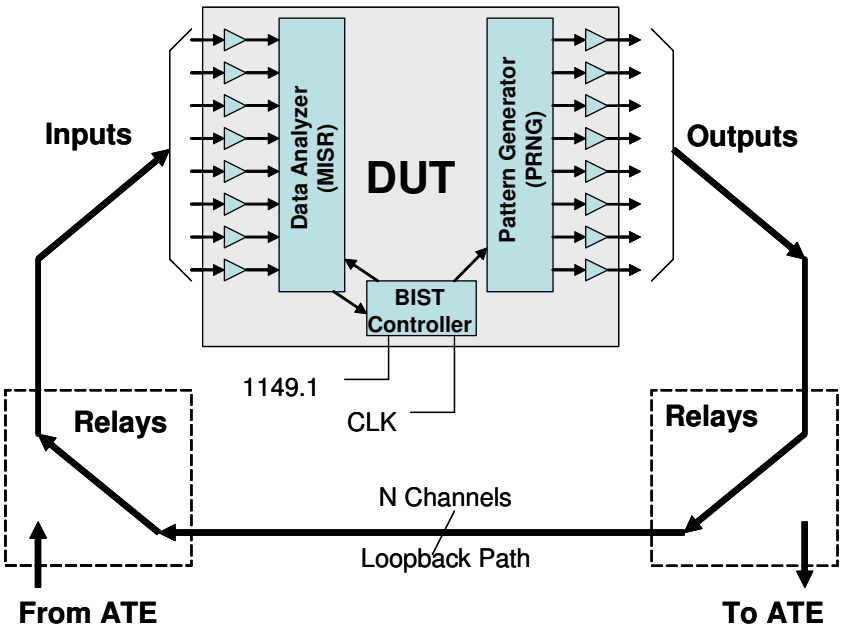


Figure 3.6. Overview of minimal loopback testing [52]

Loopback paths can be either active or passive. Passive loopback paths simply relay the signal through an external path back into the DUT. The signal may suffer attenuation travelling through longer paths and may be the test of choice for many applications. Active loopback paths generally re-clock the DUT's test signal for sharper edges. In addition to re-clocking, advanced features can be added onto active loopback cards. Depending upon application requirements, loopback cards can be designed with active logic components such as high-speed buffers, jitter manipulation, skew adjustment, amplitude adjustment, etc. Both types of paths are demonstrated in [52] & [53]. Figure 3.7 shows the output from an active loopback path with an input signal running at 10.0Gbps. Jitter is measured at 20ps (p-p), out of which only 6ps is attributed to the loopback path. The data eyes are open with 0.80UI, thus producing usable output data.

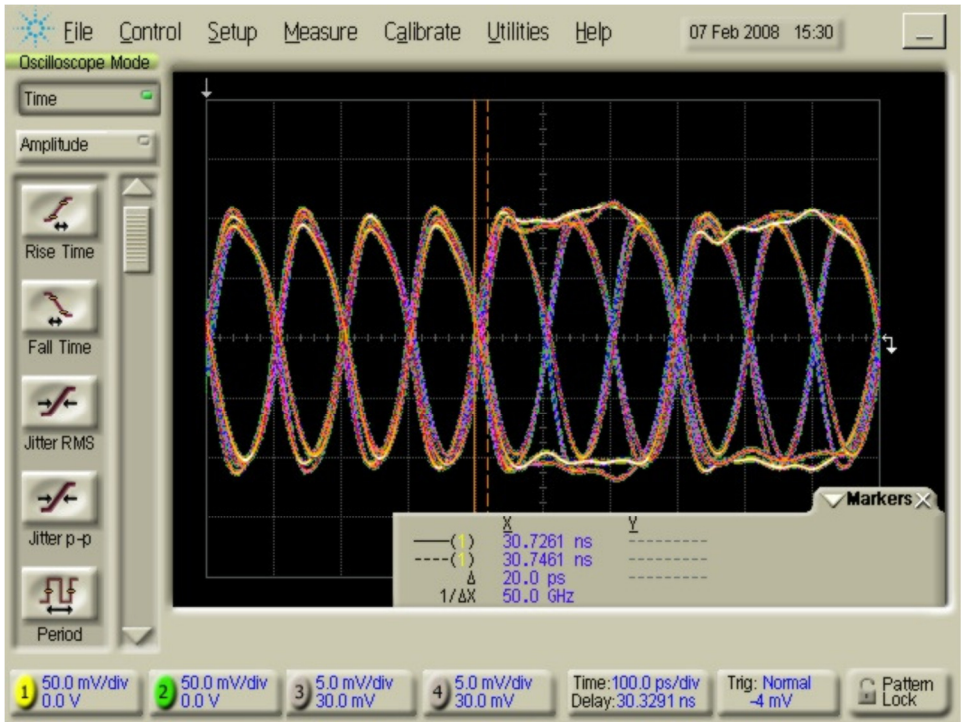


Figure 3.7 Active loopback path measured at 10.0Gbps [53]

3.3 Timing Skew adjustment

The ability to adjust the relative timing or phase between signals is often desired during testing. Timing adjustment can be used to ensure the consistent sampling of test signals. For example the phase of an ATE's sampling clock can be adjusted such that it samples at the center of an input signals' bit period, thus producing more reliable results. Most ATE have some form of timing adjustment available, such as the Teradyne Tiger offers timing adjustment capabilities with a resolution of 100ps [18]. Considering a 10Gbps signal whose bit period is only 100ps, this resolution is inadequate. Furthermore, when dealing with parallel data signals, it is necessary to ensure that all signals arrive at the destination at the same time. Multi-GHz signals highlight this necessity as the slightest mismatch in path lengths can cause timing misalignment between signals. Therefore, there is a need for timing adjustment on a finer picosecond scale.

Adjusting the phase of a constant-frequency clock signal for sampling purposes or to generate high-speed data is generally simpler than adjusting the phase of the incoming data signal itself. Many techniques utilizing VCOs and PLLs/DLLs have demonstrated this functionality [54]-[58]. However, timing adjustment is much more challenging when the problem at hand is to align multiple multi-gigabit data signals to arrive synchronously at the DUT. In [59], a method for adjusting the phase of multi-GHz test signals using stages of variable gain buffers is demonstrated on signals running up to 6.4Gbps. The variable gain buffers used are off-the-shelf buffers that contain a V_{CTRL} input pin. The pin accepts input voltages ranging from 0-3.3V in order to determine the amplitude of the buffer's output signal. Essentially, by increasing the output voltage amplitude, the signals rise time is increased. This in effect causes a delay in the signal, thus by varying the

output amplitude, delay can be finely varied. A stage of the delay circuit is shown in Figure 3.8 consisting of a buffer with an adjustable V_{CTRL} pin. An output stage buffer is shown after the adjustable buffer. The purpose of the second buffer is to recover the full amplitude of the signal.

The research done in [59] shows that one buffer allows a range of up to 10ps of delay. Although the ability to finely adjust the phase of a high-speed signal is an achievement, a 10ps range may not be sufficient for many applications. Therefore the paper presents a 4-stage fine delay adjustment which allows a theoretical range of 40ps (4 adjustable buffer offer 10x4 = 40ps). In order to allow a larger range of delay adjustment, a finite delay circuit is used in conjunction with the 4-stage fine delay adjustment circuit as shown in Figure 3.9. Discrete delay is added to the signal by using a multiplexor as shown in the figure, to select between 4 signal paths. Each path varies in length and is designed to add a discrete amount of delay to the signal, in this case 0ps, 33ps, 66ps, and 99ps. A fan-out buffer provides the input signal onto each of the paths, and the path with the desired amount of delay is selected.

The module developed in [59] is demonstrated to have a range of 95ps and the fine-delay stage demonstrated with a range of 50ps. Figure 3.10 shows the delay circuit with an input clock signal at 6.4 GHz. The circuit is used to delay the signal 32.5ps. Total jitter of the output signal is measured to be 10.5ps. Since the fine delay range is greater than the finite delay steps, this module can be used on ATE test signals to provide a fine delay range of up to 145ps, while adding minimal jitter to the signal.

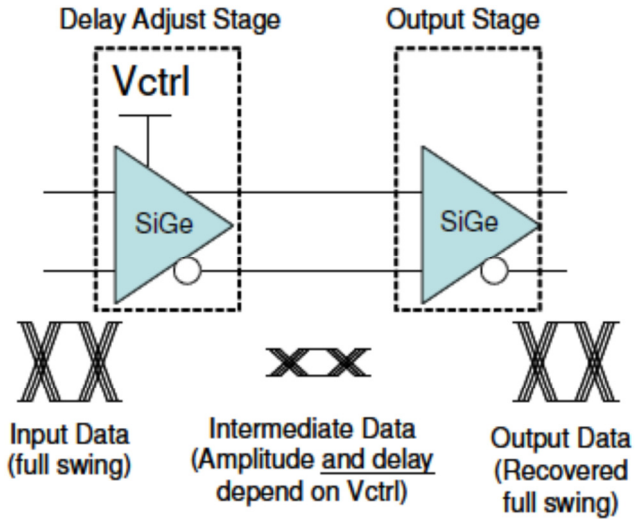


Figure 3.8 One stage variable delay circuit [59]

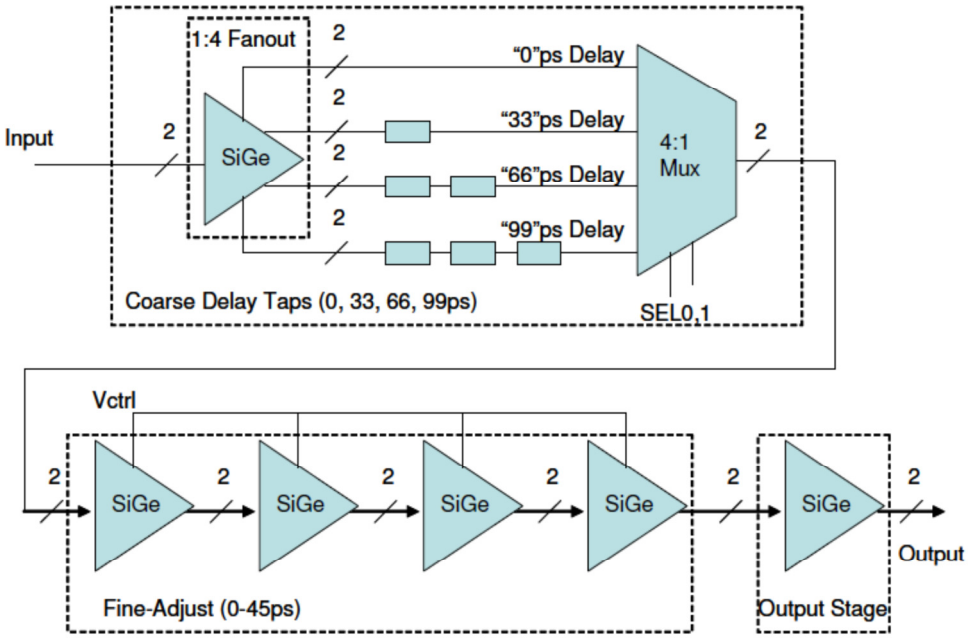


Figure 3.9 4-stage fine delay circuit with finite delay circuit [59]

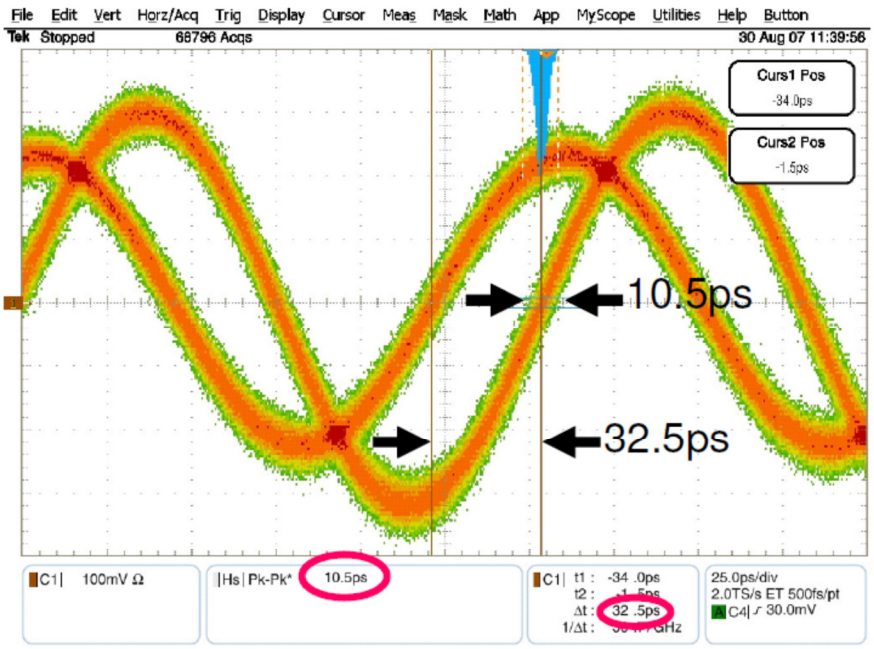


Figure 3.10 Output from delay circuit at 6.0 GHz showing a delay of 32.5ps [59]

3.4 Jitter Manipulation

Jitter is a critical issue when dealing with multi-GHz systems. Jitter is defined as "short-term non-cumulative variations of the significant instants of a digital signal from their ideal positions in time" [60]. Jitter on the order of a few tens of picoseconds can pose to be a challenge on high-speed signals. For example on a 8Gbps signal – or 125ps bit period, p-p jitter of only 25ps can reduce the open eye to 0.80UI. Therefore it is imperative to be able to control jitter on the picosecond scale on test support cards.

Adding more features to a system generally increases jitter. Therefore logic used to improve jitter can in fact add jitter to the system. This makes jitter improvement quite challenging, and it cannot be eliminated completely – especially when dealing with complex multi-GHz. However, in [61] a novel method for jitter reduction is introduced.

In this method, real time averaging is performed on a pair of identical signals. In theory this reduces the total jitter of the signal. Results presented in [61] agree with theoretical calculations. A simple circuit used to average high-speed signals in real time is shown in Figure 3.11.

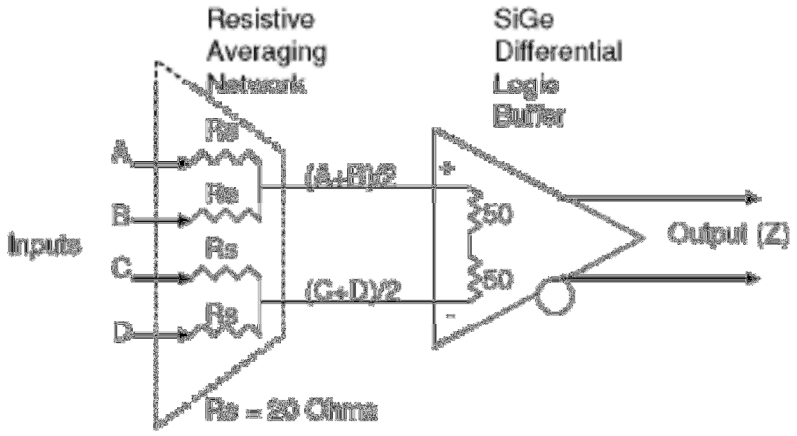


Figure 3.11. Simple jitter reduction circuit [61]

In addition to jitter reduction, many test applications require jitter injection. For example stressing test input jitter tolerance may require controlled jitter injection. Injecting jitter in a controlled and feasible manner is also a challenge especially when very small (picosecond) finite increments are required. In [62] & [63], a method of jitter injection employing passive filters on SerDes I/Os on high-speed communication devices is demonstrated. In another approach [64], Shimanouchi introduces periodic jitter injection for SerDes. However injecting jitter in finite controlled amounts still remains challenging.

In [59], a new technique for injecting jitter into a system using variable output buffers is presented. The jitter injection function is an offshoot of a timing adjustment

function discussed in the section above. The variable output buffers are primarily designed to vary the output amplitude of the signal, however in doing so, cause timing skew on the output signal as shown in Figure 3.8. Adding AC coupled noise on the V_{CTRL} input essentially randomly skews the delay of the output signal, thus causing jitter. The amount of jitter injected is a function of the amplitude of noise applied. A higher voltage noise causes longer random timing skew, which translates to higher injected jitter. Characterizing this relationship allows for controlled jitter injection onto the signal.

Figure 3.12 and Figure 3.13 show the operation of this circuit at 3.2Gbps. Figure 3.12 is an eye diagram of the input signal to the jitter injection circuit with jitter measured at 28ps and Figure 3.13 is the output signal. Here, jitter is measured at 69ps total jitter, showing the circuit adding 41ps of jitter onto the input signal. This new method is a cost-effective method for injecting jitter and can be easily incorporated into modular extension cards for ATE.

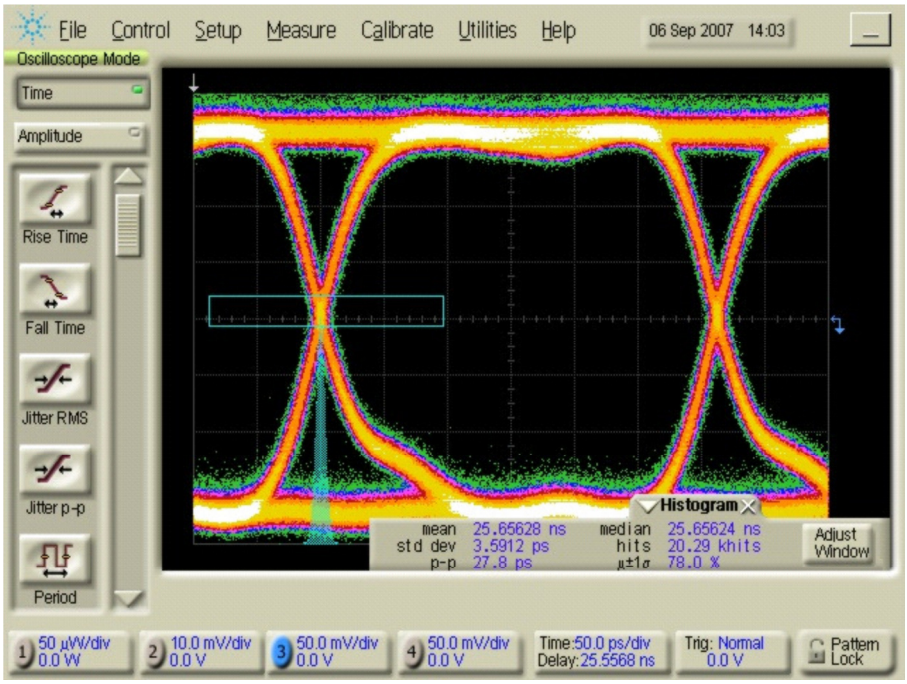


Figure 3.12 Input signal into jitter injection circuit at 3.2Gbps with 29ps of total jitter [59]

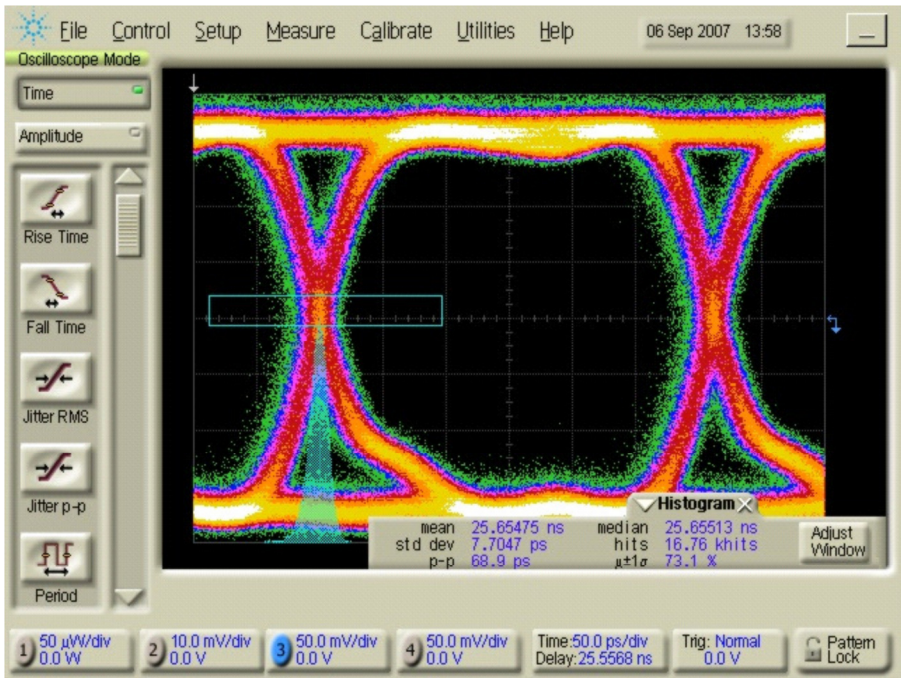


Figure 3.13 Output signal from jitter injection circuit at 3.2Gbps with 69ps of total jitter [59]

3.5 Switching

The modules to extend ATE performance discussed in Sections 3.1 and 3.2 incorporated the use of relay switches. In many test applications, in addition to high-speed tests, the DUT may require other low-speed tests such as parametric testing, continuity checks, stress testing, etc. Generally, most ATE have numerous features and are better equipped to perform such tests. Testing time on an ATE can be very expensive in a production environment. Switching between testing using modular extension cards and testing using the ATE may not be feasible in many cases. Therefore it is ideal to design modular extension cards that allow direct connections between the DUT and ATE. To accommodate this, the modules presented were equipped with relay switches. This allows the extension module to perform the required high-performance tests and also the ATE to perform any additional tests as necessary.

Generally a relay would be the last component a signal passes through before being input to a DUT or the first component the output signal passes through on an extension module. Furthermore relays are used on extension modules with other functionalities such as high-speed signal generation, loopback testing, etc. as shown in [44] & [52]. The function of the relay is similar to the function of a multiplexor in that it selects either to pass a signal from the extension module or from the ATE. Although this may be a simple task, when dealing with multi-GHz signals, the relay used must be chosen with care to support such high-speed signals.

In [52], a mechanical RF relay capable of multi-GHz signals is demonstrated. The switch is a reed type relay and simply consists of two input signals and one output signal

along with two voltage terminals. When a prescribed voltage is placed across the voltage terminals, a mechanical switch closes the path to select one input signal as the out signal – this is used as the high-speed path. In a normal state, when there is no voltage differential across the pins, the mechanical switch closes the circuit on the other input which is used for the slow-speed path. Therefore, when the switch is off, i.e. no voltage applied across the terminals, it passes ATE signals directly to the DUT. When turned on, the extension module is active and provides the signals to the DUT. Figure 3.14 shows the performance of this switch on an extension module. The output signal of a 3.2Gbps input signal measures a total jitter of 32ps, adding 11ps of jitter to the signal. At 5.0Gbps, the measurement taken shows an addition of 10ps of total jitter, however it can be noticed that at this speed, the rise time of the switch is not fast enough to achieve full amplitude swing. Regardless, the switch demonstrated sufficient performance for testing applications.

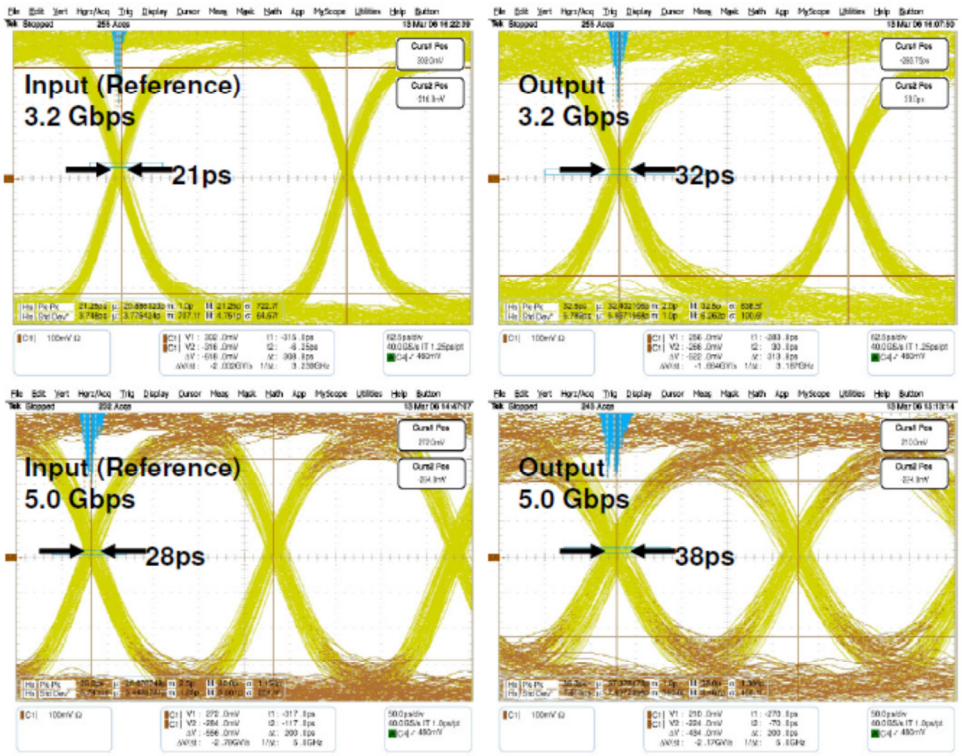


Figure 3.14 Mechanical relay performance [52]

The size of a mechanical relay package measures approximately 0.4 inches and is shown in Figure 3.15. These switches are relatively large and can consume too much board space on an extension module which is limited in size. Thus, when multiple test signals are required on an extension module, this size can be a limiting factor. Also shown in Figure 3.15 is a similar function MEMS based relay switch which is approximately half the mechanical relay's size. This MEMS based switch utilizes a device-on-package construction which fabricates the MEMS device directly onto its ceramic (alumina) wafer via conductive metal vias, thus allowing a much smaller form factor [52]. Furthermore, the use of an alumina substrate and minimizing the path between the device and printed surface by conductive vias, reduces insertion loss, thus

allowing better propagation of high-frequency signals [65]-[66]. These factors make selecting a MEMS based switch ideal when multiple high-speed signals are required on an extension module.

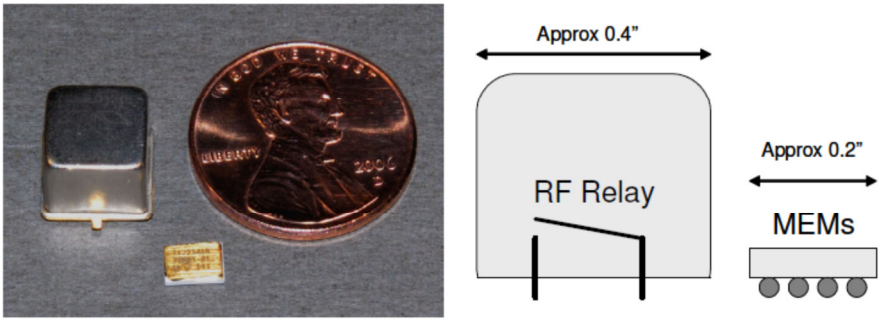


Figure 3.15 Size comparison of mechanical relay and MEMS relay [52]

In [52], an extension module using MEMS based switches is presented. Performance of this module is shown in Figure 3.16. At 3.2Gbps, the module adds 6ps of total jitter, and performs better than the mechanical relay switch. At 5.0Gbps, the module is demonstrated to add 13ps of total jitter on to the input signal. This performance is comparable to the performance of the mechanical relay shown at 5.0Gbps. However the power dissipation characteristics were measured to be much lower on the MEMS switch compared to the mechanical switch [52].

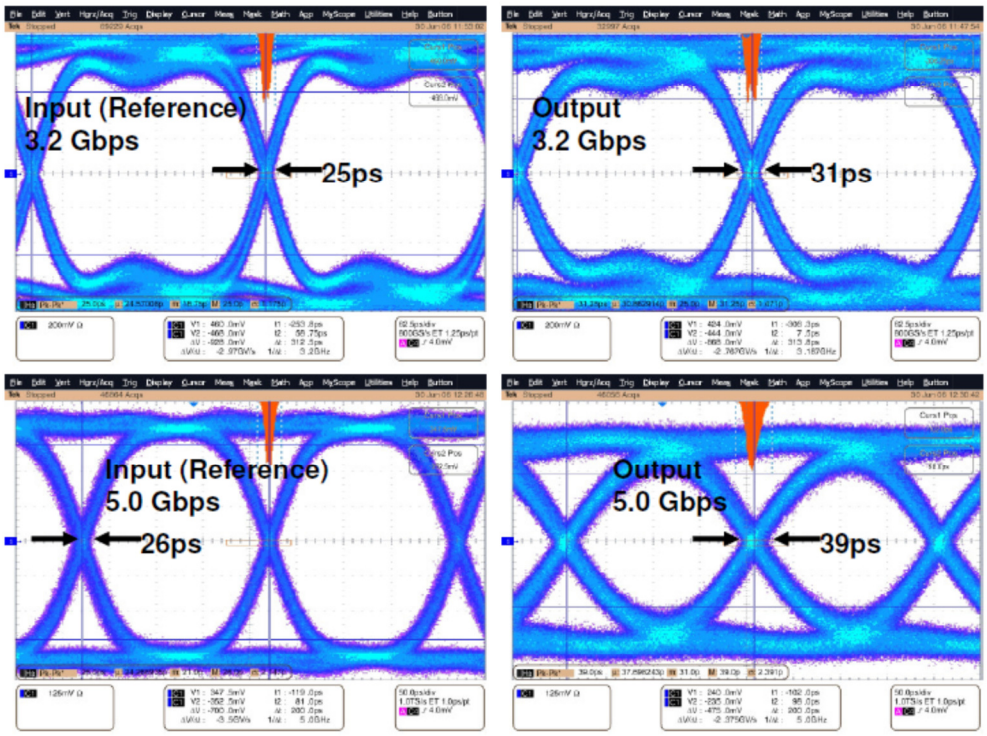


Figure 3.16 MEMS switch performance [52]

In this chapter previous research is shown to demonstrate modular extension of ATE targeting five specific applications – high speed signal generation, loopback testing, timing adjustment, jitter manipulation and switching. All of the applications shown are capable of multi-GHz speeds, although some were not demonstrated at speeds above 5Gbps. Despite this, the principles used are still valid for extending ATE performance. By slightly modifying the methods used greater performance can be achieved. In the next chapter, a different approach to extending ATE performance is discussed, in which stand-alone mini-testers are developed. The principles learned in the research discussed in this chapter helped the development of the systems in the next chapter.

CHAPTER 4

STAND-ALONE MINIATURE TESTER

In Chapter 2, methods for enhancing test capabilities are presented consisting of modular electronic cards used in conjunction with ATE. The modular cards use resources from the ATE along with on-card electronics to produce higher performance test capabilities. This method of enhancing test capabilities is limited by the performance of available ATE resources. Some test applications may require test capabilities not achievable using these ATE resources alone. For example, consider the case when a 10Gbps test signal is required for testing purposes. A modular extension card capable of doubling data rates would require 5Gbps input signals to produce this rate. Performance of ATE not capable of producing required input rates cannot be enhanced to target rates. Therefore modular enhancement methods may not be feasible solutions for certain test requirements.

Another method of enhancing test capabilities is to develop a custom test system capable of operating independently of an ATE. This method of enhancing test capabilities involves designing custom test electronics that do not rely upon ATE resources. Early work to extend testing capabilities to the GHz range using this method ranged from developing custom test heads for existing ATE [39]-[40] to developing proprietary test systems [38] & [67]. Developing proprietary test systems to rival current ATE is not feasible because ATE have become exceedingly complex and extremely costly to develop. However custom test systems designed address specific applications with limited test functions can be used to enhance testing capabilities.

In this chapter a custom test system specifically designed to test wafer-level package (WLP) devices is presented. Various testing functions can be addressed with this test system; however the particular test system presented is intended to enhance high-speed testing capabilities. The test system is designed with an FPGA that allows it to operate independently of an ATE, and additional logic to produce high-speed testing capabilities. Before the details of this test system are presented, wafer-level testing is briefly discussed to introduce the probing technologies the mini-tester is designed to work with. Subsequently, the design of the mini-tester is discussed in detail. An experimental demonstration and performance characteristics of the test system are presented in the final section.

4.1 Wafer-Level Testing

Miniaturization of portable hand-held electronic devices has stimulated the need for IC packages of even smaller size than conventional ball grid array (BGA) and chip scale packages (CSPs). This has led to the development of advanced packaging technologies such as WLP, MCM, SiP, SoC, etc. These new packaging technologies have allowed for smaller device sizes and more integration.

A WLP is a chip size package. The area that it occupies when mounted onto a system level board is as small as the size of the IC itself. WLP devices offer minimum size and weight for a given die, and cost is also expected to be lower than for traditional IC packaging [68]. WLP technology has been developed with a density of up to 12,000 leads/cm2 [69] [70]. Many of these leads are for power and ground, thereby reducing the number of signal pins to only a subset of this number. Despite the reduced number of

signal pins, many problems still arise in testing wafer-level packaged devices due to their high I/O densities, microscopic pad sizes, and high-speed testing requirements.

As package sizes have become smaller, more functionality continues to be demanded from devices. For example, consumers now want a mobile phone not only to be able to make phone calls, but also function as a camera, music device, GPS, etc. This may require multiple technology components such as MEMS, optical components, RF components, digital components, etc. to be integrated into one device. Such levels of integration and complexity exacerbate the testing problem.

To address the problem of testing complex WLP devices, a miniature tester has been developed that can be customized to the testing needs of a DUT [71]. These "mini-testers" can be designed for specific testing purposes such as high-speed testing, loopback testing, jitter injection, etc. The goal is to keep the mini-testers simple and small enough that they can be used to test devices in wafer form, yet provide easily customizable performance not available on general-purpose ATE. Additionally, these mini-testers are designed to function independent of ATE, thus requiring no (or very little) expensive ATE resources.

This mini-tester requires the use of wafer-level probers (interposers) in order to make physical contact with the device when it is still in die or wafer form. Generally the mini-tester is designed independently and then used with an appropriate interposer based upon the type of packaging technology or compliant leads present on the device. This solution allows a variety of devices with high-density leads to be tested. Furthermore, this approach can allow device components to be tested individually. For example the digital components of a device can be tested as they are fabricated, before they are packaged or

interconnected with other components. This is a relatively simpler task than testing its functionality within a fully integrated and packaged device. Undertaking such an approach reduces the complexity of testing the complete integrated device as a whole. This approach can also reduce test time and overall manufacturing cost as defective devices identified early in the fabrication process can be removed from further fabrication and packaging processes.

4.2 Wafer-Level Probing

Several different WLP technologies are available, including Bed of Nails (BoN) [72], Multiple copper columns (MCC) [73] Solder Bumps (SB), Stretched solder column (SSC) [74], and Sea of Leads (SoL) [69]-[70]. The fabrication processes for each of these technologies have varying degrees of cost and difficulty associated with them. The SoL approach uses low-cost lithographic fabrication of very high density compliant leads [69]. The SoL process is actually a continuation of the IC fabrication process and may be done without adding significant cost to the device [75]-[76].

In [71], the mini-tester is successfully demonstrated with BoN structures. The BoN is a novel compliant interconnect structure with limited z-axis compliance. To enhance this limitation, a compliant interposer is used and discussed in the next section. Since a higher column height would result in higher compliance, lower stress, and hence, longer fatigue life, interconnections as high as 50um are developed.

Single layer BoN wafer level interconnects have been successfully fabricated with a nail height of 50μm. These are designed to fulfill the following electrical requirements: DC resistance $\leq$ 25mΩ, inductance $\leq$ 50pH and capacitance $\leq$ 10-15fF. These

requirements were based on the fact that the package using such interconnects must support high frequency performance applications (e.g., microprocessors, high pin count logic devices, etc.).

The BoN fabrication process flow consisting of six steps is illustrated in Figure 4.1. Metal layers of Ti/Ni/Au are first sputter deposited onto the WLP pads. A thick photo-resist is then applied and patterned using Ti/Ni/Au etching one by one and resist removal. Secondly, BCB dielectric polyimide is spun to passivate the daisy chains and pattern the dielectric layer using UV lithography to open the pads. A Ti/Cu or Ti/Au seed layer is then sputtered. The bottom Ti layer is applied to improve the adhesion between dielectric and Cu or Au. Then a thick photo-resist is spun, soft-baked, UV patterned, and developed. The copper post is electroplated. Solder is then electroplated at the tip of the copper post for bump formation. Thick photo-resist is then removed and Ti/Cu or Ti/Au seed layer is etched away to complete the interconnect structure. Finally, solder is reflowed in N_2 atmosphere. This fabrication process is based on photolithography and electroplating processes which are compatible with the conventional IC fabrication and the fabrication is integrated into wafer-level processing as batch process [77]. Additional masks are not needed as the UBM mask can be used to pattern the photo resist for copper column deposition.

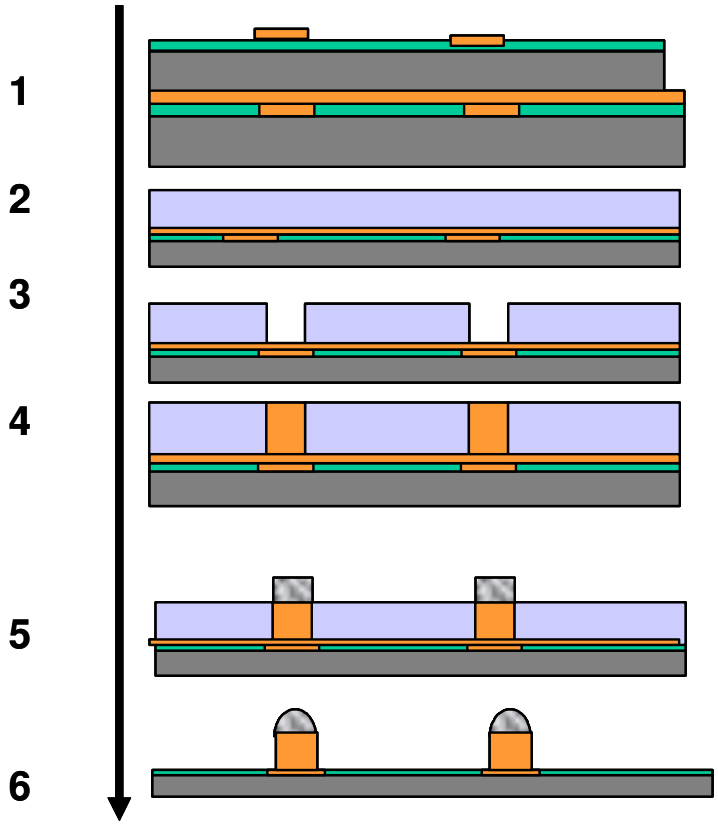


Figure 4.1 Process flow of bed of nails wafer level interconnects by photo resist method

The mini-tester does not make direct contact with the DUT, but requires a wafer-level prober such as an interposer to make physical contact with the device. This allows mini-testers to be generically designed, and then use customized interposers to make contact for actual testing. This parallel development approach greatly enhances the usability of the mini-tester, since it can be used with almost any WLP technology as long as an interposer to connect to it is available. In the next section, a few notable interposers are described.

4.2.1 Interposer

An interposer is similar to a probe card and serves as an electromechanical interface between the DUT and the mini-tester. A major role of the interposer is to serve

as a space transformer. To accomplish this, the interposer has large pitch metal bump pads on one surface to connect to the tester and fine pitch compliant interconnects on the opposite surface to make contact with the WLP device.

There are currently a few approaches used to construct an interposer. A MEMS based interposer, is proposed in [78] and shown in Figure 4.2. Another approach uses MEMS based spring probes, as discussed in [79], to make electrical contacts with the DUT. The spring interposer allows for a non-destructive low resistance contact with the wafer leads [79]. FormFactor, Inc. uses a proprietary technology called MicroSprings™ [80] optimized to provide low contact force and low contact resistance with over 900,000 touchdown rates [81].

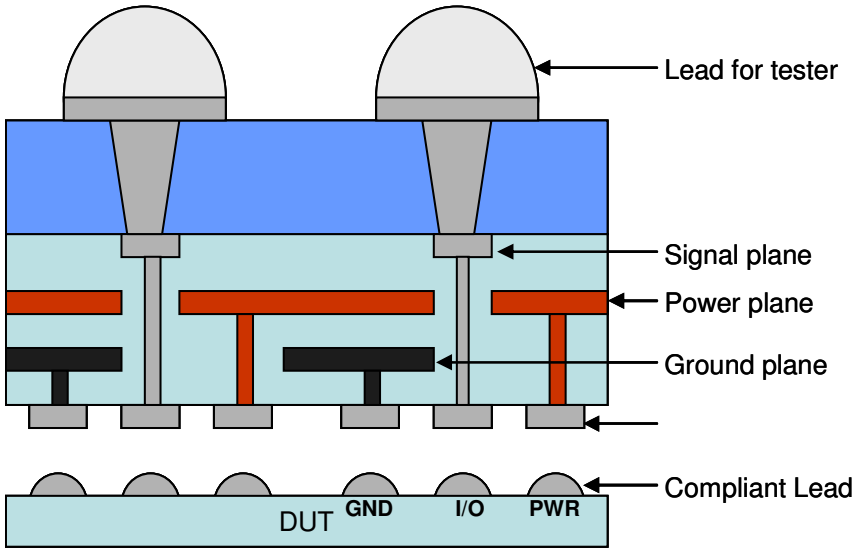


Figure 4.2 Interposer incorporating vertically connected signal traces created using through wafer vias.

Mini-testers may be connected to an interposer to make electrical contact with the wafer [82]. As illustrated in Figure 4.3, an interposer is used to redistribute the high-density WLP signals to a macroscopic scale (similar to a micro-BGA). In the figure, a

customized version of the mini-tester is illustrated as a self-contained module mounted to the top side of the interposer. The interposer has a contact interface that is designed to make contact with the appropriate complaint leads on the wafer. This allows testing to take place, if necessary, directly on the wafer chuck and even in-between fabrication steps. The mini-tester shown does not require an ATE or its resources to perform testing. Due to its stand-alone nature, connections to the miniature tester are limited to: DC power, USB, and a high-performance (low-jitter, multi-GHz) clock input.

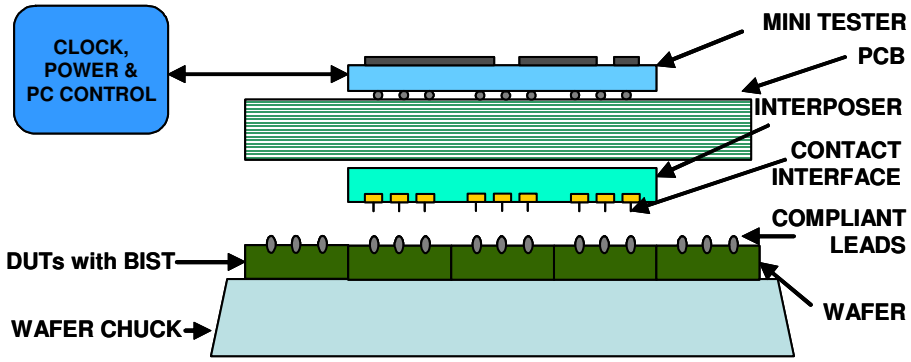


Figure 4.3 Testing of wafer-level packaged (WLP) devices using a "miniature tester" and a high-density interposer.

Using similar concepts as described above, the mini-tester may be used to perform parallel testing. When parallel testing is required, the miniature tester may be replicated in array form as illustrated in Figure 4.4. The complexity of the PCB is minimized by using only a small number of signals for each mini-tester, thereby taking advantage of BIST features of the DUT. This strategy is a logical extension of existing parallel tests (such as used in memory testing) employing highly aggressive WLP testing techniques [83]. As testing is done on multiple devices at the wafer level, manufacturers gain significant cost and time savings.

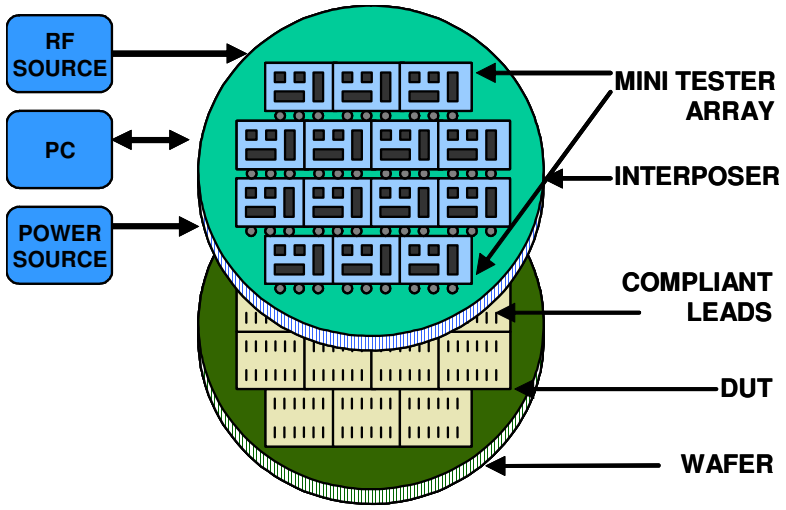


Figure 4.4 Parallel high-speed wafer probing using multiple miniature testers.

In [71], wafer-level probing is done by cantilever probe needles in a bare-die test socket. For testing purposes, signals from the mini-tester are relayed to the bare-die test socket, which then probes the DUT using cantilever probe needles to deliver the test signals. The probing method is discussed in the following section.

4.2.2 Bare-die test Socket

Cantilever probe needles [84] have traditionally been used for testing wafer level devices. They are mostly useful for frequency applications below 100MHz due to long lead inductances. Coaxial probes [85] are available with multi-GHz performance for pad pitches as small as 120 micron and have been used for probing solder bumps. However, coaxial probes do not scale well for higher I/O density, fine pitch packages.

In [77], a novel approach based on metallized elastomer mesh which meets the small size, high frequency and compliance requirements of WLP is described. The elastomer mesh has coplanar contact probes that make the actual contact between it and the DUT as shown in Figure 4.5. Each probe location consists of three fingers that

correspond to ground-signal-ground (GSG) placed at a pitch of 100 microns. Gold plated metallization is used for the signal and ground contacts. The metallization lines are screen printed on the elastomer in the form of tapered GSG probes. The thickness of the mesh is 50 microns. The probes make contact with BoN copper column interconnects on the DUT.

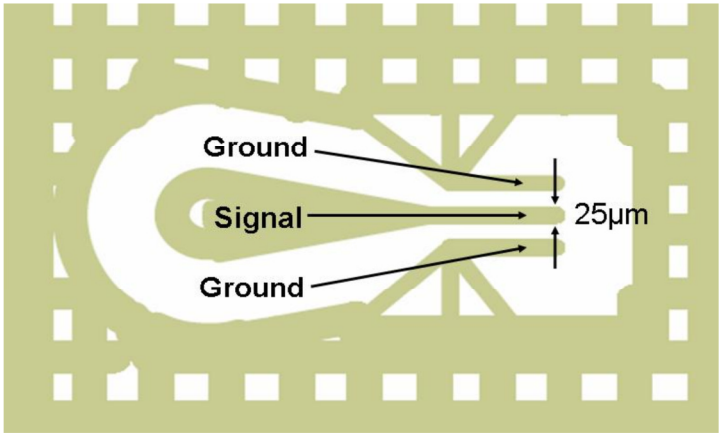


Figure 4.5 Layout of elastomer coplanar contact probe.

A prototype of the test fixture is shown in Figure 4.6 & Figure 4.7. The design consists of two parts. First an elastomer mesh provides an electrical and mechanical interface to the WLP. The elastomer material is itself in mesh form. Metal lines are screen printed on the mesh on both sides. The signal wiring pitch is 100 micron (not discernable in the figure). The compliance achieved depends on the mesh thickness. For the prototype used, the compliance is on the order of 2 µm/gF. Secondly a multilayer PCB substrate made of BT resin material provides support for 3.5mm SMA connectors for use up to 6 GHz. One side of the mesh connects to the device under test while the opposite side makes contact with the PCB. The PCB connects to the mini-tester through the SMA connector.

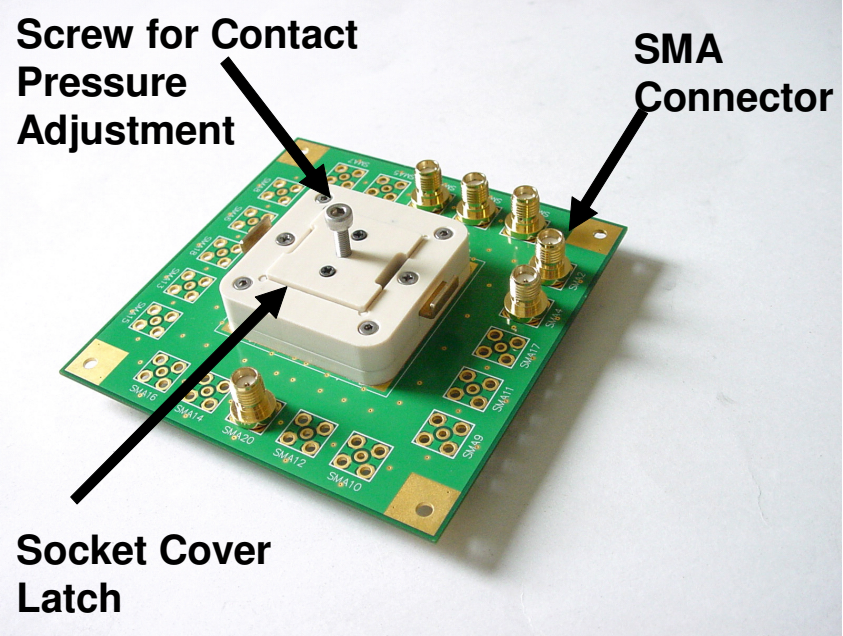


Figure 4.6. Prototype Test Socket

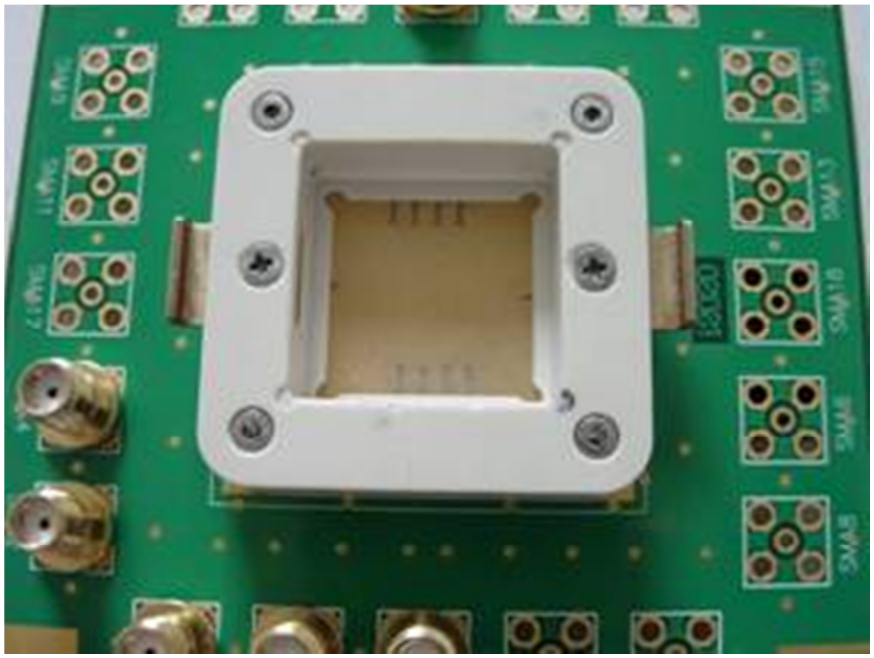


Figure 4.7. Elastomer Coplanar Contact probes inside test socket.

The multilayer PCB has four metal layers, two of which form signal trace layers on either side of the board with 50Ω transmission lines and the other two are buried layers used as ground planes. The PCB serves as a space transformer between the fine pitch WLP (at the 100 micron level) and the instrumentation connectors (millimeter scale).

The contact probes are made by screen printing metallization lines onto the elastomer mesh. Some areas of the metallization are patterned to from the ground and power grids, while others provide short signal traces and contact pads forms in the sparse elastomer matrix. The contacts can be densely populated to test fine pitch, high I/O density WLP devices. The elastomer mesh material has spaces into which metallization can be plated. This arrangement contributes towards the necessary compliance while maintaining low contact resistance.

A diced WLP device is placed inside the socket on the test hardware. Connectors surrounding the WLP device are connected to measurement instruments. The coplanar transmission line on the substrate printed circuit board and the probe on the mesh provide efficient high frequency transmission.

4.3 Miniature tester

The concept of a "test support processor" (TSP) was introduced earlier [82] and is the precursor of the mini-tester. A TSP is a customized circuit that can be included in a testing environment to enhance an ATE's performance or functional capabilities. Customizing the TSP to the DUT's testing needs could be done quickly by reusing a core logic structure called the Digital Logic Core (DLC - discussed in the next section) and

adding specialized components. In this way the TSP could rapidly and economically address new test requirements. However, the TSP was technically still a “support” processor, i.e. it used significant resources from the ATE in order to create specialized test signals. This limited TSP performance, as many of its functions were limited by performance characteristics of the ATE. Furthermore, the TSP approach was not so inexpensive, since it needed many very expensive ATE resources.

The main purpose of the DLC in the TSP was to provide control functions between the TSP and the ATE. It served as a critical component of the TSP as it acted as in interface between the TSP and ATE, and also the user. Additionally, it could be used to control peripheral logic used in the TSP as necessary. The main component of the DLC was an FPGA. As FPGA technology started to advance ahead of ATE performance in some respects, it was realized that a DLC could be designed with a state-of-the-art FPGA that could control the TSP without the need for resources of an ATE. This was achieved by extending the basic TSP by adding connections for a PC controller, RF clock source and DC power sources [86]. The DLC now controlled the TSP’s test functions via a PC through which users could enter commands. These developments set the stage for a stand-alone mini-tester to be designed.

Using this DLC, a customized mini-tester was deigned at low cost, since it did not have all of the general purpose features found in traditional ATE [87]. It was designed to provide only the specific test features needed for a particular application. Furthermore, to keep costs low, off-the-shelf chips are typically used for constructing the mini-tester. The main component of the mini-tester, the DLC is presented in detail in the next section.

4.3.1 Digital Logic Core (DLC)

The central component of the mini-tester is known as the Digital Logic Core - DLC. The main component of the DLC is an FPGA, which can be programmed to serve as a test controller. In [86], a Xilinx Virtex XCV300E FPGA was used which had over 150 available I/Os, each capable of running up to 400 Mbps. In some applications, these signals could serve directly as I/Os for testing the DUT. However, in this design, these I/O signals were supplied to additional logic to enhance performance. A top level schematic of the mini-tester is shown in Figure 4.8. In addition to the FPGA, the DLC includes a specialized microcontroller chip for interfacing to a USB. A personal computer communicates through the USB with the DLC, and provides high-level control of the tests (which otherwise are synthesized in the DLC). Supporting these are a 12 MHz crystal oscillator, and a flash memory chip to store the FPGA programming information. FPGA programming can be quickly changed by overwriting the flash. An RF clock source is also required to provide a low-jitter (picoseconds) timing reference.

In this mini-tester, signals from the FPGA are formatted and supplied to additional PECL logic. The PECL devices take the formatted signals and create multi-Gbps. State machines encoded in the FPGA, together with higher speed PECL multiplexers and sampling circuits synthesize the desired tests in real time. This process is discussed in detail in the next section.

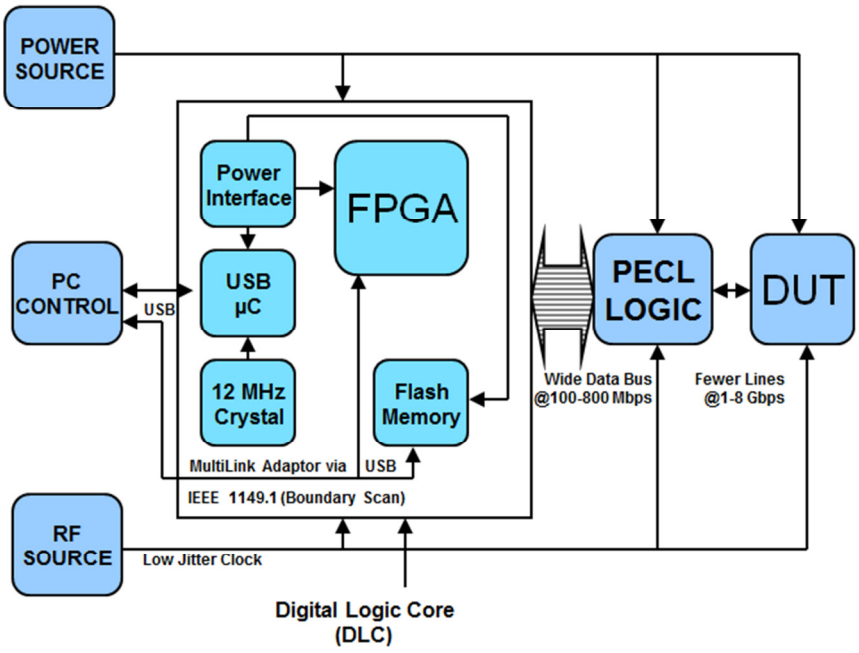


Figure 4.8. Miniature tester with high-speed PECL for testing multi-GHz DUTs (DLC enlarged).

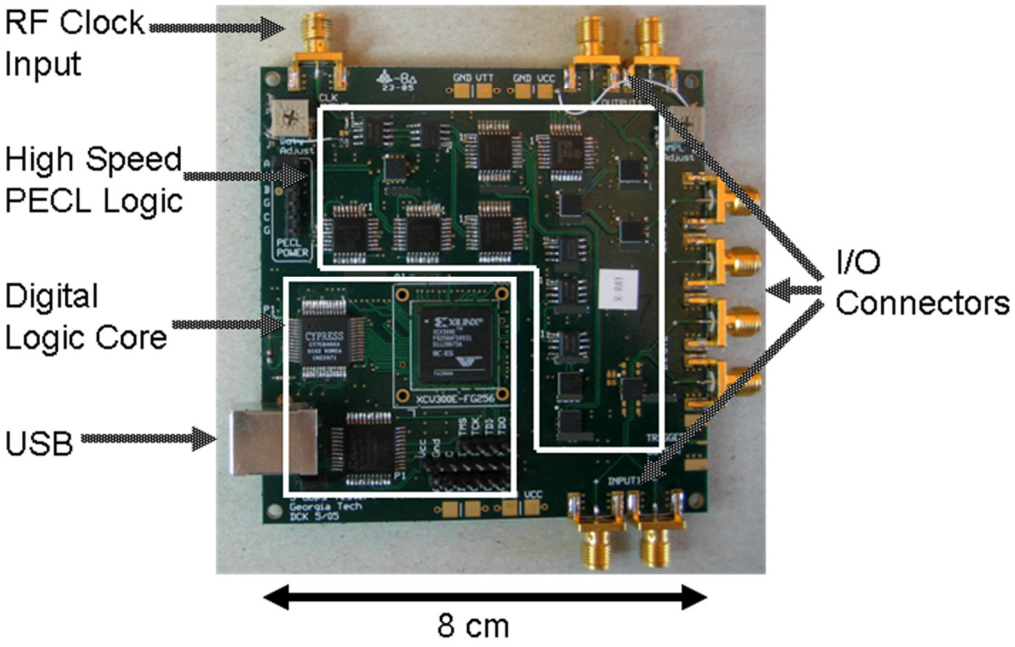


Figure 4.9. Prototype miniature tester with embedded DLC.

A photograph of the mini-tester prototype board is displayed in Figure 4.9. Although the prototype is large compare to the devices it is designed to test, this prototype can be used to aid design of an ASIC-based mini-tester, reducing its size by an order of magnitude. Each mini-tester ASICs can be integrated onto an interposer to test one die site in an array (see Figure 4.4). The tests are designed to demonstrate high-speed signal propagation through the compliant lead structures. The mini-tester produces a programmable data source up to 8.0Gbps with 10ps timing resolution (see Section 4.3.2). A high-speed PECL sampling circuit is designed to capture the returned signal, also with a 10ps resolution (see Section 4.3.3).

4.3.2 High Speed Signal Generation

The DLC frequency is limited by the chosen FPGA to about 300-400Mbps, therefore PECL is used to produce higher speed signals as shown in Figure 4.10. The high-speed signal generation logic includes a multiplexer that allows the user to choose between two input clock sources. This source clock is then fanned-out into two programmable clock delay chips, which are controlled by the DLC. Two parallel-to-serial converters are used to convert 8-bit words from the DLC into a serial data stream. Test signals are independently programmed in the DLC and sent to the individual serial converters. The clock outputs from the delay chips are used by the serial converters. These signals are then sent to the next stage where they are logically combined using an XOR gate to produce even higher speeds. This method creates a double data rate. The signal is then fanned out to four separate channels.

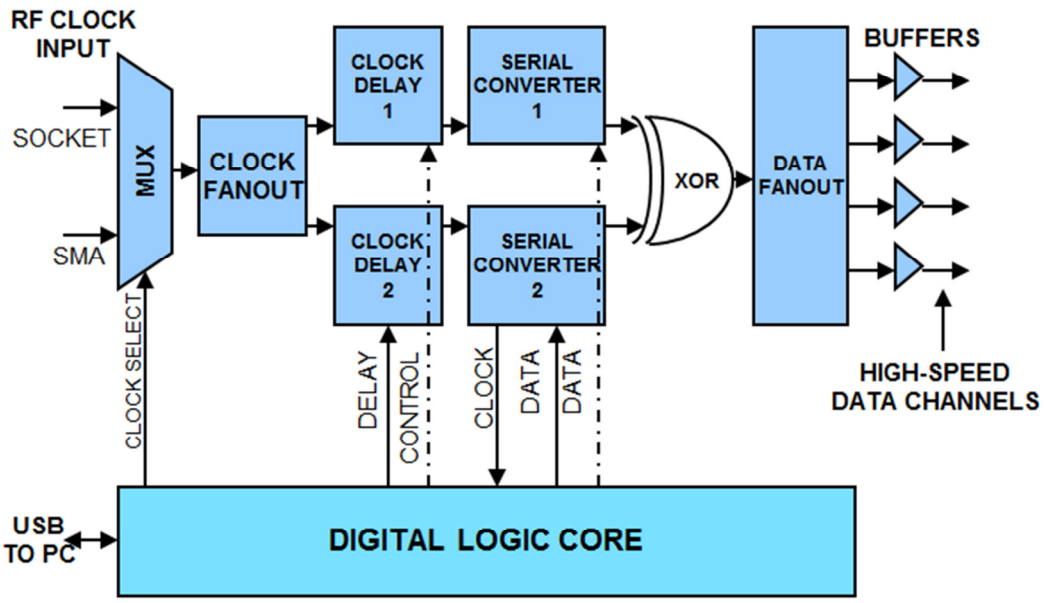


Figure 4.10. PECL logic used in mini-tester for high speed signal generation.

4.3.3 High Speed Signal Sampling

For capturing the DUT output signals, PECL devices allow the DLC to sample one bit at a time from the high-speed incoming signal. The receive side of the mini-tester, as shown in Figure 4.11, can receive up to 4 high-speed signal channels. Data capture takes place using a high-speed flip-flop that is clocked by another programmable delay chip. The delay chip is used to sweep the sampling time across the received signal. Sampled data is then accessed by the PC though a USB link and used to plot a reconstructed waveform.

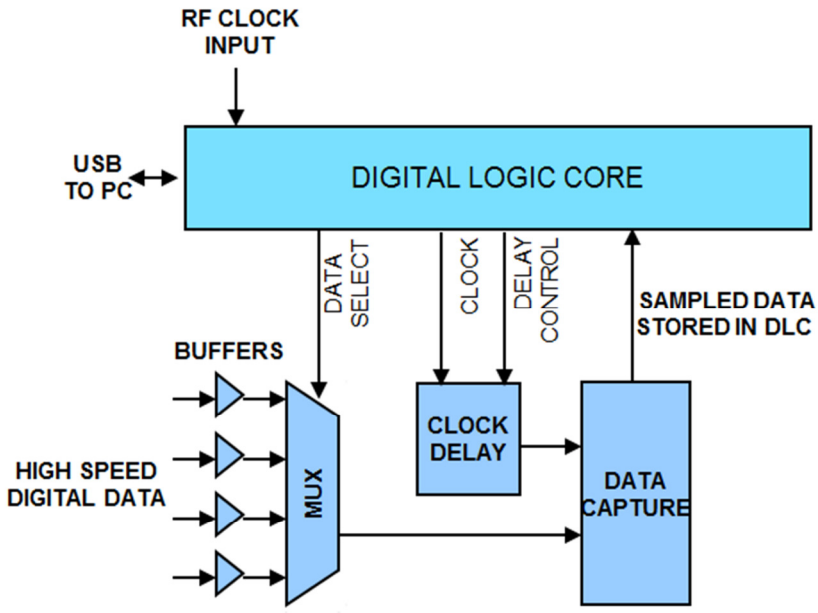


Figure 4.11. Logic used in mini-tester for high speed signal sampling.

4.4 Experimental Demonstration of the Mini-Tester

At speeds above 5Gbps the mini-tester is required to produce bit periods shorter than 200ps. Most PECL devices are not sufficient for this application as they have rise and fall times above 100ps [86]. They cannot achieve full amplitude swings when running at such high speeds. In order to accommodate higher speeds, the mini-tester was redesigned using SiGe logic devices [88]. To demonstrate the high-speed signal generation capability, a pseudo-random data pattern was generated with the mini-tester, using an LFSR encoded into the FPGA. An oscilloscope was used to measure eye diagrams from one of its output channels. Figure 4.12 shows an eye diagram at our target rate of 5.0Gbps. The eye diagram shows eye openings of about 0.8UI and sharp logic transitions.

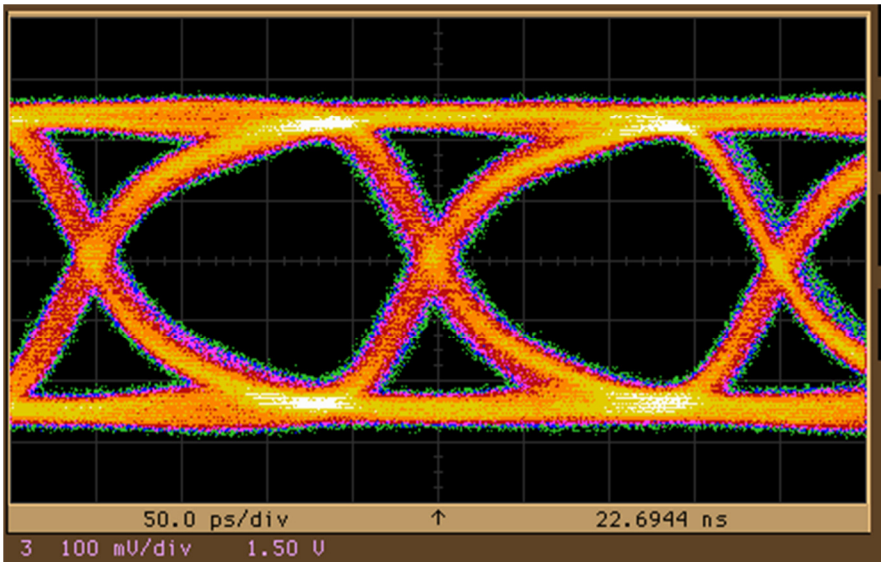


Figure 4.12. 5.0Gbps eye diagram produced by mini-tester.

Figure 4.13 shows an eye diagram at 6.4Gbps. Again open data eyes were obtained, with about 0.75 UI. Even at such high speeds, sharp transitions and full amplitude swings can be obtained using SiGe devices. Jitter was measured on the 6.4Gbps signal to be about 35ps (including 6σ random jitter and deterministic effects).

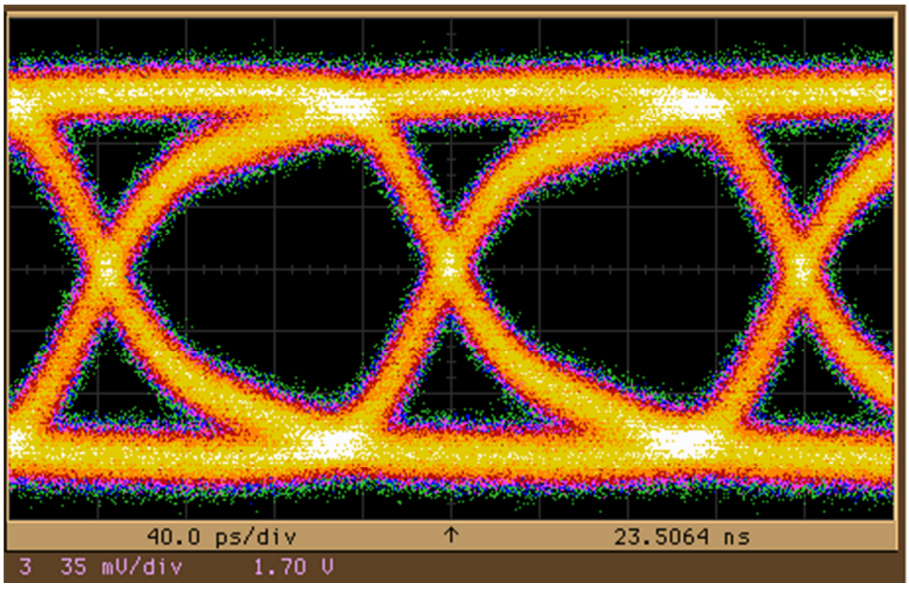


Figure 4.13. 6.4Gbps eye diagram produced by mini-tester.

The mini-tester was able to exceed its 5Gbps goal and reach speeds up to 8.0Gbps. Figure 4.14 shows an eye diagram at this speed. Signal speeds were even pushed up to 9.6Gbps, where the eyes entirely collapsed. Jitter was measured on the 8.0Gbps signal and found to be about 40ps (including 6σ). By using SiGe devices in this design, a significant reduction in p-p jitter was achieved compared with earlier designs that used standard PECL technologies. However the programmable delay chips are known to add picosecond range random and deterministic jitter [17]. Despite this fact the jitter observed was at acceptable levels, producing eye openings of 0.75UI.

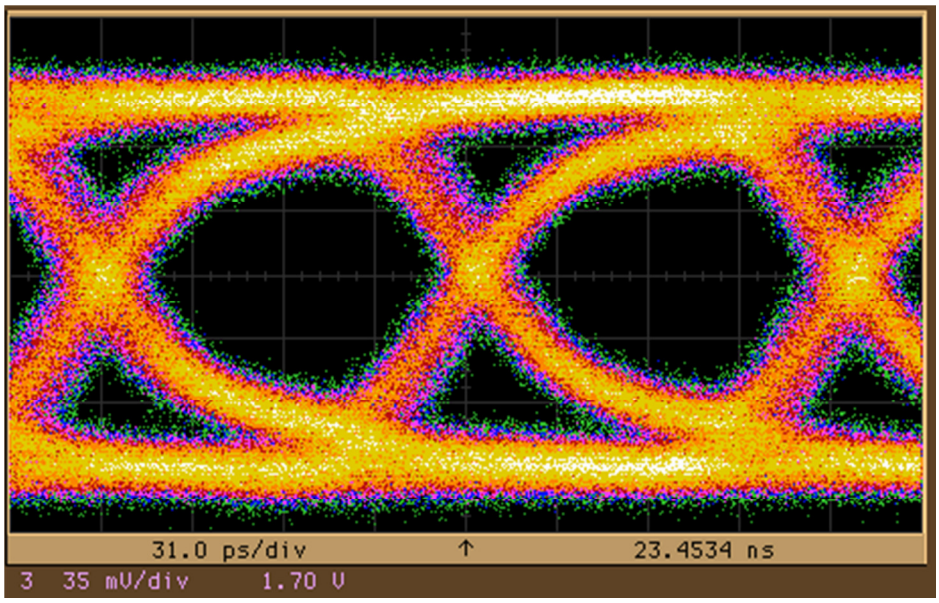


Figure 4.14. 8.0Gbps eye diagram produced by mini-tester.

To demonstrate the receive side of the mini-tester, a 6.4Gbps signal (Figure 4.15) was generated using the testers output channel and looped-back to the input channel. The DLC was then used to program the receiver sampling delay to sweep across the signal in 10ps intervals and record the values. However, the delay chip itself exhibits non-linear

behavior, which can be minimized using a previously reported calibration process [88]. Using the raw data measured by the mini-tester, a received signal waveform was reconstructed by graphing software as shown in Figure 4.16. Better accuracy can be obtained by using the calibration method described in [88].

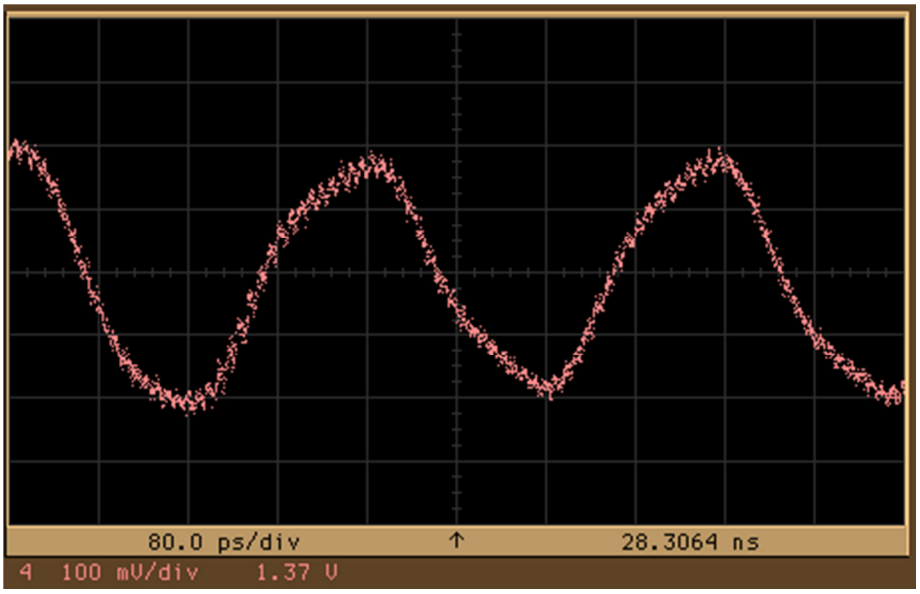


Figure 4.15. 6.4Gbps signal to be received.

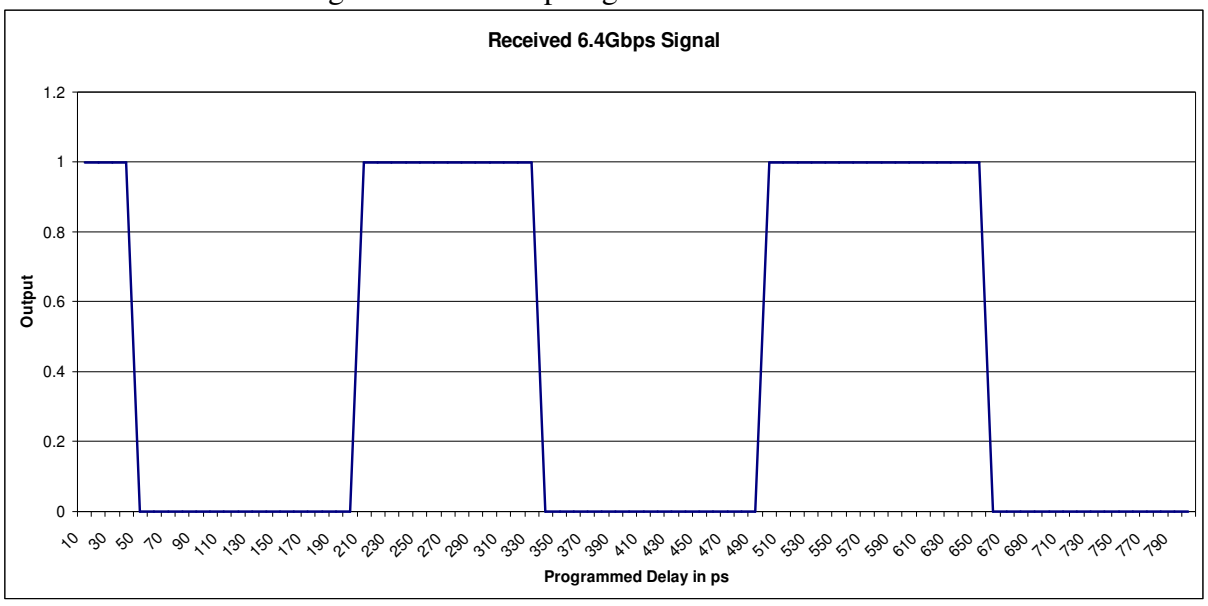


Figure 4.16. Bit pattern plotted with sampled data.

4.5 Experimental Demonstration of the Bare-die Test System

The high-speed signals from the mini-tester must pass through an interposer to test WLP devices. Signals passing through the interposer may experience some quality loss. To characterize this effect, high-speed signals from the mini-tester are passed through the interposer and a test sample DUT. One high-speed mini-tester channel was connected to the oscilloscope directly (as a reference), and its compliment was connected to the interposer. The output of the interposer/DUT was then connected to another channel of the oscilloscope. The setup is shown in Figure 4.17.

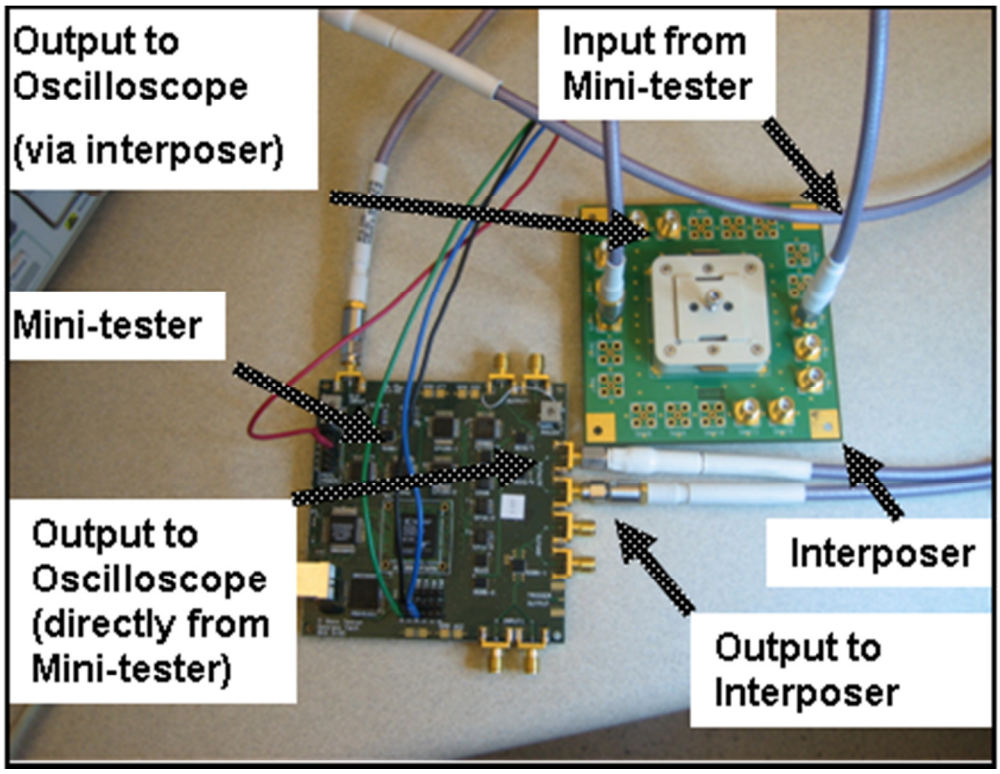


Figure 4.17. Lab setup showing high-speed signal from mini-tester prototype passing through interposer prototype to oscilloscope.

Figure 4.18 shows two eye-diagrams at 5.0Gbps, the bottom one is produced directly from the mini-tester high-speed data channel, and the top one is produced from its complementary signal passed through the interposer/DUT. From the two eye diagrams, it can be noticed that the signal from the interposer/DUT is slightly attenuated. This is due to the losses through the interposer board material, the traces on the elastomer mesh, the connectors, and the DUT itself. The high-speed signal through the interposer exhibits an increase in jitter. The jitter is at acceptable levels, as the eyes are still open.

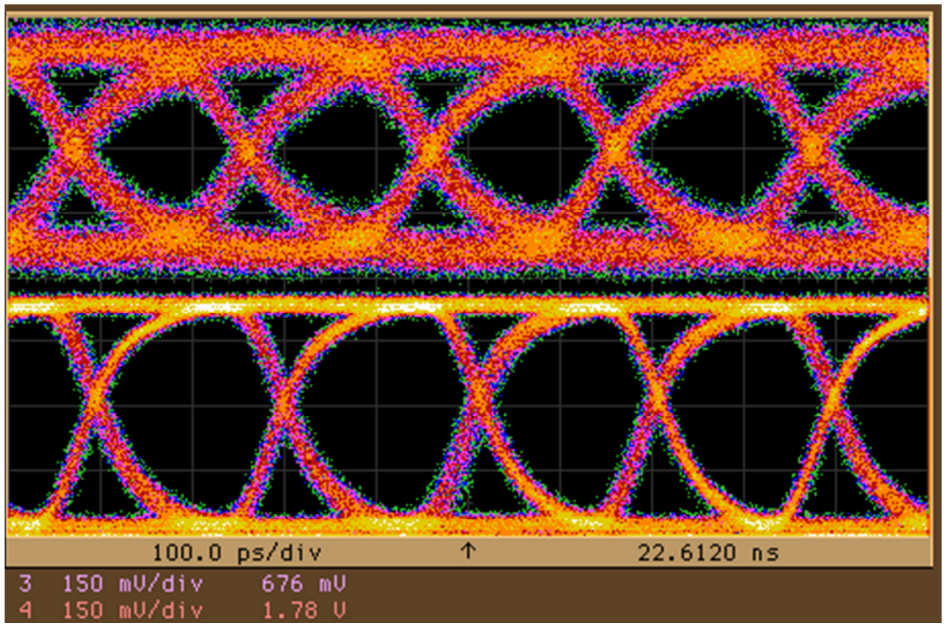


Figure 4.18. 5.0Gbps eye diagrams. Bottom signal directly from mini-tester, top signal via interposer.

Figure 4.19 is a comparable eye diagram at 6.4Gbps. The high-speed signal through the interposer (top) again exhibits some attenuation. Also, at such high speeds, the signal is not exhibiting full amplitude swing. The interposer rise and fall times start to limit amplitude swing, thus at higher rates we are rise-time limited.

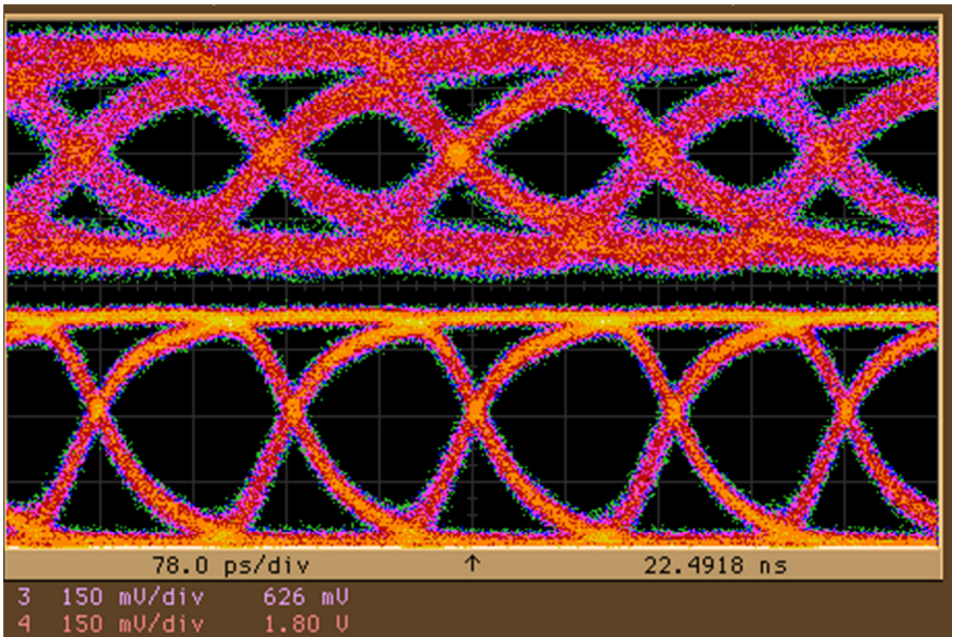


Figure 4.19. 6.4Gbps eye diagrams. Bottom signal directly from mini-tester, top signal via interposer.

At 6.4Gbps the mini-tester, jitter was measured to be about 35ps (including 6σ). Jitter increases on the signal when it is passed through the interposer, due to the inherent properties of the interposer itself. The jitter through the interposer was measured to be about 65ps (including 6σ) as shown in Figure 4.20. However usable eye openings can be seen. In order to fully utilize the high-speed capabilities of the mini-tester, an interposer with improved rise and fall times must be developed.

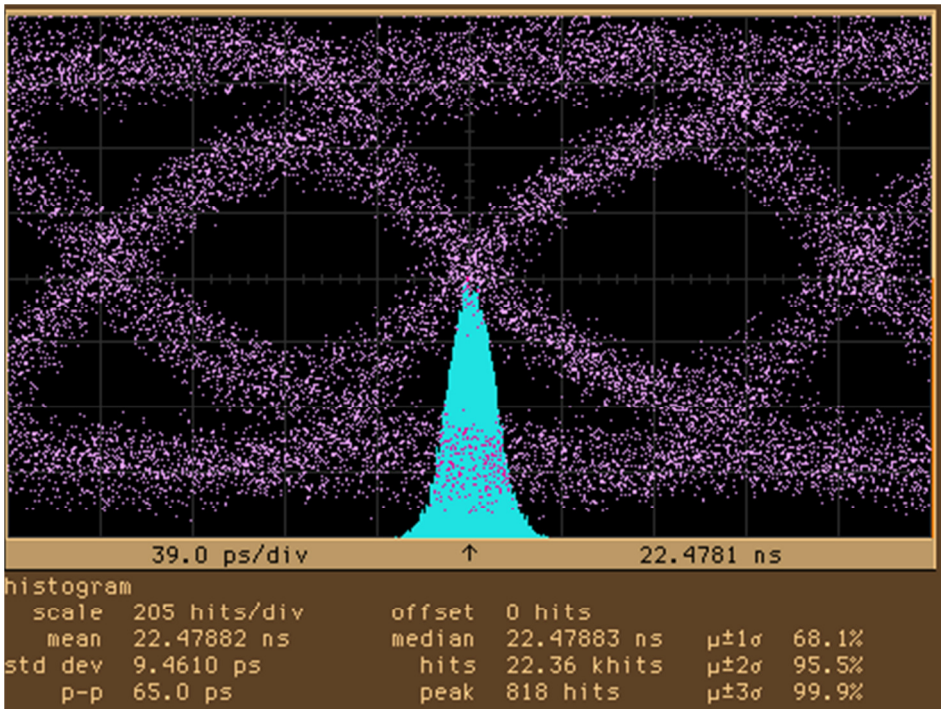


Figure 4.20. Jitter measurement of 6.4Gbps through interposer.

CHAPTER 5

ENHANCED TEST MODULE USING MULTI-GHZ FPGA TECHNOLOGIES

In the Chapters 3 & 4, two distinct approaches to enhancing ATE test performance were presented. The first approach consisted of designing modular test electronics to be used within ATE infrastructure in order to enhance test performance. The modular test cards use resources from the ATE, such as multiple high-speed channels, to generate higher-performance test capabilities. This approach was demonstrated to be highly effective for many test applications (see Chapter 3). However, this approach was limited by ATE resources, which could only be enhanced to a certain degree.

A second approach to enhancing ATE performance is presented in Chapter 4, in which stand-alone test systems are developed with higher performance test capabilities. Although designing a test system can be an arduous task, it can be achieved in a feasible timeframe by limiting functionality. The chapter demonstrates a mini-tester to perform wafer-level testing at multi-GHz rates. By limiting the function of the mini-tester to only produce high-speed test signals, it is developed within a reasonable time-frame and at a reasonable cost using off-the-shelf components. However, the mini-tester does not provide the complete test solution by itself. Another (low-cost) test system is required to perform traditional tests such as DC parametric measurements, reliability testing, etc.

The research presented in this thesis aims to take advantage of both approaches to enhancing ATE performance by developing a stand-alone test module in the form of an n ATE plug-in module. A state-of the-art multi-GHz FPGA is used in the test module, which allows it to operate independently of the ATE. Designing the test module as a plug-in module allows it to operate within ATE infrastructure, thus permitting full use of ATE resources when required. Therefore the test module is designed to enhance test capabilities within ATE infrastructure, while not using any of its resources. They can be designed to enhance various ATE performance criteria based upon DUT test requirements, thus allowing much flexibility.

This chapter presents the test module in detail. The test module's concept and test methodology are discussed in the next section. Next, its design is presented. The design of the test module is separated into two blocks, the core logic block, and the application specific block. The application specific block is designed based on specific test functions. In this research, six functions are demonstrated – specifically: high-speed signal multiplexing, loopback testing, amplitude adjustment, timing adjustment, jitter injection, and low speed testing. These functions are discussed in the final section.

5.1 Concept

Over the past four decades, ATE performance has improved and new testing capabilities added. However, the advances in many ATE performance measures and capabilities have not kept up with the advances in semi-conductor technology [1]. While device data and clock rates have surpassed that of most ATE, some ATE manufacturers offer add-on instrumentation to generate higher speed signals. For example, Verigy has a high-speed extension card to generate signals up to 12.8Gbps [89]. Credence offers a card

to extend signal generation up to 6.4Gbps [90]. Both these cards are ATE specific. If a designer is using another manufacturer's ATE, a huge capital outlay would be required to access such instrumentation. Therefore this research presents the development of generic (ATE independent) test modules that interface, not directly with the ATE, but through a device interface board (DIB - explained below).

The concept of the test module is a generic electronic module designed to enhance certain testing capabilities within ATE infrastructure. This concept allows the enhancement of specific performance criteria of ATE quickly and feasibly. For instance, most existing ATE are limited in signal speed to 6.4Gbps. Limiting the solution to focus on increasing the high-speed capability of the ATE, a test module can be developed to generate signals above 6.4Gbps. Since the test modules do not connect directly to an ATE, they can be used with multiple ATE systems via specific DIBs. This also permits full access to ATE testing resources, as the designed solutions are compatible with existing ATE infrastructure.

The DIB acts as an interface between the ATE and the DUT. It is basically a PCB that is designed to meet the mechanical and electrical requirements of a particular ATE test head and provide the electrical interconnect paths between ATE and DUT. Most ATE have "pogo" pins or similar compliant contacts on their test head to support signal connections. The DIB makes physical contact to these pogo pins and routes these signals to a more central or convenient location where they are delivered to the DUT. In this case, the DIB is also designed to accommodate specific test enhancement modules. The test modules fit directly into the DIB and are in between the signal path from the ATE to the DUT. Therefore signals from the ATE to the DUT must pass through the test

modules. Developing a DIB is much easier than redesigning a test module for a specific ATE test head. Therefore, generically designing a test module with the notion of using a DIB greatly enhances its portability across various ATE platforms.

Figure 5.1 shows a conceptual cut-away view of an ATE test head, with a multi-layer PCB DIB. Multiple test modules are shown to connect on the top side of the DIB and can be controlled synchronously by an external PC. However, in most production testing environments, where robotic handlers are used, the test modules would connect to the bottom side of the DIB.

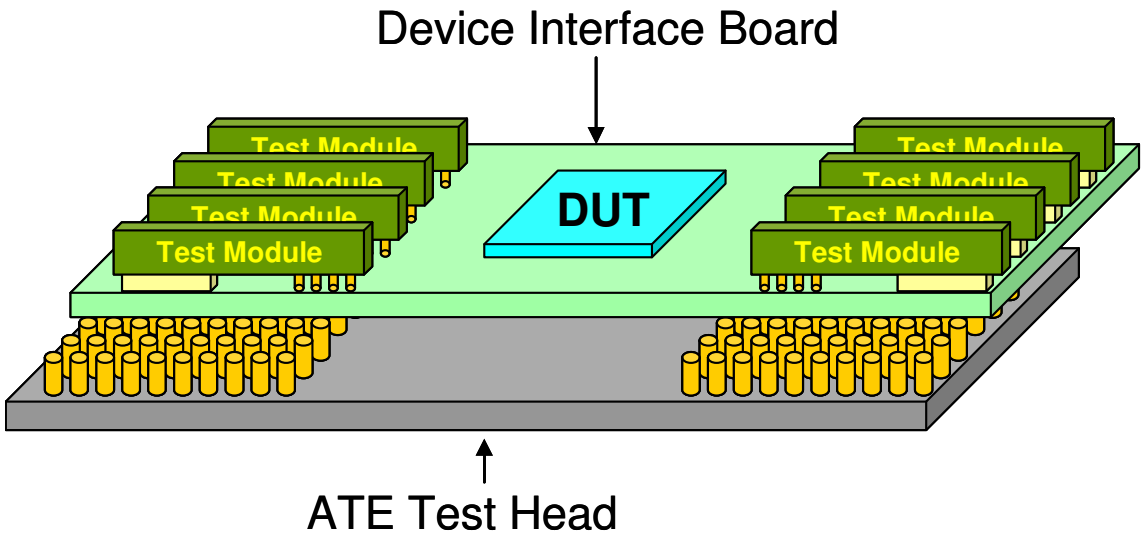


Figure 5.1 ATE test head shown with device interface board. Test modules are designed to plug into DIB.

In most test applications simple DC tests such as I/O voltage and current measurement are also required and are better handled by existing ATE. Therefore the test module is designed with provisions to allow ATE signals to pass directly to and from the DUT. Low speed data and control signals connect between the test module and ATE, while high-performance test signals connect between the module and DUT. This is done

using high-performance RF switches and allows the full suite of ATE functions to still be available for use on the DUT.

5.2 Methodology

The test module itself is designed in two blocks (see Figure 5.2). The first block is a core component that exploits state-of-the-art FPGA technologies. The use of an FPGA allows the test system to operate independently of the ATE. Also with the FPGA, the core component controls and generates many test functions itself.

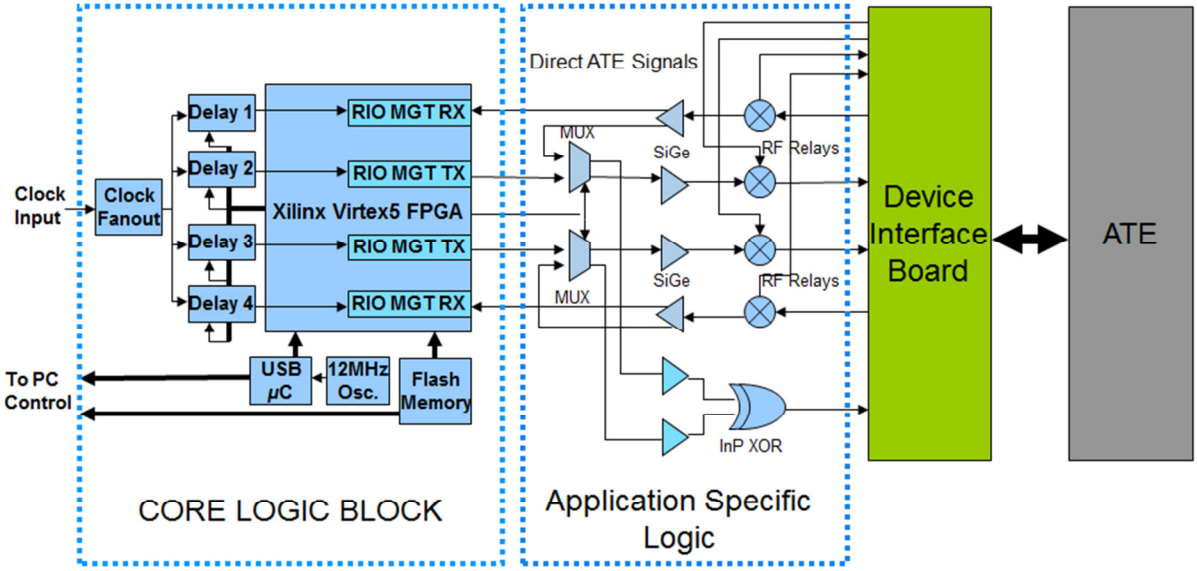


Figure 5.2 Block diagram of test module design with DIB and ATE.

In addition to a core component, the test module is designed with logic to enhance ATE capabilities based upon specific test application requirements. This block is called the application specific block. The primary enhancement targeted in this research is high-speed testing. This is the most critical enhancement due to rapidly rising clock and data

rates. Based on this requirement, the application specific logic is designed to perform the following six functions: high-speed signal multiplexing, loopback testing, amplitude adjustment, timing adjustment, jitter injection, and low speed parametric testing. Both of these blocks are discussed in detail in the next section.

Designing the test module in separate blocks allows for greater flexibility, customizability and future upgradability. The test module can be adapted as new functionalities are required and further technological advances take place. When higher performance is required of the test system, the core logic block can be redesigned independently of the application specific logic. Similarly, the application specific logic can be redesigned or additional logic added to accommodate new or improved test applications. By limiting the solution to focus on specific enhancements, the methods presented in this research allow ATE performance to be extended in a feasible, timely and cost-effective manner.

5.3 Test Module Design

In this section the design of the test module is described. The test module is divided into two blocks, namely the core logic block and an application specific logic block. The core logic block is designed by leveraging previous work done developing stand-alone mini-testers as described in Chapter 4. The application specific logic block is developed by leveraging previous work done in the development of modular ATE extensions cards as described in Chapter 3. This approach allows for more flexibility when upgrades are required and more customizability. Figure 5.2 shows a block diagram of the test module and how it connects to the DIB. The core logic block and the

application specific block are distinctly shown. The details of both these blocks are provided in the following sub-sections.

5.3.1 Core Logic Block

The core logic block consists of an FPGA that controls testing functions. In addition to the FPGA, The core- logic block includes a specialized microcontroller chip for interfacing to a USB. A personal computer communicates through the USB with the core-logic block, and provides high-level control of the tests. Supporting these are a 12 MHz crystal oscillator, and a flash memory chip to store the FPGA programming information. An RF clock source (usually an external instrument) provides a low-jitter (picoseconds) timing reference. The core logic block can be seen in the left-hand side of Figure 5.2 within the light-blue dotted box.

The latest core logic block design exploits new advancements in FPGA technology to reduce and/or eliminate dependence on ATE. For example, recent Xilinx FPGAs can produce and capture high-speed signals up to 6.25Gbps using RocketIO^{TM} multi-gigabit transceivers (RIO MGT) [91]. Each RIO MGT block in the FPGA has two TX and two RX channels. The particular FPGA used in this design contains 4 RIO MGT blocks, which implies that each test module can have up to 8 high-speed TX and RX channels. These transceivers are basically dedicated high-speed serializers/deserializers. Internal logic in the FPGA takes a 20-bit wide parallel data bus, sends it to the SerDes and converts it into a serial data stream. Therefore, internally, the data bus is required to run at only 1/20th of the serial output data rate. The slower internal data rates allow the test data manipulation, either algorithmically, comparatively or randomly, to be done inside the FPGA in parallel fashion. This allows data to be manipulated (transferred

to/from memory, or algorithmically-generated or compared) at the full sustained rate needed by the multi-Gbps serial channels. Only within the dedicated RIO MGT logic are the extreme, mutli-Gbps serial rates encountered.

How specific stimulus and response data is produced and manipulated within the FPGA is flexible and depends upon the intended application. Therefore the details of this are not shown in the figure, but discussed in detail in the next chapter. Typically the parallel data is stored, at least temporarily in RAM and/or registers within the FPGA. The data could represent deterministic test patterns, or could be produced in real-time using programmed logic to implement algorithms related to the DUT. The obvious example of algorithmic pattern generation is for memory testing, but the available FPGA logic can accommodate even complex logic functions.

On the other hand, certain aspects of the FPGA RIO MGT are common across many applications. Each RIO MGT block requires a reference clock that determines both the frequency and phase of the generated (or captured) serial data. The FPGA supports a number of possible modes. In this particular case, a 1/20th rate reference clock is used. For example, to produce a 5Gbps signal, a 250MHz reference clock is supplied. Internally the RIO MGT circuitry phase-locks to this reference clock and synthesizes a high-speed clock for the serializers and/or deserializers. This approach greatly reduces the need for high-speed clock distribution external to the FPGA. Nevertheless, the SiGe clock networks in the development platform work extremely well for distributing the 250MHz reference clocks, providing very stable (low-jitter) signals to the modules.

In the core logic block, the reference clock is delivered to a 1:4 SiGe fanout buffer in the module. Each of the four buffered outputs pass through a programmable delay

chip. The FPGA is used to program these delays chips with a 10-bit word. Delay can be set in 10ps increments. However these delay chips also include a fine tuning input port that takes an analog voltage input from 0-3.3V and allows the tuning of the delay between 0-60ps, thus allowing a much finer control of the delay. Programmable DACs can be added to the core logic block (not shown), and controlled by the FPGA, to achieve such precise timing control.

Since the clock is relatively slow (<500 MHz), and does not have "data dependant" properties, these delay elements pass the reference clocks with only minimal added random jitter – 3ps as per manufacturers specifications. Notice in the Figure 5.2 that each of the four RIO MGT channels has its own reference clock, with an independently-programmed phase. This degree of control allows the channel-to-channel skew across all channels to be adjusted independently, providing maximum flexibility for testing.

For communication to the external world, a microcontroller is used in the module to allow control from an external PC via a USB port. A flash memory device is included for programming the FPGA. These features are described in the next Chapter discussing the FPGA design.

Figure 5.3 shows a photograph of the test module. In this figure the core logic block consisting of the FPGA, USB microcontroller, flash, etc. can be seen in the central and right side of the board, along with four programmable delay chips (for the reference clocks) and the multi-pin Gbps signal connectors. Application-specific logic is seen on the left side of the board, including the SiGe select logic, adjustable-amplitude buffers; fan-out buffers, relays and SMP connectors.

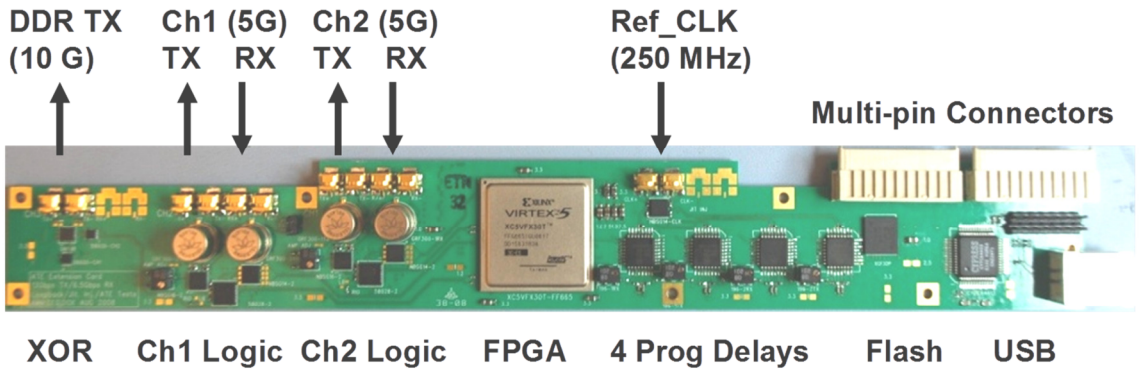


Figure 5.3 Photograph of the FPGA 5/10Gbps Module.

The application specific logic shown in Figure 5.3 is a critical component of the test module design. The concept of designing the application specific logic as a separate block, allows many different applications to be designed. They must be designed to be compatible with the core logic block. If an application is desired that is not compatible with the core logic block, provisions can be made in the core logic block and the test-module redesigned. The application specific block is discussed below.

5.3.2 Application Specific Block

The application specific block can be seen within the green dotted box in Figure 5.2. Signals from the core logic block are passed to the application specific block where they are either manipulated or passed through to the DUT based on the application. In this particular design, each of the RIO MGT TX signals are passed through a 2:1 fan-out multiplexer that allows the selection of either the core signal or the application specific signal. This signal is then passed through a variable-amplitude SiGe buffer (where the amplitude can be adjusted between 100mV and 700mV), and then through an RF relay to SMP connectors. When the relay switches pass these signals, the FPGA controls the test

module independently, and an ATE (if present) is idle. This is the most novel mode. Also, using similar modular extension principles in [44], the application specific block is designed to extend its signals from the core logic block, without needing critical ATE resources. This allows for increased customization as various ATE enhancements can be developed depending upon specific needs.

Still another possibility is to use the alternate signal paths for in-situ calibration. In past modules this flexibility has been well-worth the extra cost of the relays. On the other hand, it is critical for these high-speed applications that the relays be carefully chosen to support the extreme bandwidth requirements. The ones used in this card have a bandwidth well above 10GHz, and work down to DC.

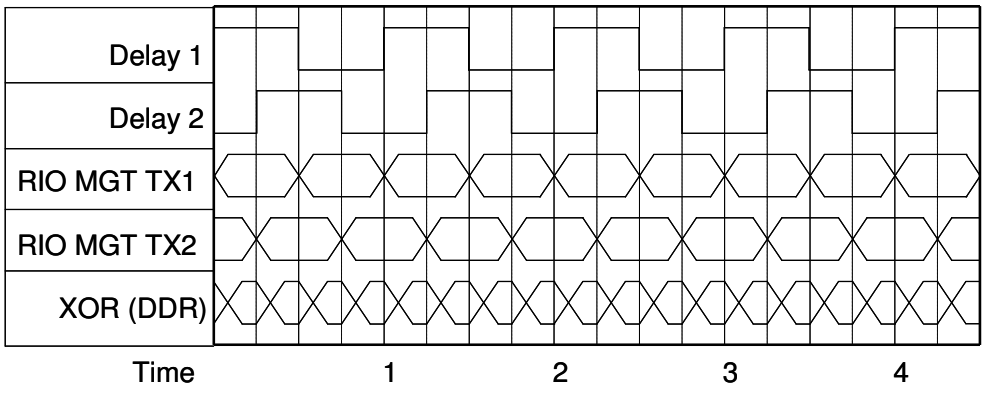
5.4 Applications

The application specific logic can be designed based upon the desired testing needs. This feature removes many limits on what types of tests can be done with the test module. Demonstrating all the test functions the test module can be designed to perform would not be a feasible task and outside the scope of this research. Therefore in this research, six commonly required test functions have been targeted. The test functions are described in the following sections.

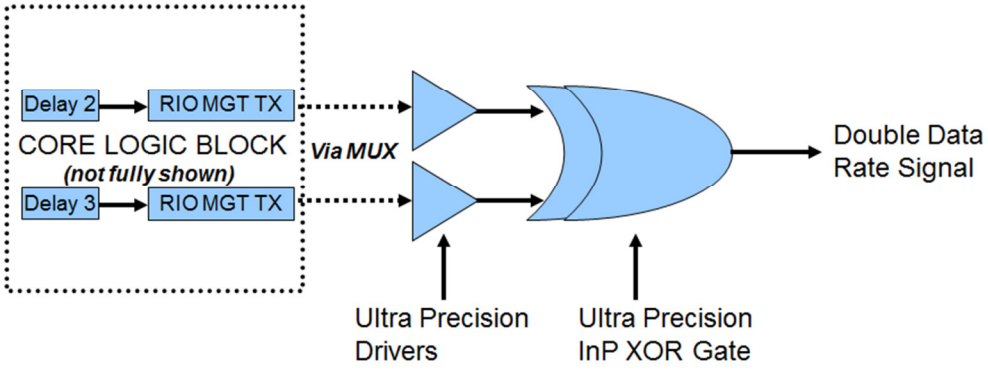
5.4.1 High Speed Signal Multiplexing

The first function developed was high-speed signal generation. There are many techniques to generate higher-speed signals from slower-speed signals, such as serializing, multiplexing, etc. However this task becomes very difficult when the slower-speed signals are running at multi-GHz rates. At these speeds, jitter is a concern, and must be minimized to retain signal integrity [92]. Also at such speeds, precision timing becomes a critical issue, as control of signals must be established on the pico-second level.

In this research, high-speed signal generation is achieved by multiplexing high-speed data signals from the core logic block and producing a double data rate signal. This is done by selecting both the RIO MGT TX channels through the muxes. The signals are passed through ultra-precision buffers to sharpen their edge rates, and then multiplexed through a high-performance InP XOR gate. As shown in [71] it is possible to create a double-rate serial pattern by encoding two normal-rate signals each offset by half a clock period simply by combining them with a high-performance XOR gate (as configured in Figure 5.4-(b)). This is a well-known logic technique for DDR generation. However, its success for synthesizing multi-Gbps test signals has met resistance due to the extreme timing accuracy requirements of sub-nanosecond bit periods. Therefore, the FPGA in the core logic block is used to set the proper timing offsets on the programmable delay chips and also to select the proper mux channels.



(a)



(b)

Figure 5.4 Multiplexing high-speed signals from core logic block to produce a double date rate signal. (a) shows a timing diagram to generate a double date rate. (b) shows logic components used.

Since the simplicity (and low-cost) of the XOR multiplexing method is retained when the high speed data is not re-clocked, any jitter present on either input shows up on the XOR gate output. So, the first requirement for success is that the two input signals to the XOR have minimal jitter. Furthermore, XOR gates have historically been notorious for adding data-dependant jitter (DDJ), not to mention the unavoidable increase in random jitter. One must further recognize that the timing accuracy demands for DDR signals are generally about twice as tight as for the normal-rate signals. Therefore the

tendency of the XOR to increase timing errors is heading in just the wrong direction (increasing errors rather than decreasing them). Furthermore, the XOR mux technique requires careful calibration at each frequency, since the optimal delay offset is frequency-dependent.

Even with all these potential difficulties, the promise of a 2x increase in data rate can be worth the effort. Many of the timing errors can be minimized through calibration techniques. However, those inherent to the XOR itself cannot be corrected (without re-clocking). So an XOR gate with minimal inherent timing errors must be utilized. For signals above 10Gbps, this means only a few picoseconds of allowable DDJ, and well under 1ps of random jitter. In this test module an InP technology XOR for exactly these reasons has been used. The specific part shown here is intended for low-jitter applications up to 13Gbps (although good performance up to 20Gbps has been demonstrated). As such, it works extremely well at the demonstrated 16Gbps speed (see Chapter 8).

5.4.2 Loopback Testing

Another test function the test module addresses is a high-speed low-jitter adding loopback path. BIST allows a device to test itself. However, a device testing itself within its own packaging does not resemble real world operating conditions. Therefore routing self-test stimulus through an external loopback path back into the device is preferable, and creates a more realistic test environment [93]. At multi-GHz speeds, the simple routing of a signal, while maintaining its integrity, is not a trivial task. The loopback path must have enough bandwidth to support the multi-GHz signal while adding minimal jitter

and minimizing losses and signal distortion. Such a loopback path has been developed and is discussed in the next section.

Loopback paths may be either passive or active. In this test module an active loopback path is designed using a RF relay, a mux, and finally a high-speed variable amplitude SiGe buffer to sharpen signal edge rates. This is shown in Figure 5.5, along with its typical setup.

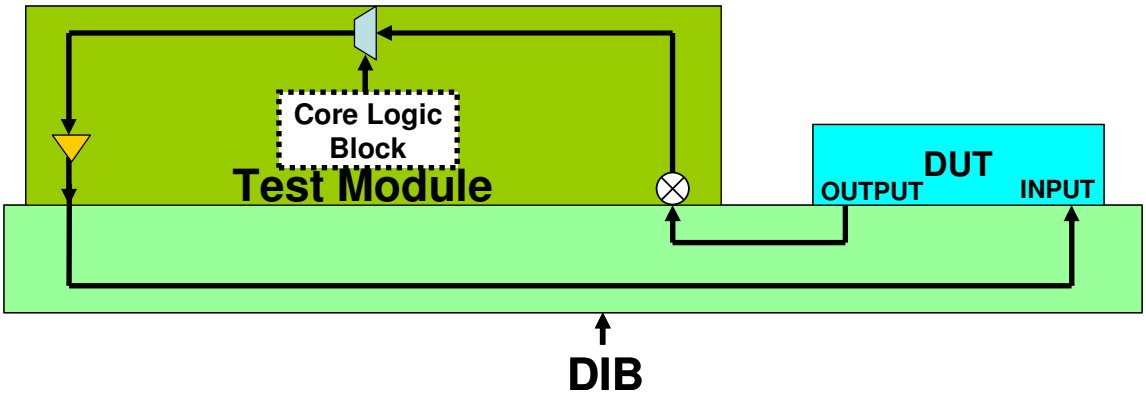


Figure 5.5 Typical loopback path for external loopback test.

To implement the loopback path, a copy of the RX channels (coming from the SiGe 1:2 fanout buffer) is taken and rerouted to the mux inputs of the core logic block. The core logic block selects the proper mux channel to allow the loopback signal to pass. As the signal is passed through the SiGe buffers, its final output amplitude going to the DUT input is adjustable (100mV-700mV). This permits dynamic input sensitivity characterization/test and allows the DUT to perform self-test through an exterior path. The test module can also sample RX data to verify that the DUT is responding correctly if desired.

5.4.3 Amplitude/Voltage Adjustment

When testing high-speed digital circuits, there is often a need to adjust the amplitude/voltage of the test signal. For example, a circuit designer may wish to determine the upper and lower threshold voltage of a circuit. Amplitude adjustment may seem like a trivial task, however at high-speeds becomes quite challenging. The device used for amplitude adjustment must be able to handle high-speed data rates. However, adding another component into the signal path can add unwanted jitter to the signal.

On the test module, the high-speed variable-gain SiGe buffers used in TX1 and TX2 are capable of amplitude adjustment. The devices contain a voltage input pin used to modify the bias voltage supplied to the buffer. Figure 5.6 shows the schematic for the output buffer. Modifying the bias voltage subsequently modifies the output signal amplitude. This allows amplitude adjustment without introducing another device into the test module. The operation of these buffers is demonstrated up to 10Gbps; therefore amplitude adjustment is also available on 10Gbps signal.

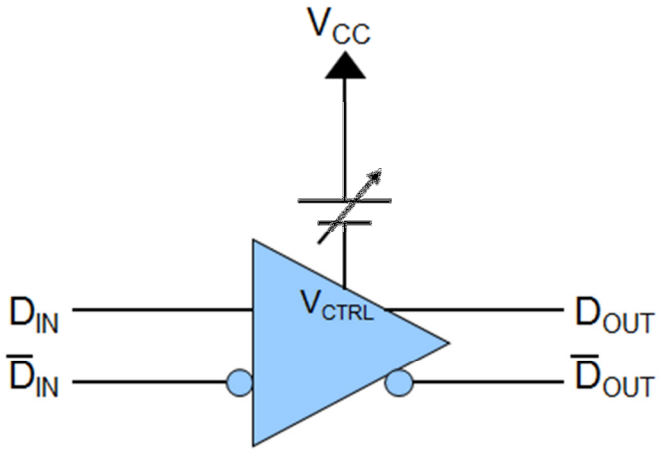


Figure 5.6 Variable-gain output buffer.

The output buffers have a V_{CTRL} input voltage pin that operates between the range of 2.375-3.465V. The input voltage is directly proportional to the output amplitude. The input voltage can be modified with a potentiometer as is designed currently on the board. However, this requires manual adjustment during testing. To automate this process, a DAC can be used as shown in Figure 5.7.

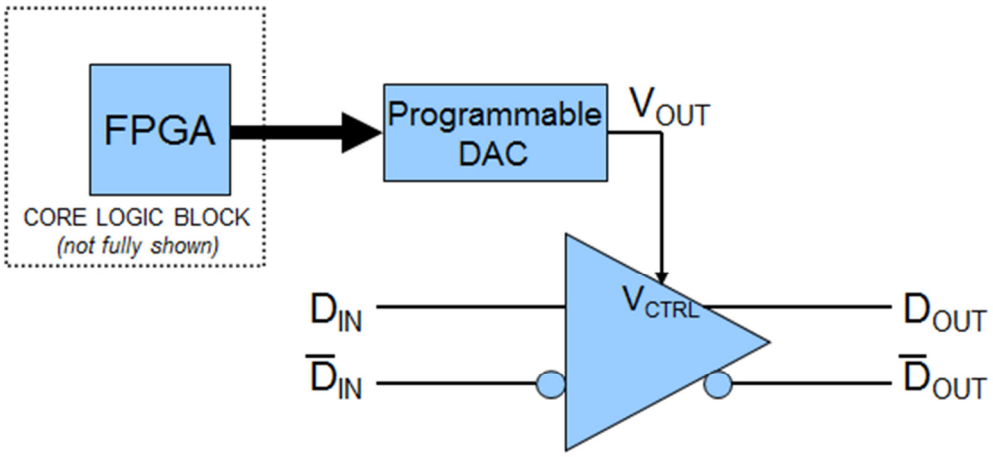


Figure 5.7 Variable-gain output buffer controlled by a DAC.

In Figure 5.7, the variable-gain buffer is controlled using a DAC. The DAC is programmed to output a voltage between the range of 2.375-3.465V. The DAC is controlled using the FPGA. The FPGA can be programmed to output 12-bit words to control the DAC. This allows for full automation during test cycles. In the current design, DACs have not been included. The inclusion of DACs is planned for the next iteration of the test module.

5.4.4 Timing Skew Adjustment

Testing high-speed digital circuits often require the need for timing or phase adjustment relative to one another. There may be a variety of reasons to desire this functionality. One reason may be to adjust the center of the data signal eye to align with the clock signal at the receiving end. This would ensure more consistent sampling of the high-speed data signal.

Another use for timing adjustment is with parallel data buses. When dealing with high-speed signals, the slightest path length mismatch between multiple high-speed signals can cause them to arrive at the destination at different times. Timing adjustment is required to ensure that the data signals arrive at the destination at approximately the same time. This is a common problem seen with ATE, where multiple test signals must be aligned at the DUT input. Generally ATEs have a feature of programming delays on each channel, thus allowing for timing adjustment. However, the precision of timing adjustment is limited by the ATEs programmable resolution. For example the Teradyne Tiger ATE has a timing resolution of 100ps [94], which may be adequate for many applications. However, a 10Gbps signal has a bit-period of 100ps, thus an adjustment resolution of 100ps is not adequate. In these situations, a picosecond timing control is required for accurate alignment of high-speed data signals.

Picosecond timing control in the test module is achieved using off-the-shelf programmable delay chips. The delay chips have a digital finite timing precision of 10ps and a range of about 10ns which is more than adequate for high speed signals. In order to set the delay, a 10 bit delay word is loaded onto the chip by the DLC. The schematic for delay chip used is shown in Figure 5.8.

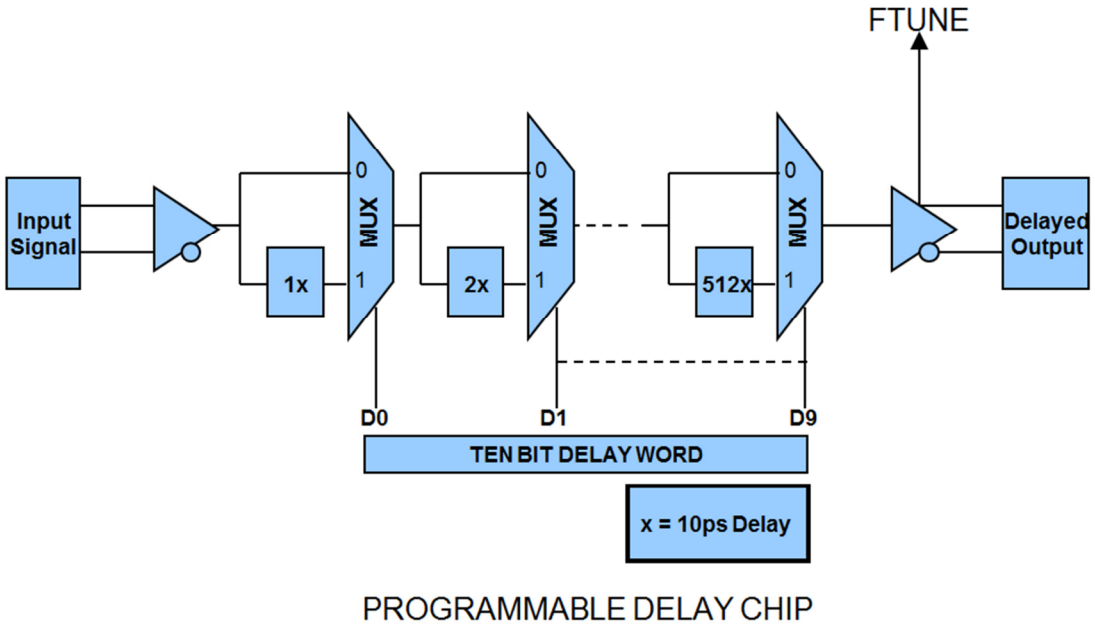


Figure 5.8 Schematic of 10-stage delay chip

The delay chip has ten distinct stages of delay. Each stage has a delay of $10x2^n$ps, where n ranges from 0 to 9. At each stage, a multiplexer is present that is used to select the delayed value or the non-delayed value. The multiplexer select lines are set by loading the 10-bit delay word onto the chip. Thus any multiple of ten between 10-10240ps can be programmed on the chip. In addition to the finite delay increments, the programmable delay chip allows for more resolution via a F_{TUNE} analog pin. This pin basically supplies a bias voltage to an output buffer. Modifying the bias voltage of the output buffer adjusts the amplitude of the signal, changing the time the signal reaches the 50% threshold, thus adjusting its skew. The F_{TUNE} pin allows an analog timing adjustment of an additional 60ps. Generally a programmable DAC is used to modify the voltage supplied to F_{TUNE}, however a potentiometer can also be used to adjust the voltage

manually. A potentiometer has been used in the current design of the test module. Future iterations will include programmable DACs controlled by the core logic block.

Each stage of the programmable delay chip ideally provides a finite amount of delay. However the actual measured delay may differ at some points. A sample of three delay chips was taken and their programmed delay versus their measured delay was plotted in Figure 5.9. As can be seen from the graph, the relationship between these two values is not linear. Lower delay values have higher percentage discrepancies. However, these differences are similar throughout the chips used. Since every delay value is made up by a combination of these ten delay values, a calibrated table can be constructed, in which the programmed delay corresponds to an actual delay. Alternatively, the F_{TUNE} feature can also be used to calibrate the actual delay to the programmed delay.

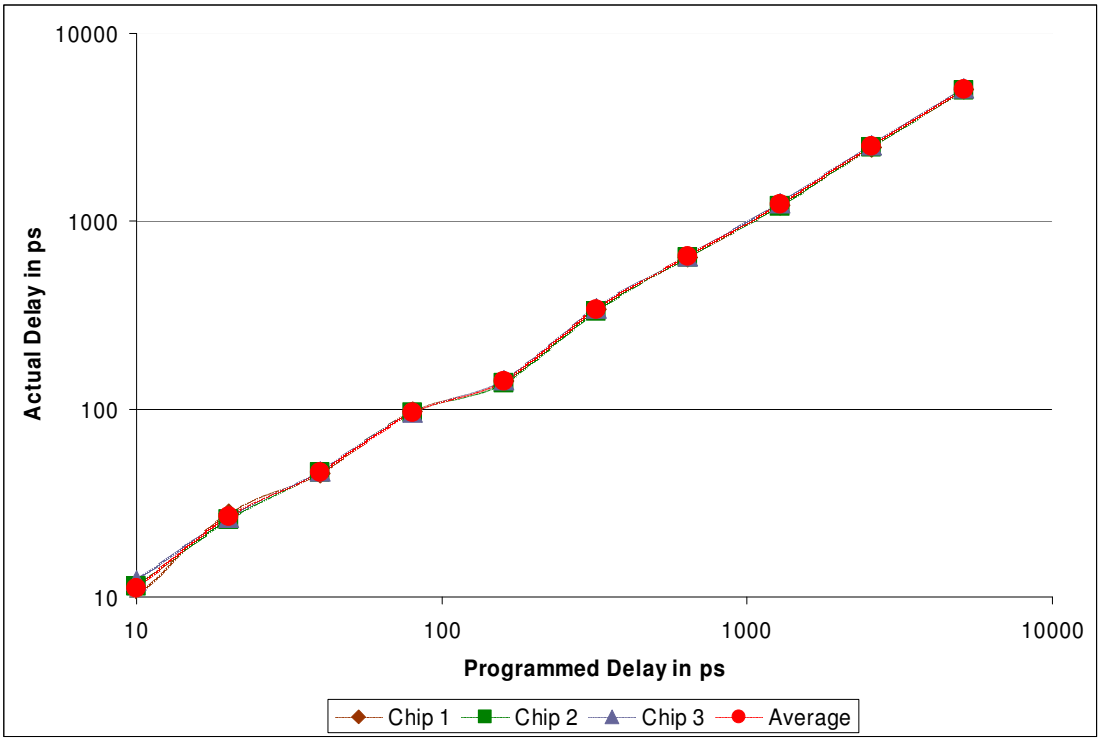


Figure 5.9 Measured delay plotted against programmed delay for three delay chips.

It is generally easier to adjust a narrow-bandwidth clock signal rather than a wide-bandwidth data signal. Similarly, it is easier to adjust a slower speed signal rather than a higher speed signal due to device limitations. Therefore in the test module, the programmable delay chips are used on the reference clock inputs for each RIO MGT as shown in Figure 5.10. The core logic block requires a clock input that is fanned out via a low-jitter SiGe 1:4 buffer. The output of the buffer is passed on to the programmable delay chip. The FPGA is used to control the delay chip and program the desired delay value onto it. The adjustable output of the delay chip is used as the reference clock for the RIO MGT blocks in the FPGA. The speed of the reference clock is usually 1/20th the speed of the RIO MGT output signal. Internally the RIO MGT synthesizes a higher speed clock from the input clock to serialize the RIO MGT data (this process is discussed in detail in the FPGA section). Thus adjusting the clock input of the reference clock to the RIO MGT, causes a timing adjustment in the RIO MGT output signal.

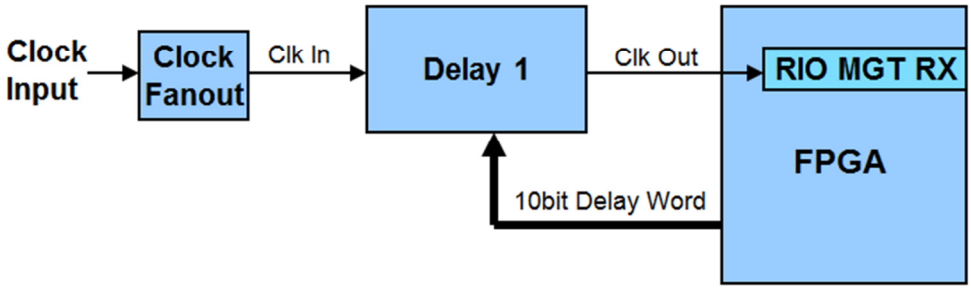


Figure 5.10 Delay Chip used with clock input for RIO MGT

5.4.5 Jitter Injection

Jitter is a critical issue with high-speed data signals. Amounts as small as 50ps can render a 10Gbps signal useless. In most cases test designers strive to minimize jitter on test signals. However in many causes the ability to add controlled amounts of jitter is desired. For example, data signals are subject to various sources of noise, such as crosstalk, simultaneous switching noise, etc. These sources are greatly amplified at higher speeds. Therefore input jitter tolerance testing is required.

The test module has provisions for adding controlled jitter by slightly modifying devices already used on the board. There are two methods by which this can be done. In the first method, the output SiGe buffer is modified. As discussed under the amplitude adjustment section above, the SiGe buffers have a V_{CTRL} input pin that allows the user to adjust the bias voltage of the buffer for output amplitude adjustment (see Figure 5.6). Controlled jitter can be injected onto the signal by AC-coupling a voltage noise source onto the V_{CTRL} input. The voltage noise is injected by an external source via an SMP connector present on the board. However, adding a noise source to V_{CTRL} will adjust the amplitude of the output signal. This effect may not be desired in some test cases.

In another method, jitter injection is achieved in the test module by slightly modifying the use of the programmable delay chips in the core logic block. As discussed above, the programmable delay chips have a F_{TUNE} input for analog timing control. Varying the voltage input on F_{TUNE}, the phase of the output signal is shifted. If an AC signal or noise is placed on the pin, the output signal shifts back and forth, causing jitter, as shown in Figure 5.11. This allows the addition of jitter onto the reference clock for the RIO MGTs. Since the reference clock is used in the serializing logic of the MGT, this

jitter is ultimately transferred to the output data signal. The RIO MGTs have internal logic that will remove small amounts of jitter and even reject very larger amounts using PLLs. This limits the amount of jitter that can be added to the output signal. However, for high speed signals, large amounts of jitter are not required e.g. a 10Gbps signal will close off with only 50ps of jitter added to it. Therefore adding jitter through the programmable delay chips is preferred to first method described.

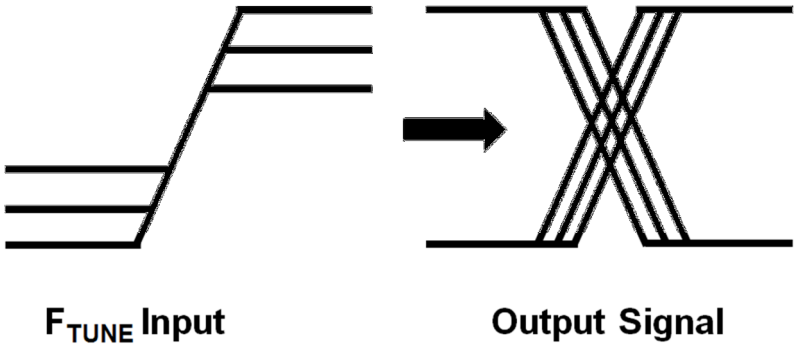


Figure 5.11 Jitter injection using F_{TUNE} pin

5.4.6 Low Speed/Parametric/ATE Testing

The test modules are designed to target specific enhancements for an ATE. As such, they are not designed to handle all testing requirements a manufacturer may desire. Additionally, there are tests that an ATE can perform more efficiently and precisely than the test module can be designed to perform, such as DC parametric testing, low speed testing, etc. Since the test modules are designed to interface with existing ATE infrastructure, it is advantageous to allow ATE tests through the test modules. Therefore the test modules are designed with RF relay switches that allow signals from the ATE to pass through the test module to the DUT. The RF relays are used to select external

signals to replace those from the module. These external signals may be connections to DC parametric test instruments. Alternatively, the relays may be used to connect to moderate-speed (<1Gbps) ATE functional test channels. In the low-speed mode, the four relays connect the DUT signals directly to ATE channels, via the DIB. This mode allows the ATE to control the test directly. Thus, standard ATE tests can be performed. This mode is valuable for debug.

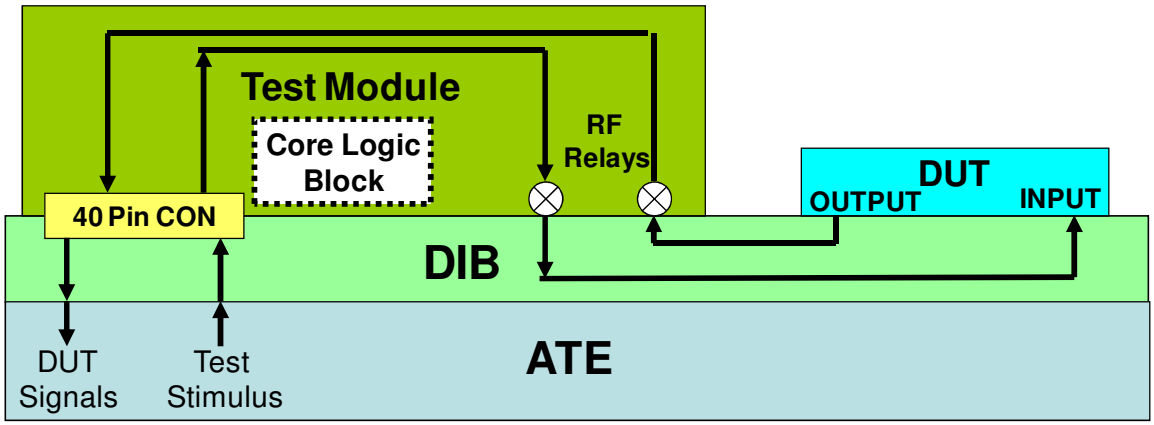


Figure 5.12 Low speed/ATE testing

Figure 5.12 shows the logic used to implement ATE testing. Test signals from the ATE are passed to the DIB, which passes them to the test module through a 40-Pin connector. These signals are routed to the RF Relays. When these relays are switched off, they pass the signals from the ATE to the DUT. Similarly the return signals from the DUT are passed through the receive relays, routed through the test module, and returned to the ATE via the DIB. In this mode the test module behaves as a passive component.

CHAPTER 6

FPGA DESIGN AND IMPLEMENTATION

FPGAs generally keep pace with electronic device trends, i.e. as newer faster electronic devices are developed; newer faster FPGAs in comparable ranges are developed. ATE have been comparatively slower to advance performance and features [2]. Current FPGA performance, in terms of speed, already exceeds that of the fastest available ATE [89] & [95]. This fact has made FPGAs an ideal choice in developing custom testing systems. Recent advancements in FPGA technology have allowed them to be used for many testing purposes [52]-[53], [71], [86]-[88], [96]-[98]. These applications include the development of stand-alone test systems, test modules and even test platforms. The ease of usability and flexibility of FPGAs allows for the addition of high-performance circuitry to enhance and add functionality to them that may be required for certain test applications. The FPGA along with its support circuitry alleviates the reliance on ATE, thus making testing more efficient and cost effective.

This chapter discusses the selection criteria of the FPGA in order to develop a suitable test module. Based on these criteria, a suitable FPGA and its features are discussed in detail. Furthermore, the design and development of the FPGA in order to support a test module and achieve testing functionality is discussed. This includes communication to the FPGA, FPGA firmware development and a software interface for user control.

6.1 FPGA Selection

This research aims to exploit new technological advancements in FPGAs to enhance the performance of ATE. The selected FPGA provides many of the functions necessary in order for the test module to operate. Therefore, the FPGA is a critical component of the test module. Its performance determines much of the high-speed performance of the test module. As such, the incorporated FPGA must be chosen carefully based up on four main criteria:

i) I/O compatibility

ii) performance

iii) capacity

iv) physical size

I/O compatibility between the ATE, FGPA, application logic and ultimately the DUT is the most critical criteria. The test module is not designed for any specific ATE, but instead should be able to enhance ATEs from most manufacturers with various I/O standards. Application specific logic for certain applications may include several devices with several different I/O standards. And finally the DUT may require another I/O standard. Therefore the FPGA should be compatible with most available I/O standards in order to increase flexibility.

The performance of the FPGA dictates the ultimate high-speed performance of the test module. For example, in this research, application specific logic is used to produce a double data rate signal from the signals generated by the FPGA. Therefore, if the FPGA is limited to 5Gbps, a maximum speed of 10Gbps can be generated from two 5Gbps signals. Obviously FPGAs capable of higher speeds are thus preferred.

The capacity of the FPGA is another issue of consideration. Much of the test generation logic, control logic, test pattern storage, etc. is stored in the FPGA. Therefore the chosen FPGA must be able to accommodate all the necessary logic and storage requirements.

Finally, the physical size of the FPGA is another factor that must be considered. In production environments, robotic handlers are usually used to load and unload DUTs onto the ATE's test head. The test module must be designed to fit underneath the load board on the test head of the ATE, where it cannot interfere with the robotic handlers. Accommodating this limitation severely restricts the maximum physical size of a test module as the available area between the test head and load board is relatively small. In this research, the available space between the test head and the load board is 1.5 inches; therefore the test module is required to be slightly less than this height.

In addition to the criteria mentioned above, the concerns for power consumption and heat dissipation are always present. Power consumption must be kept low in order to minimize resource usage, while high performance must be maintained. Lower power consumption also generates lower heat within the device, and subsequently the test module. In previous ATE enhancement modules, water cooling has been used [99], and is still an option on most ATE. An excess build-up of heat can cause undesirable test results and in some cases an unrealistic testing environment. Therefore lower power consuming and low heat dissipating devices are ideal.

Considering all the above criteria, the Xilinx Virtex-5 family of FPGAs was selected for used in the designed test module. The Xilinx Virtex 5 family is discussed in the next section.

6.1.1 Xilinx Virtex 5

The Virtex-5 family is representative of leading edge FPGA technology. It provides some of the most powerful features available in the FPGA market including high-speed transceivers with I/O speeds up to 6.25Gbps. The Virtex-5 FPGAs contain many hard-IP system level blocks, including powerful 36-Kbit block RAM/FIFOs, second generation DSP slices, SelectIO™ technology with built-in digitally controlled impedance, ChipSync™ source-synchronous interface blocks, system monitor functionality, enhanced clock management tiles with integrated digital clock managers (DCMs) and PLL clock generators, and advanced configuration options [100]. They are manufactured using a 65nm copper process technology and have an internal core voltage requirement of 1.0V, thus consuming relatively low power and dissipating low heat.

The Virtex-5 family was also a suitable choice as it supports most widely used single-ended and differential signaling I/O standards. This feature allows for the design of a generic test module that can be compatible with most available ATE platforms and a vast majority of devices. Table 6.1 summarizes the I/O standards supported by the Virtex-5 family. In this research the LVCMOS standard was mainly used and sufficient to demonstrate compatibility with other devices.

Table 6.1 Virtex-5 Family supported I/O standards [100]

Single-ended Standards	**Differential Signaling Standards**
LVTTL	LVDS and Extended LVDS (2.5V only)
LVCMOS (3.3V,2.5V 1.8V, 1.5V and 1.2V)	BLVDS
PCI (33 and 66 MHz)	ULVDS
PCI-X	HypertransportTM
GTL and GTLP	Differential HSTL 1.5V and 1.8V (Class I and II)
HSTL 1.5V and 1.8V (Class I, II, III and IV)	Differential SSTL 1.8V and 2.5V (Class I and II)
HSTL 1.2V (Class 1)	RSDS (2.5V point-to-point)
SSTL 1.8V and 2.5V (Class I and II)	

As an additional benefit, the Virtex-5 family supports numerous widely used serial protocol standards. The protocol encoding is done in the RIO MGT blocks after the transmit data has been serialized. This feature is helpful for testing devices such as network controllers, bus controllers, etc. that require specific I/O protocols such as PCIE, XAUI, etc. [101]. Additionally, the FPGA allows the user to use a custom protocol, or no protocol at all. This research mainly used the no protocol option to demonstrate proper functioning of the output. The XAUI protocol with 8B/10B was also tested to determine the feasibility of the using protocols directly from the FPGA.

The Virtex-5 FPGA family consists of 4 main platforms: the LX, LXT, SXT and FXT. The LX platform does not have any RIO MGT available, thus was not considered. The LXT and SXT platforms contain RocketIO GTP transceivers; however these transceivers are designed to run up to 3.75Gbps. The FXT platform was the only platform to contain RocketIO GTX transceivers capable of running up to 6.5Gbps, and therefore the FXT platform was selected to be used in this design.

The FXT platform is available in 5 models, out of which the XC5VFX30T model was selected. This was mainly due to the fact that the model's physical package size was the largest of the models that fit within the physical constraints of the underside of an ATE test head and could be mounted on the test module PCB board. The XC5VFX30T contained 8 RocketIO GTX transceivers, each capable of running up to 6.5Gbps and 360 user I/O pins. Additionally it contained 5,120 Virtex-5 slices (each Virtex-5 slice consists of four LUTs and four flip-flops) and 2,448 Kb of RAM blocks. As the RocketIO GTX transceivers are a critical component of the FPGA and the test module, they are discussed in detail in the next section.

6.1.1.1 RocketIO GTX Transceivers

The RocketIO GTX transceiver is essentially a high-speed serializer/deserialzier (SerDes) developed by Xilinx for its Virtex-5 family of FPGAs. It is a power-efficient highly configurable module that can produce line rates up to 6.5Gbps [102]. The transceivers support transmit pre-emphasis and receive equalization programming for optimized signal integrity. Additionally, the transceivers have built-in support for 8B/10B encoding, comma alignment, channel bonding, clock correction and PCIE interfacing.

Xilinx has designed the GTX transceivers into dual transceiver columns and placed them strategically close to other logic blocks in order to minimize size and power consumption. Figure 6.1 shows an example block diagram of a dual GTX transceiver column in a Virtex-5 FXT device (used in this design). Adjacent to each GTX transceiver is a cyclic redundancy check (CRC) block to provide data validation. Integrated blocks for interfacing with PCIE and Ethernet MAC are also present. The configuration and clock block provides access to the clock and configurable ports on the GTX transceivers.

Clock Management Tiles (CMTs) are used to manage the synchronization and routing of clocks and clocking parameters. Two embedded processor blocks are also present for each GTX dual column containing a PowerPC 440x5 32-bit embedded processor developed by IBM. Each processor contains a dual-issue, superscalar, pipelined processing unit, and other functional elements required to implement embedded system-on-chip solutions [103]. Although, these processors are not used in the current designs, their high-performance and flexibility may come in handy in future designs requiring more on board processing power. Finally there are two I/O columns that have direct access to the GTX transceivers. These are used to load configuration blocks, interface with the embedded processors and general communication with the GTX transceivers.

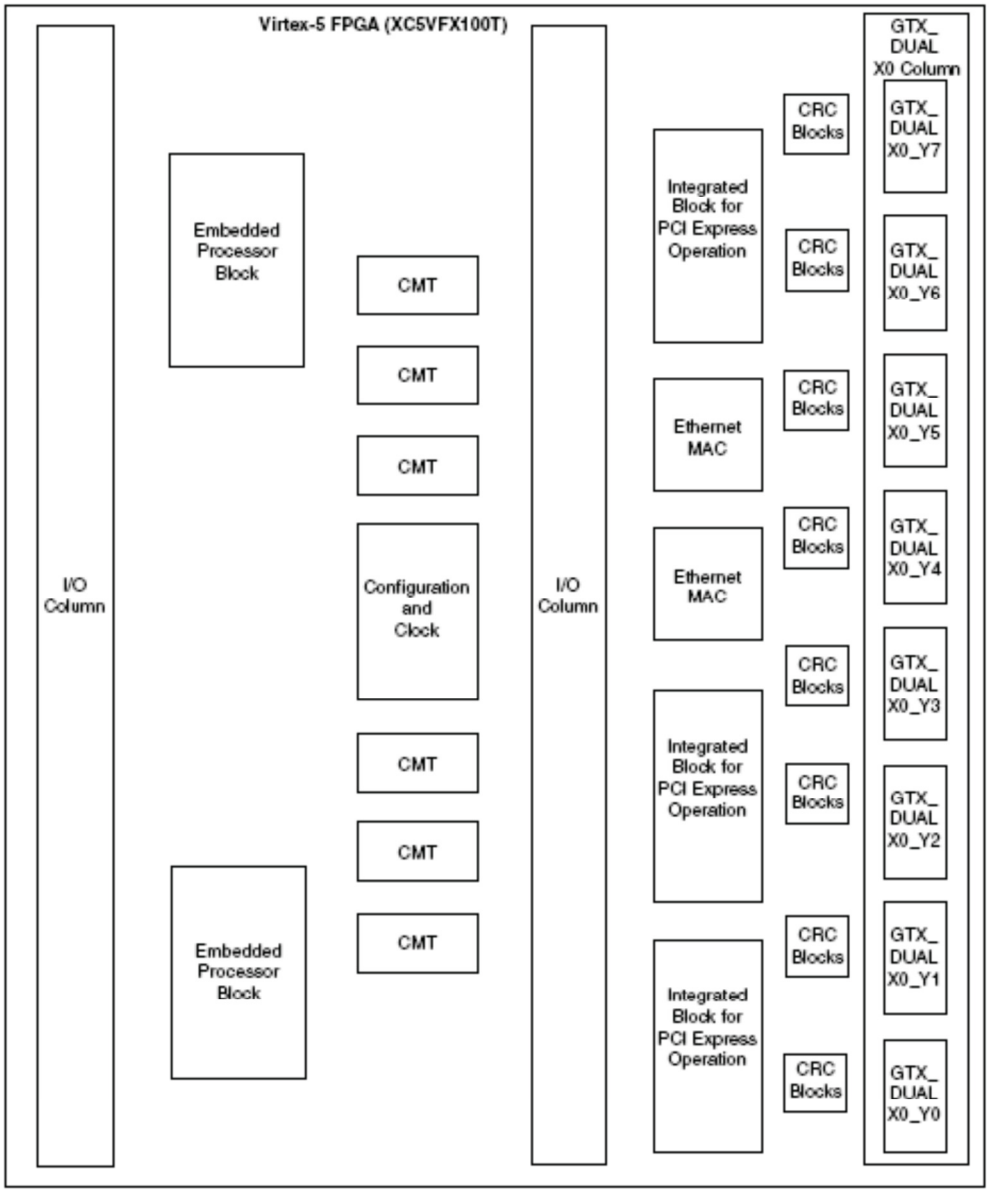


Figure 6.1 Example of GTX Transceiver Tile column in a Virtex-5 FXT device [102]

GTX transceivers are built into GTX_DUAL tiles, with each tile containing a pair of GTX transceivers as shown in Figure 6.2. Each GTX transceiver has a pair of differential transmit and receive pins which are directly accessible on the FPGA package, as seen on the left-hand side of the figure. Furthermore, each GTX_DUAL tile has five analog voltage pins and one clock input pin shared by both GTX transceivers. Sharing a

single clock source in each GTX_DUAL tile allows both transceivers to be synchronized, and also reduces size and power consumption of the FPGA. On the right-hand side of the figure, the GTX_DUAL tile's interface signals to the FPGA can be seen. Data and control signals from the FPGA are passed via these pins.

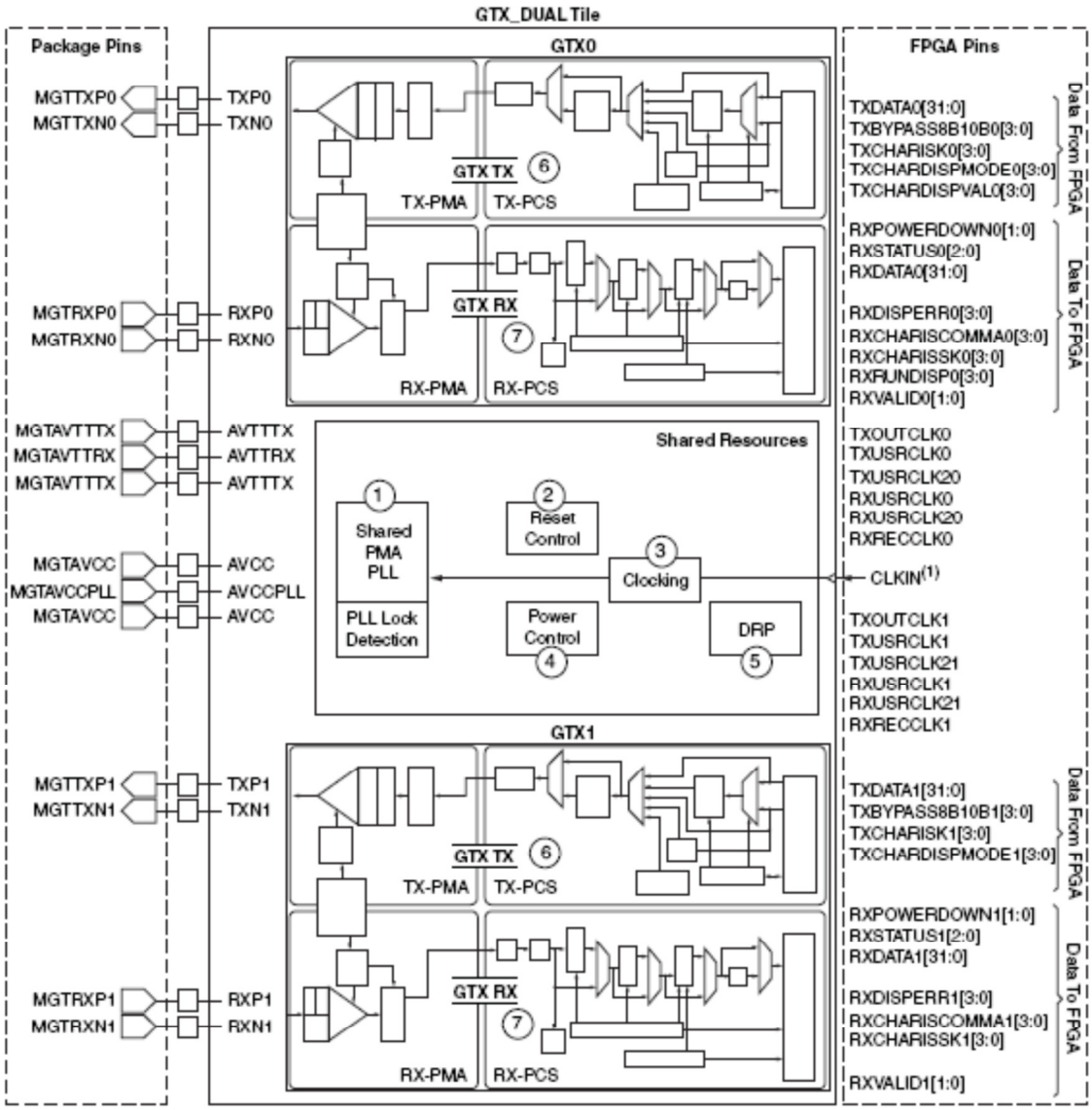


Figure 6.2 GTX_DUAL Tile block diagram [102]

As mentioned above, each GTX transceiver is divided into a transmit block and a receive block. Figure 6.3 shows a diagram of the transmit block in a GTX transceiver. As shown in the figure, data flows from the right to left. On the furthest right-end, is the FPGA TX interface, where parallel transmit data and configuration parameters are supplied. An 8B/10B encoder is built into each TX block, which allows 8B/10B encoding by simply setting an input parameter. Similarly, the TX Gearbox can encode the signal based on a 64B/66B scheme which is preferred on some high-speed data protocols. Another useful feature of the TX block is it's built in pseudo-random binary sequence (PRBS) generator which can produce 2^7-1, 2^{23}-1 and 2^{31}-1 PRBSs internally. A loopback path is also available in each TX block, which takes deserialized data from the receiver and queues it for transmittal. This feature is useful for characterizing the performance of the transceiver and debug purposes. Based upon the configuration parameters provided from the FPGA, multiplexors in the TX block select the particular data path to be serialized. This data is then sent to the parallel in serial out unit (PISO) in the TX block and serialized. The TX driver is configurable, allowing pre-emphasis voltages to be set as well as PCIE options. After serialization, data is supplied directly to the GTX TX pins on the FPGA package.

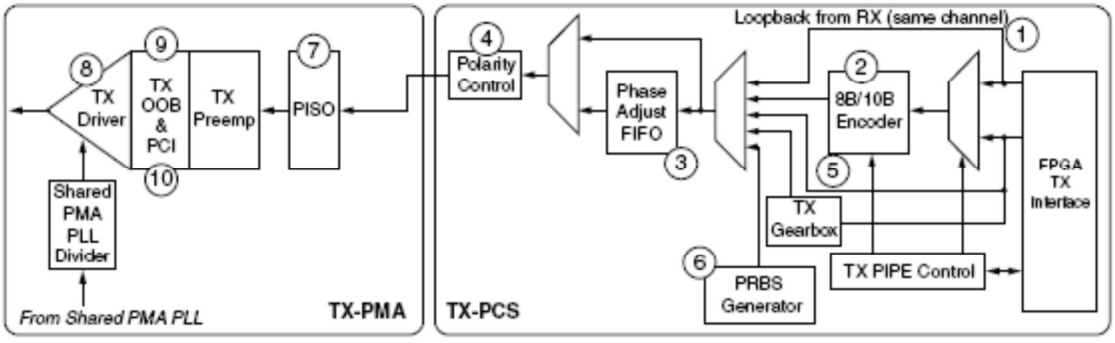


Figure 6.3 GTX TX block diagram [102]

The GTX RX block in shown in Figure 6.4. The flow of data in this diagram is from left to right. Serial high-speed data is directly input to the RX block from pins on the FPGA package. The RX data driver has built-in circuits to terminate the incoming signal and an equalization circuit to compensate for high-frequency losses. A clock data recovery circuit is also present to extract an embedded clock in the received signal. The serial data is then passed to the serial in parallel out unit (SIPO) where it is parallelized for further processing. A comma detect block is present to align the input signal accordingly. A Loss of Sync state machine alerts the FPGA if the RX channel is out of sync and malfunctioning. 10B/8B and 66B/64B (via RX Gearbox) decoding is available and can be selected in the RX block. Similar to the TX block, configuration parameters from the FPGA set multiplexor select signals in the RX data path to select a particularly processed data signal. The processed high-speed received data, now in parallel, is then provided to the FPGA through the FPGA RX interface.

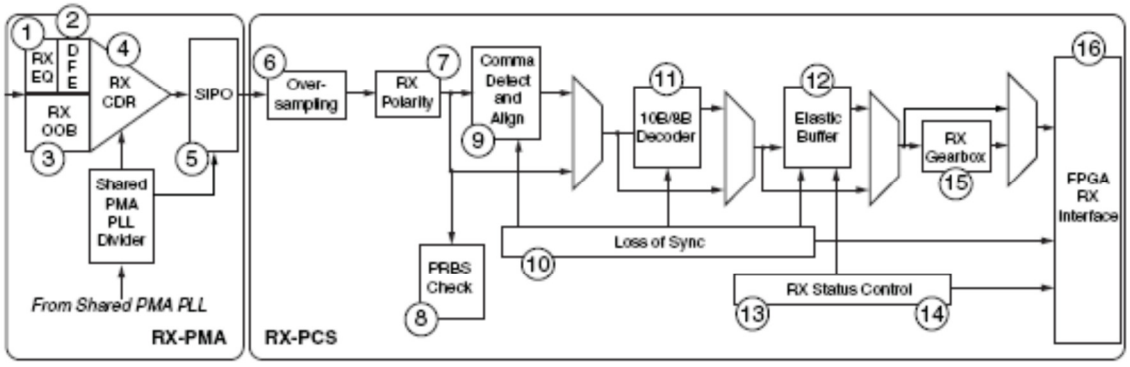


Figure 6.4 GTX RX block diagram [102]

In addition to the above functions, the FPGA contains many more functional blocks that can be used for in the future as necessary, such as DSP blocks, MicroBlaze™ processors, elementary logic blocks, complex logic blocks, etc. Controlling and designing

functionality for these complex devices is not a trivial task. The FPGA can be programmed to perform a variety of functions using hardware description languages (HDLs) such as VHSIC hardware description language (VHDL), Verilog, and schematic entry. Very complex designs can be developed for this FPGA, however the goal of this research is to design a generic control structure that can easily be ported onto other FPGAs when needed. The design of the FPGA design logic is described in the next section.

6.2 FPGA Implementation

The FPGA is the main component of the core logic block in of the test module. It is tasked with ensuring the proper functionality of the test module. It is the actual interface between the user and the DUT through which commands are sent and executed. Furthermore it is used to control the functionality of the application specific logic for testing purposes. Figure 6.5 shows a logical overview of the FPGA in the core logic block of the test module and its surrounding components. An external PC can be seen on the left of the FPGA which is required to communicate with the FPGA. However, unlike a specialty workstation as an ATE requires, the test module requires only a typical off-the-shelf PC with standard features. The PC is used for two main functions, the first of which is to program the FPGA with its internal logic. The second function of the PC is to control the FPGA, which is done through a custom software interface. Therefore the three main stages of implementing the FPGA are communication, internal FPGA logic, and software interface. These stages are discussed in the following sections.

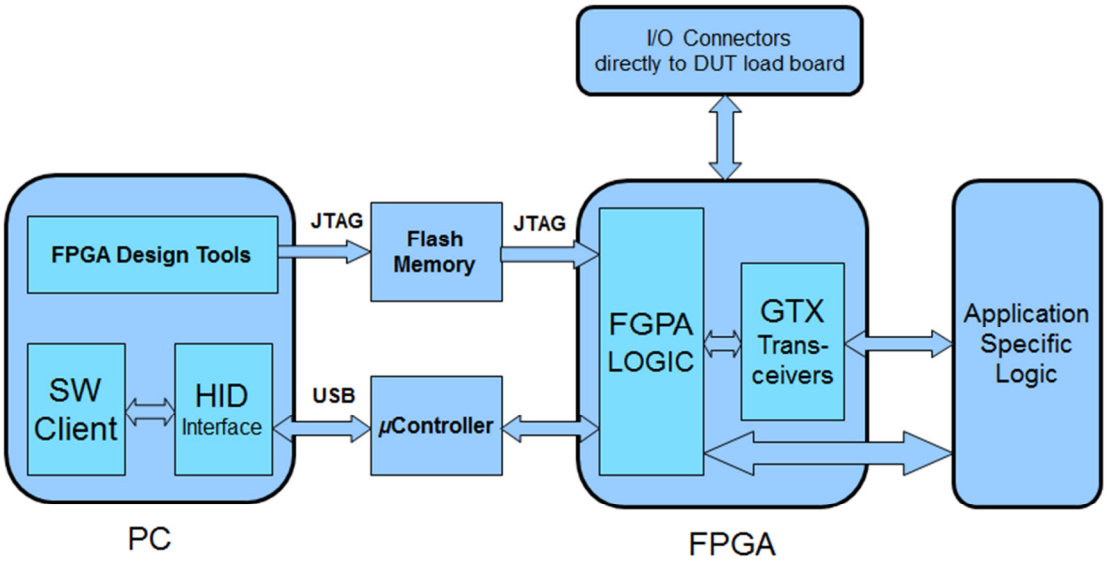


Figure 6.5 Logical overview of FPGA in core logic block and surrounding components

6.2.1 Communication

The PC communicates with the FPGA for two primary purposes. The first purpose is to program the FPGA with its internal logic or FPGA firmware. The FPGA firmware is designed on the PC using FPGA designs tools and will be discussed in greater detail the next section. Once the FPGA firmware code has been developed and compiled for the correct device, it is downloaded to the FPGA. The firmware is downloaded via a JTAG link [104]. Since a PC generally does not have a JTAG port, this process is usually done using a serial port or USB JTAG adapter.

Once a proper functioning FPGA firmware has been developed and tested, a user need not continuously download to the FPGA, unless upgrades/modifications are required. However FPGAs do not store programming on power downs, therefore when the test module is powered down, the FPGA loses its programming. To solve this

problem a flash memory device is used as an intermediary storage device. The chosen flash device must be compatible with the FGPA and large enough to store the FPGA programming information. In this research a Xilinx XCF32P In-System Programmable Configuration PROM is used [105]. The flash device is linked to the FPGA via a JTAG chain, therefore the FPGA is essentially programmed directly from the flash device. The test module developer simply programs the flash device with the desired firmware, and upon every power up, the FPGA is reprogrammed.

The second purpose the PC communicates with the FPGA is to control its functioning. Once a functioning FPGA firmware has been loaded onto it, the FPGA can be communicated to for application purposes. This entails many options - a standard PC generally has many communication ports such serial, parallel, PS/2, USB, Firewire, etc. The FGPA can communicate with any of these ports given the correct resources. For this research, large amounts of data may need to be transferred to and from the FPGA. Large test sequences, real time test data at gigabit/second speeds, sampling high-speed data, etc. can be potential applications that require large amounts of data to be transferred between the PC and FPGA. Therefore a communication port capable of high-speed data transmission is required, and as such, the USB port was chosen.

The USB is a communication standard between two devices. The USB 1.0 standard can communicate up to 12Mbs and the USB 2.0 standard can communicate up to 480Mbs [106]. The newest standard USB 3.0 can communicate up to 5.0Gbs, but requires additional pins [107]. USB 1.0 and 2.0 require a 4-pin cable consisting of power, ground and a differential data bus. The USB provides a communication link between a host device and a USB function on another device. USB device communication is based

on pipes of logic channels. A pipe is a connection from the host device to a logical entity, found on a device, and named an endpoint [108]. A pipe is formed when a host makes a connection to an endpoint. The host controller manages all the traffic to the devices and can manage up to 32 endpoints. On a PC, a host controller is usually a separate device on its motherboard, with which the operating system interacts. The host controller uses a serial interface engine (SIE) to access the physical bus. The SIE is responsible for converting the serial USB packets into valid bytes for the host controller. The SIE is required to meet the bit timing requirements of the bus and handles signal level and connections/disconnections of devices, i.e. creating endpoints. Once a USB device is connected to a PC, the PC's host controller starts an enumeration process in which a reset signal sent to the device in order to read its device class code [109]. If a valid device class code is read by the PC's operating system, it loads the proper device drivers and creates an endpoint. Once an endpoint is detected, a valid pipe can be created. This pipe can now be available through the PC's operating system, and client software can be used to access the pipe. Despite all these levels of protocol, once a valid pipe is established, communicating to a USB device becomes a seamless process.

In order for the FPGA to communicate via USB, it needs a SIE. Theoretically, this can be programmed onto the FPGA; however this process is cumbersome requires a lot of on-chip processing. Also, using an FPGA as an SIE uses much of its resources and a reliable connection is generally not achieved using it. Dedicated USB controller devices offer a more reliable and more efficient solution to communicating via USB. Therefore, in this research a Cypress Semiconductor EZ-USB FX^{TM} USB microcontroller [110] was used to handle USB communication from the FPGA to the PC. This microcontroller has

an integrated USB transceiver which handles the complex USB standards and an enhanced 8051 microprocessor which makes accessing it simple and streamlined. The particular microcontroller used operates the USB 1.1 standard which is much slower than the USB 2.0 standard, however was sufficient for evaluation purposes. The microcontroller simply requires a 12MHz crystal oscillator, which has been designed on to the test module specifically for this purpose and a 3.3V power supply.

Once the microcontroller is connected to the PC, the PC detects a default Human Interface Device (HID) and loads the proper drivers. Two endpoints are created, one for the write buffer and one for the read buffer on the microcontroller. At this point, the microcontroller is ready for USB communication, but not much else. The host computer simply polls the read buffer at regular intervals and the microcontroller accesses an interrupt routine when new data is pushed to it.

In order for usable communication to be established between the PC and the microcontroller, the microcontroller must be programmed with a core. This core is programmed in assembly language and runs on the 8051 processor available in the microcontroller. The microcontroller core is programmed to be an interface between the software client on the PC and the FPGA firmware, however is independent of both. It simply relays commands and instructions from the PC to the FPGA. As such, the microcontroller core is not expected to change unless significant changes are made on both the client software and the FPGA firmware.

Foundations of the microcontroller core were developed in [111]. The core allows simple commands and data to be written to the microcontroller's write buffer, which are then taken by the processor and processed accordingly. The microcontroller takes data

from the USB buffer and extracts instructions from the data. Essentially there are two commands that are sent to the microcontroller, read and write. The read and write commands refer to a memory block within the FPGA and discussed in the following section. Commands are sent by the client software interface hosted on the PC and discussed in Section 6.2.3. Each command is sent along with an address and a data value. When a write command is received, the microcontroller separates the address and data, and writes the data to the FGPA memory block. Similarly when a read command is received, the microcontroller reads the appropriate address from the FGPA memory block and writes the data to the USB buffer, from which the PC SW client can read it. A logical overview of the process is shown in Figure 6.6.

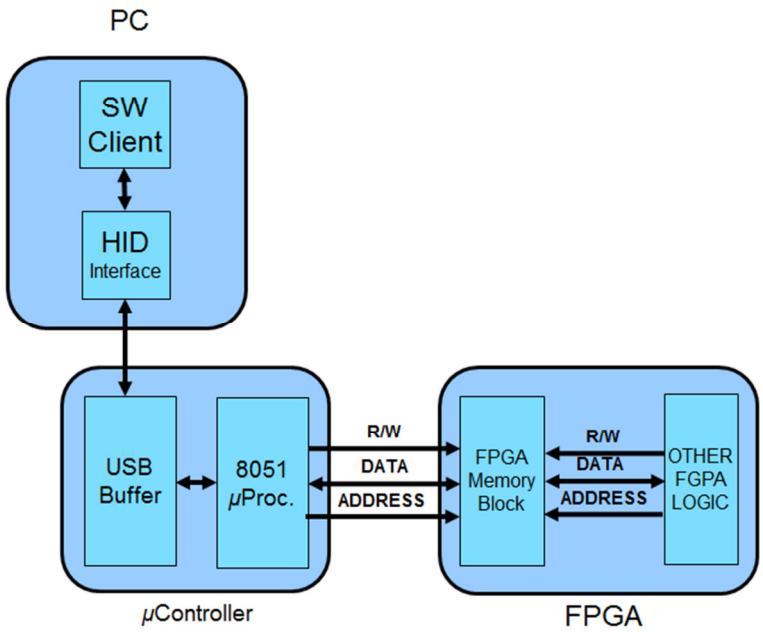


Figure 6.6 Logical overview of communication to test module

In Figure 6.6, the SW client resides on a PC connected to the test module, but specifically to a microcontroller on the test module. The microcontroller is physically

connected to the FPGA. Commands sent from the SW client are passed onto a memory block in the FPGA via the microcontroller. The memory block is a dual port memory block that can be accessed by additional FPGA logic. Once data is written to the memory block, the FPGA firmware can access and process it accordingly. The FPGA firmware is discussed in the following section.

6.2.2 FPGA Firmware

The internal logic of the FPGA, or firmware as referred to in this research, is developed on a PC using an FPGA development tool. For this research, since a Xilinx FPGA was used, the Xilinx ISE Design Suite was used to develop the FPGA firmware. The ISE Design Suite is a powerful package that allows the development of HDL logic for Xilinx FPGAs. It provides many additional design functions such as simulation tools, chip-verification/debug tools, optimization tools, etc. The ISE Design Suite also comes with an IP Solutions package. The IP Solutions package allows a developer to access Xilinx's library of pre-built logic components. The library includes a vast selection of components from as simple as 2-bit counters to as complex as DSP cores and RIO MGT blocks. These tools make a developer's task much easier, especially when portability is a concern, as the tools modify the compile logic based on the specific FPGA device chosen.

The main function of the FPGA firmware is to control the functioning of the test module. In essence, the FPGA firmware is the test module's central processing unit (CPU). Therefore it is tasked with reading instructions from memory registers and executing these instructions. In order to carry out these functions, a simple state machine

is designed in the FPGA firmware. The simple state machine implements a pseudo instruction set architecture (ISA). In its default state the state machine reads from a designated memory register in the FPGA. Data in this memory block are assumed to be instructions. After each valid memory word is read, the state machine decodes the instruction and carries out the designated function. Each instruction requires supplementary logic to implement and must also be designed in the FPGA firmware. Figure 6.7 shows a conceptual state machine that represents the logic implemented in the FGPA firmware and is discussed below.

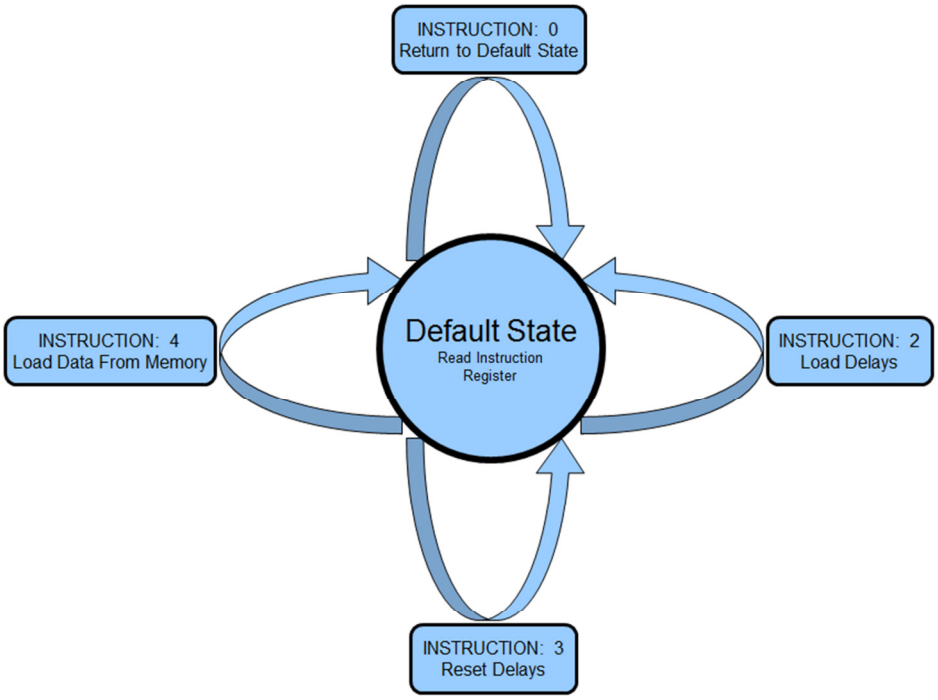


Figure 6.7 State machine implemented in FGPA firmware to execute instructions

Figure 6.7 shows an overview of a state machine implemented in the FPGA to control the functioning of the test module. The state machine is programmed to

continuously read an instruction register. An empty register or zeros in the register cause the state machine to continue reading the register in a loop, i.e. nothing different is done. Similarly, if an unknown instruction is read, the state machines clears out the register, and resumes to continuously reading it. In the case of a valid instruction, the state machine enters into a sub-routine to carry out the said function. Upon completion of the function, the instruction register is cleared out and the loop sequence resumed, unless overridden by the sub-routine.

In this research a few instructions were implemented to prove the concept of using a state machine for control architecture of the test-module. The first instruction implemented is a sub-routine that loads delays onto the programmable clock-delay devices on the test-module. The clock-delay chips require a 10-bit delay word and an enable signal to be pulsed in order for the delay to be loaded. Specific addresses are reserved in a memory block that stores the delay words for each delay chip. Since the delay chips require a load enable signal to be pulsed, a common 10-bit bus is used to route the delay words to each chip. Once the load delay instruction is received, the state-machine accesses the memory addresses for the first delay word and places it on the bus, while pulsing the load enable signal for Clock Delay 1. The next delay word is then accessed and a similar process repeated for each clock delay chip. Figure 6.8 shows a logical overview of this process.

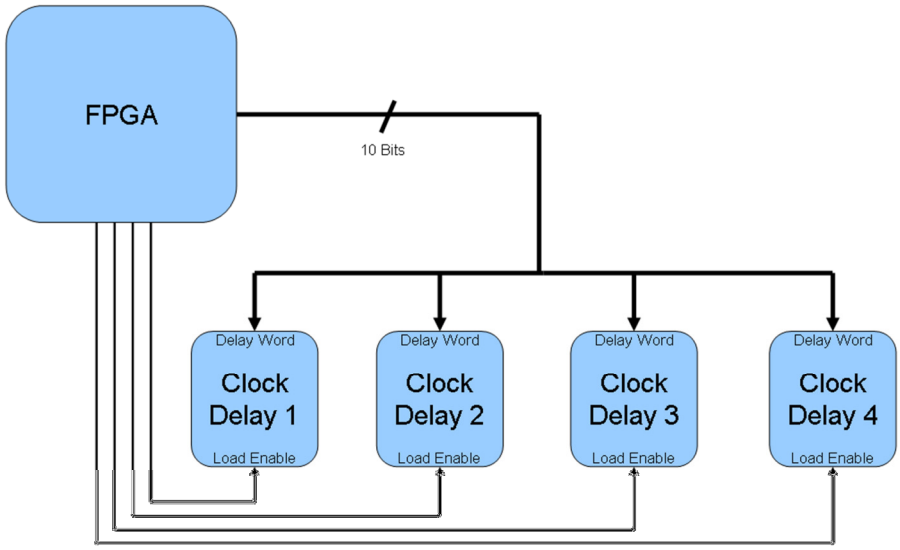


Figure 6.8 FPGA clock delay control logic overview

The second instruction implemented in the FPGA firmware is an instruction to reset all the clock-delay chips. This was a simple function to implement and used the same logic as the shown in Figure 6.8. However, instead of accessing the proper memory registers to load programmed delay values, the 10-bit delay word bus is set to zero and all the load enable signals are pulsed sequentially. The third instruction implemented is loading data from the memory onto the transmit data bus of a RIO MGT. When this instruction is sent to the test module, the state machine simply accesses data in specified locations and forwards them to the transmit data bus. This is a simple, but necessary instruction to demonstrate that specific data structures can be transmitted via a RIO MGT. This instruction can be extended in future designs to transmit blocks of data sequentially through the RIO MGT, thus sending test patterns to the DUT. Further, if automatic test pattern generation (ATPG) logic is implemented in the firmware, test patterns can be sent to the DUT via this instruction.

Instructions to the test module are issued from the SW client on the PC, therefore the FPGA firmware must also to able to communicate with the SW client. As discussed in the previous section, this is done via a USB connection and through a microcontroller. The microcontroller is physically connected to the FPGA. The connections of the microcontroller are routed to access a dual port memory block designed in the FPGA. Among other functions, the memory block in the FPGA is designed to function as an instruction register. Specific addresses in the memory are designated for specific purposes. The first two addresses are designated for instruction words; each address holds an 8-bit word, therefore allowing storage of a 16-bit instruction word. When instructions are sent from the PC via the USB connection, they are written to these two addresses automatically through one port of the dual-port memory in the FPGA firmware. The state-machine in the FPGA hardware is constantly reading these addresses for new instructions via the other port of the dual-port memory block.

Figure 6.9 shows a map of the memory block implemented in the FPGA firmware. The memory block is a dual port memory block that is pre-designed in the Xilinx ISE software suite [112]. One port of the memory is connected to a microcontroller on the test module, while the other port is accessed through the internal FGPA firmware. This establishes a communication link between a user and the FPGA, as both entities can access the same data. Specifying memory addresses for specific uses simplifies the communication process and also allows the creation of an instruction register. In this research, Addresses 0 and 1 are reserved to instruction storage purposes only, i.e. an instruction register. Addresses 2-5 are reserved for future purposes. The 10-bit clock delay values for the four clock delay chips are stored in addresses 6-13. The

remaining addresses are currently available to the user. Therefore the user has access to nearly 2K-bits of memory storage for test patterns, etc.

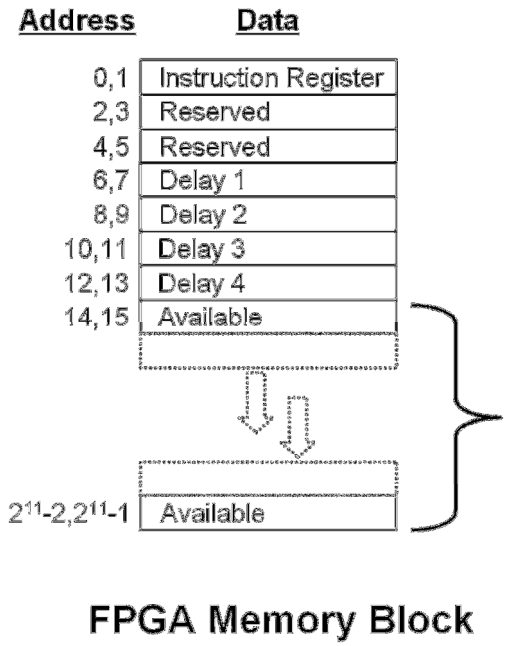


Figure 6.9 FPGA firmware memory map

Implementing the memory block as an instruction register greatly simplifies the development of new instructions. Two 8-bit locations allow for a total of 255 instructions. Most instruction set architectures do not have nearly as many instructions. If additional functionality is required, another four locations have been reserved to meet needs. The logic to execute each instruction is handled by the state-machine and implemented in the FPGA firmware. Developing new functions for the test module is done incrementally. This significantly reduces implementation time and does not require a complete design overhaul.

6.2.3 Software Client

As discussed above, a user interfaces with the test-module through a PC software client. Once the test module is connected to a PC with a USB cable, the host PC detects a HID and creates two end points and loads the proper drivers, which allow access to the microcontroller on the test module. As communication is established with the microcontroller, a simple core is loaded on to its 8051 processor (Section 6.2.1), which makes it ready to read and write commands. These commands are issued by a software client on the PC that can access the HID device detected. Figure 6.10 shows the client software screen developed for this research.

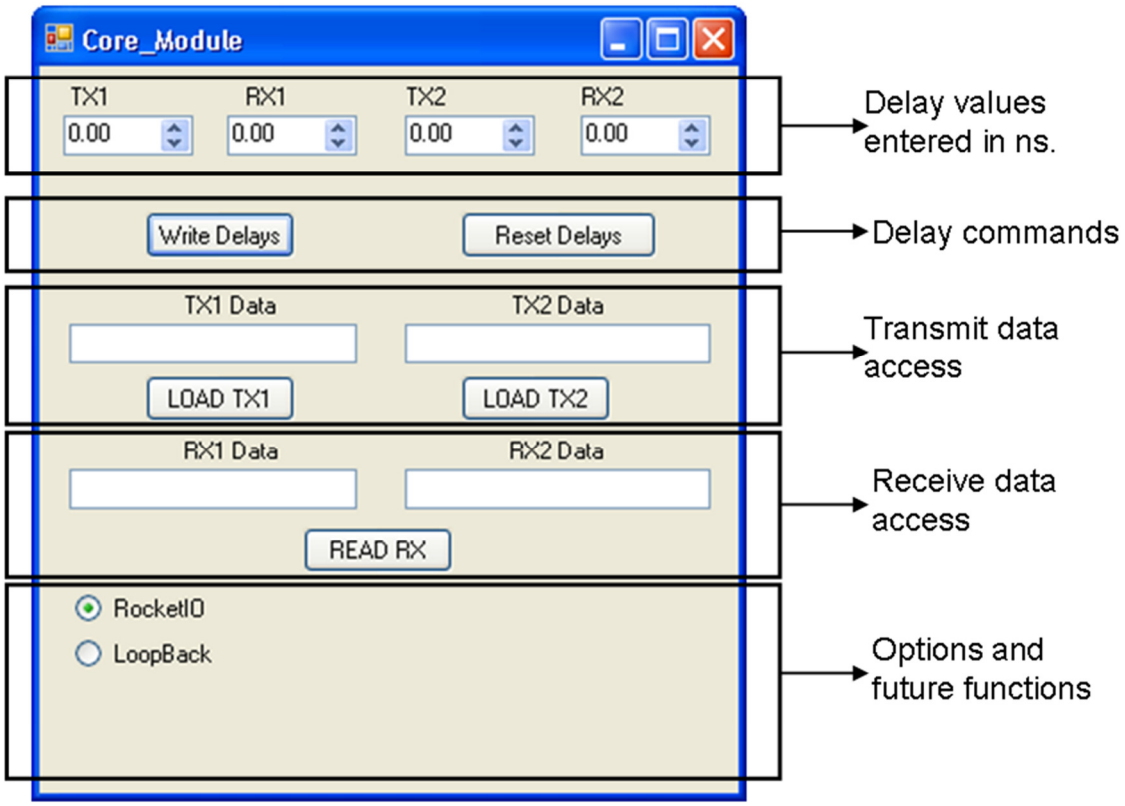


Figure 6.10 Client software screen

The client software is the interface with which a user will control the test module. It is tasked with issuing the commands that are sent to the FPGA via the USB. As such, it is usually developed in tandem with the FPGA firmware. When new instructions/functions are added to the FPGA firmware, the software client simply needs to be updated with a method for executing them. To facilitate usability, much of low level communication is hidden from the user. For example, in the previous section, it was mentioned that Addresses 6-13 are the storage areas for the delay values. These addresses are pre-programmed into the software client such that a user would simply enter a delay value and click a button. The software converts the delay value into a 10-bit word, writes the delay value into the proper memory location, and writes the instruction word to load delays into the instruction register.

In this research a basic software interface was developed to handle the instructions and functions available by the FPGA firmware, as seen in Figure 6.10. The top section is where delay values are entered in nano-second units up to two decimal places (the delay chips accept values in 10ps increments). Below that section are the buttons to load and reset the delay values on the test module. The third section is where a user can enter transmit data onto the RIO MGT channels (binary input only allowed in the text boxes). In, the following section, clicking on the “READ RX” button causes the FPGA to sample the RIO MGT RX channels, and write the data to a memory block, from which the software client reads and displays it. The final section of the window is primarily for setting optional parameters such as using the RIO MGT on the output channels or the loopback path. Future functionality can also be placed in this section.

This chapter described the implementation of an FPGA for the test module, starting from selecting an appropriate one to the software required to operate it. The previous chapter described the logical design of the test module. Once the stages have been completed, the next stage consists of physically designing the hardware of the test module. Since high speed signals require special considerations, hardware design for the test module is not a trivial task and is the topic of the next chapter.

CHAPTER 7

PHYSICAL DESIGN AND BOARD LAYOUT

The design of high-speed digital systems requires additional considerations versus designing at-speed digital systems. Passive circuit elements such as wires, circuit boards, device packages, etc. cause undesirable effects on signals such as ringing, reflections, crosstalk, electromagnetic interference, etc. In the case of at-speed signals, these passive circuit elements simply act as an extension of the device's package. At higher speeds, the effects of these same passive circuit elements are accentuated, such that the electrical performance of the signal is directly affected [113]. Therefore careful consideration of the effects of these elements must be taken when high-speed digital systems are designed.

In this chapter the physical design of the test module is discussed. Since the test module is designed to operate at multi-GHz speeds, special considerations are made in its design. These physical design considerations are discussed in the first section. In the next section, the actual physical design of the test module is presented and discussed.

7.1 Physical Design Considerations

All electrical signals travel over wires as analog signals, and are thus affected by the analog limitations of the wires upon which they travel. As frequencies increase, electric charges migrate to the edges of the wire, and essentially reduce the cross sectional area available for carrying current. This phenomenon is known as skin effect,

and is mainly due to the larger magnetic fields in higher frequency signals pushing current flow in the perpendicular direction towards the perimeter of the conductor. The effect increases the resistance and parasitic capacitance of a wire while reducing its signal to noise ratio. This effect is more pronounced as frequency increases and limits the throughput of a wire, i.e. the maximum bandwidth it can carry. Skin effect is also a contributor to energy loss at speeds in the multi-GHz range. The knee frequency (discussed below) of a digital signal determines the bandwidth required to carry said signal. Wires upon which skin effect does not start to limit bandwidth up to the desired knee frequency are required to carry high-speed signals.

The equivalent sine wave frequency of a digital signal can be approximated by its "knee frequency". Signal edges contain frequency components called harmonics. Each harmonic is a multiple of the signal frequency and has significant amplitude up to a certain frequency. The knee frequency is the frequency above which harmonics present in the signal pulse can be ignored. The knee frequency of a signal is related to the 3-dB frequency of the signal and determined by a constant and the signal's rise/fall time as shown in Equation 7.1.

$$f_{3dB} \approx \frac{K}{T} \tag{7.1}$$

Where K is the constant of proportionality related to the pulse shape, equal to 0.35 for exponential rise, and 0.337 for Gaussian rise. T is the smaller value of the signal rise (T_r) or fall (T_f) time.

The faster the rise/fall time of a signal, the higher its knee frequency is. This requires a higher bandwidth to carry the signal, and ultimately materials capable of carrying such bandwidths. Resistive losses at higher frequencies increase the rise and fall

times of a signal as the amplitude of higher harmonics are decreased, with the highest frequency harmonics most affected. Resistive loses can fundamentally change a signals performance, and therefore need to be minimized.

In addition to frequency effects, transmission line effects also need to be considered when designing high-speed digital systems. In an at-speed system, as long as the round trip propagation delay of a signal trace is small comparable to its rise time, the reflections generated by the signal can be ignored and not terminated [113]. This assumes that the path of the signal is infinitely short, as no reflections can occur on an infinitely short line since there is no propagation time between the signal and its reflection from the end of the line. A transmission line is said to be short if its electrical length ($l_{electrical}$) is less than 1/6 of its rise time [114]. The electrical length of a transmission line is calculated by dividing its physical length ($l_{physical}$) by the propagation velocity ($v_{propagation}$) of the signal. The Equations 7.2 and 7.3 define these relationships.

$$l_{electrical} < \frac{T_r}{6} \tag{7.2}$$

$$l_{electrical} = \frac{l_{physical}}{\vartheta_{propagation}} \tag{7.3}$$

The maximum speed a signal can travel is limited by the speed of light which is 84.7ps/inch. In physical media (besides air), this speed is much lower. Realistically, on transmission lines, the propagation velocity is closer to 66-75% of the speed of light, or closer to 120-180ps/inch. The propagation velocity of a transmission line is dependent on its effective dielectric constant [115] (discussed below) as shown in Equation 7.4.

$$\vartheta_{propagation} = \frac{l_{physical}}{\sqrt{\varepsilon_R}} \tag{7.4}$$

In high-speed systems, when the transmission line is not infinitely short, a reflection of the signal travels back. This happens when there is a change in the characteristic impedance on a line, for instance when a line is routed to a device pin. These reflections can be minimized by placing a terminating resistor in parallel to a fixed voltage source and the signal. To achieve maximum minimization, the resistor must have the same characteristic impedance value as the transmission line [113]. Using the equations above, it can be seen that a high-speed signal with 100ps rise time travelling through an FR4 medium (dielectric constant $\approx$ 4.5) has a maximum allowable unterminated line of less than 10mm. This is an extremely short distance when designing physical circuit boards; hence practically all high-speed signals must be terminated with impedance matched resistors.

In most digital systems, PCBs are used, and signal traces are designed on them. Any transmission medium is lossy, and at multi-GHz speeds these effects are emphasized. Therefore the choice of PCB materials can have a large impact on the performance of a high-speed system [102]. The parameter used to describe the performance of PCB materials is its dielectric constant. The dielectric constant of a material is its relative permittivity for frequency 0. Relative permittivity is a measure of the effect of the dielectric on the capacitance of a conductor. Higher dielectric constants cause signals to travel slower in a medium, thus lower dielectric constants are almost always preferred.

FR4 is the most commonly used substrate material for PCBs. Standard FR4 with a dielectric constant of 4.5 provides good performance when basic design rules are followed. However, FR4 is available in a wide range of dielectric constants, 2.8-4.5

[113]. Lower dielectric FR4 materials are generally more expensive, however preferable when designing high-performance high-speed digital systems [102].

In PCBs, additional metal layers are placed around the signal layer. Changing currents on any metal wire produce magnetic fields through induction which can generate undesirable electric currents. This is true for traces on a PCB and the effect is known as crosstalk. To mitigate this effect, the additional layers placed around a signal layer are generally used as ground or voltage planes, upon which the effects of crosstalk are negligible. These layers are often referred to as reference layers.

The characteristic impedance for a trace in a PCB is dependent on its stack up as well as its geometry. The impedance of a trace is determined by its inductive and capacitive coupling to nearby conductors, such as other traces, other layers, pads, vias, connectors, etc. Other factors also contribute to the final impedance of the trace such as substrate properties, conductor properties, distance to nearby conductors, etc. Two of the most commonly used configurations on PCBs are microstrip and stripline; these are shown in Figure 7.1. When a trace is routed on the outer most side of a PCB, i.e. does not have two reference layers around it, the configuration is known as microstrip. Conversely, when a trace is routed in an inner layer of a PCB, i.e. the trace does have two reference layers around it, the configuration is known as stripline. Based on these definitions, there are obviously more stripline traces available versus microstrip traces in a multi-layer PCB. Striplines are generally preferred to microstrips as striplines have two reference layers providing radiation shielding. Microstrips leave one side exposed to the environment, however the upper and lower layers are preferred to minimize via stubs, and therefore both configurations have distinct advantages.

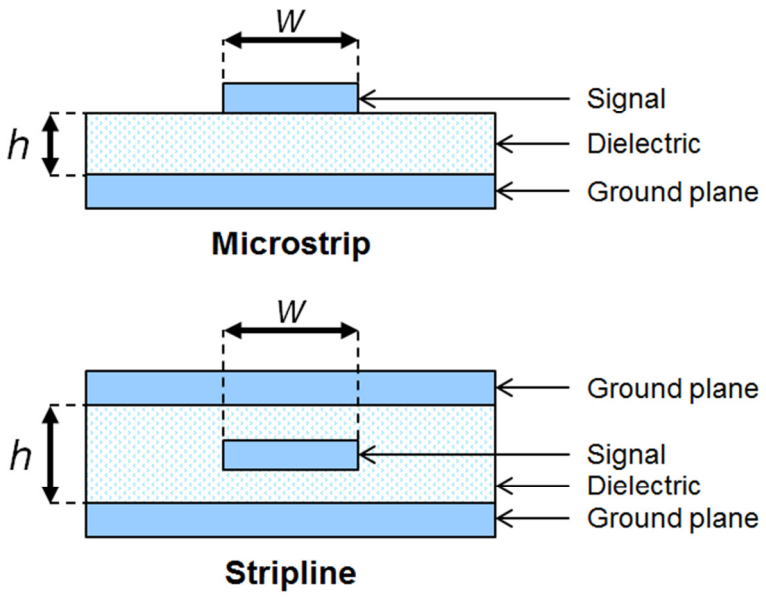


Figure 7.1 Microstrip and stripline configurations

The characteristic impedance of each configuration depends on its form. When a signal passes through a trace in either configuration, there is a difference between the trace voltage and the layers around it. This difference in voltage forms a capacitive effect which varies depending on the height, width and length of the trace. The equation to calculate the characteristic impedance of a trace is shown in Equation 7.5.

$$Z_0 = \left(\frac{L/inch}{C/inch}\right)^{\frac{1}{2}} \tag{7.5}$$

As seen in the equation, the characteristic impedance of a trace is a function of its inductance and capacitance, and not dependent on the signal frequency. However this constant ratio is a function of the physical geometry of the transmission line. The equations to find the characteristic impedance of a trace based on geometry are shown in Equations 7.6 and 7.7 [113][12].

$$Z_0 = \frac{120\pi}{\sqrt{\varepsilon_R}\left[\frac{w}{h}+1.393+\frac{2}{3}ln\left(\frac{w}{h}+1.444\right)\right]} \qquad for\ \frac{w}{h} \geq 1 \qquad (7.6)$$

$$Z_0 = \frac{60}{\sqrt{\varepsilon_R}} ln\left(8\frac{w}{h} + 0.25\frac{h}{w}\right) \qquad for\ \frac{w}{h} < 1 \qquad (7.7)$$

Characteristic impedances commonly range from 10Ω to 300Ω in transmission lines, and are typically between 50Ω to 75Ω in PCB traces [113]. When using FR4 based substrates in a PCB, Equations 7.6 and 7.7 can be simplified to find the following impedance approximations shown in Equations 7.8-7.11.

$$w = 2h \qquad for\ 50\Omega\ microstrip \qquad (7.8)$$

$$w = \frac{h}{3} \qquad for\ 50\Omega\ stripline \qquad 7.9)$$

$$w = h \qquad for\ 75\Omega\ microstrip \qquad (7.10)$$

$$w = \frac{h}{8} \qquad for\ 75\Omega\ stripline \qquad (7.11)$$

Standard resistors are easily available in 50 Ω and 75 Ω values on the market. Using the above equations, trace widths can be calculated to result in these standard values. The actual impedance value of the trace will vary slightly than calculated using these equations. Therefore when using such standard resistors as termination resistors, minimal reflections may be seen. This may cause a marginal change in resulting voltage swing of the signal compared to the original voltage swing.

Traces that are routed in straight lines have a constant width (w). When a trace is turned or bent, its width can change. In order to keep the width constant, and thus the impedance of a line constant, straight lines are preferable. However routing signals with

all straight line traces is simply not realistic. When turns and bends in signal traces are required, right-angle bends must not be used. At a 90˚ bend, the effective width of the trace changes, causing an impedance discontinuity due to the capacitive coupling of the additional conductor area to the reference plane. This change in impedance can cause undesired reflections on the signal depending on the amount of impedance mismatch. Instead mitered 45˚ bends should be used, which keep the width and thus the impedance of the trace constant.

7.2 Test Module Physical Layout

Since the goal of the test module is to enhance ATE performance, maximum digital performance is desired from the PCB. Therefore the test module is designed by paying careful attention to the physical design considerations discussed above. Ideal performance could be achieved by following the considerations literally, for example by using custom termination resistors to match trace impedance exactly, routing signals straight lines only, not using vias, etc. For obvious reasons, this is not feasible or realistic. Furthermore, there are physical restrictions that must be followed.

Chapter 5 discussed the physical width and height restrictions imposed on designing an ad-on module to fit within existing ATE infrastructure. In addition to width and height restrictions, there is a thickness restriction imposed on the module in order for it to connect to the DIB. In this research, the maximum thickness available for designing the test module was 0.062 inches. This allowed for a design using ten layers. Figure 7.2 shows the PCB stack up for the test module.

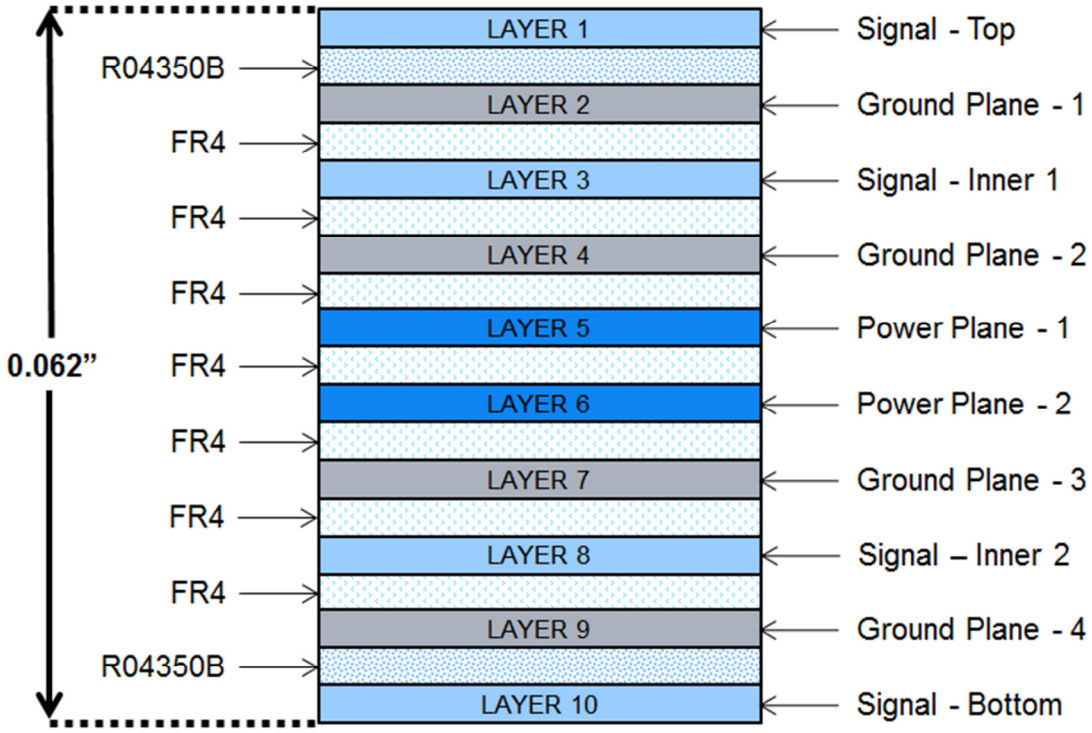


Figure 7.2 Test module PCB stack up

All of the active device components used in the test module are surface mounted on the top side of the PCB. Therefore the top layer is used to route the highest-speed signals. The top and bottom layers are both microstrips. Although using stripline to route higher speed signals may have been preferred, via transitions would have been required that would have distorted signal performance. When transitions were required on the high-speed top layer signals, they were routed directly to the bottom layer. This was to minimize ringing effects potentially caused by mid-board via stubs. Since the performance of the top and bottom layers was critical to the performance of the test module, a high performance dielectric was used for these layers. The dielectric used was R04350B, a glass-reinforced hydrocarbon and ceramic dielectric which had a dielectric constant of 3.66 [116] compared to 4.5 for standard FR4, thus providing greater

bandwidth. This dielectric can be fabricated on PCBs using standard FR4 processes, however is more expensive than standard FR4 materials. Two more layers are used as signal routing layers – layers 3 and 8. These layers were used to route non-critical signals such as control signals. Since the signals routed on these layers were not critical to the high-speed performance of the test module, standard FR4 was used for these layers.

For each signal layer, a ground layer is present as a reference plane. Reference planes should be contiguous for the length of a trace and splits should be avoided. Splits on the reference plane cause impedance discontinuities on the trace above or below them, as the coupling effects to the reference plane are changed abruptly [102]. Routing over plane splits also causes issues with return currents. Due to skin effect, return currents also travel near the surface of tightly coupled reference planes and have a tendency to follow the original signal carrying trace. At plane splits, return currents must find alternative routes, causing suboptimal current return paths and increasing the current loop area [102]. This effect increases the inductance of the trace at the split, and should be avoided. Therefore contiguous ground planes are used as reference planes and placed adjacent to the signal routing layers in the test module, layers 2, 4, 7 and 9. Also ground planes are preferred to contiguous power planes as power planes tend to be noisier and can cause undesired crosstalk effects [102].

Using four layers for routing signals requires an additional four ground layers as reference layers for optimal performance. This leaves only two layers left on a ten layer board. In the test module, the two remaining layers are used as power planes – layers 5 and 6. The test module requires six separate power supplies. Supplying six power supplies over only two planes becomes a challenging task. In the test module, these two

planes are carefully divided in four split planes where the voltage sources are needed. It should be noted that the power planes are in the middle of the PCB and well isolated from the signal layers, thus minimizing undesirable interference. The remaining two power sources were required for only a few pins. Since the two power planes could not be feasibly split to accommodate these additional supplies, they were routed as traces on the inner signal planes. Again, care was taken to keep them as far away as possible from high-speed signals.

The exact dimensions of each layer are typically optimized by the PCB fabrication house. A CAD tool, such as Mentor Graphics PADs, is commonly used to design PCBs. Trace widths for each layer are set with the CAD tool to result in the desired impedance values. On the test module, 50Ω traces were used. Once the design is complete, CAD files of the design are sent to the fabrication house. The fabrication house sets the thickness of each layer based on the total board thickness specification – 0.062". Once the individual layer thicknesses are set, trace widths may need to be slightly modified in order to result in specified impedance. This is all handled at the fabrication house by automated optimization software and seamless to a board designer.

The PCB design for the test module was done using a PCB CAD tool called Mentor Graphics PADS Suite. The first step of the process involved entering all the devices and components into a schematic editor. Once the schematic is complete, the pads of all the components are exported to a layout editor. The layout editor can attempt to auto-route the schematic connections is desired. However for high-speed performance, manually routing each signal produces more efficient results. By entering the design into a schematic editor first, manual routing of the traces is simplified. The software also has

the capability to check the schematic version against the layout version, to ensure that signals are routed to their proper destinations. Figure 7.4 shows the physical layout design of the test module. Based on the physical limitations imposed by ATE infrastructure, the height of the test module is set to 1.5” and its length set to 12”. Signal propagation in the test module can be said to move from the right side of the board to the left, i.e. input ports are on the right side of the board and outputs on the left. For greater clarity, Figure 7.3 is divided and enlarged into three sections – starting from the right – and shown in Figure 7.4-Figure 7.6. The figures are discussed below.

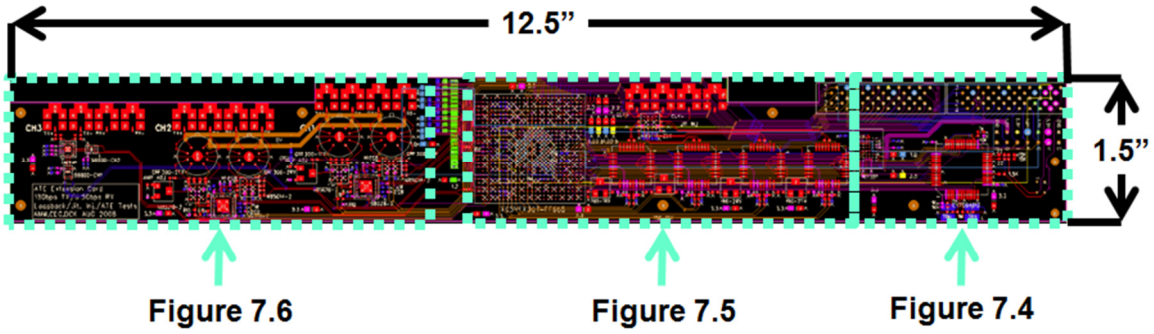


Figure 7.3 Test module PCB layout using CAD software.

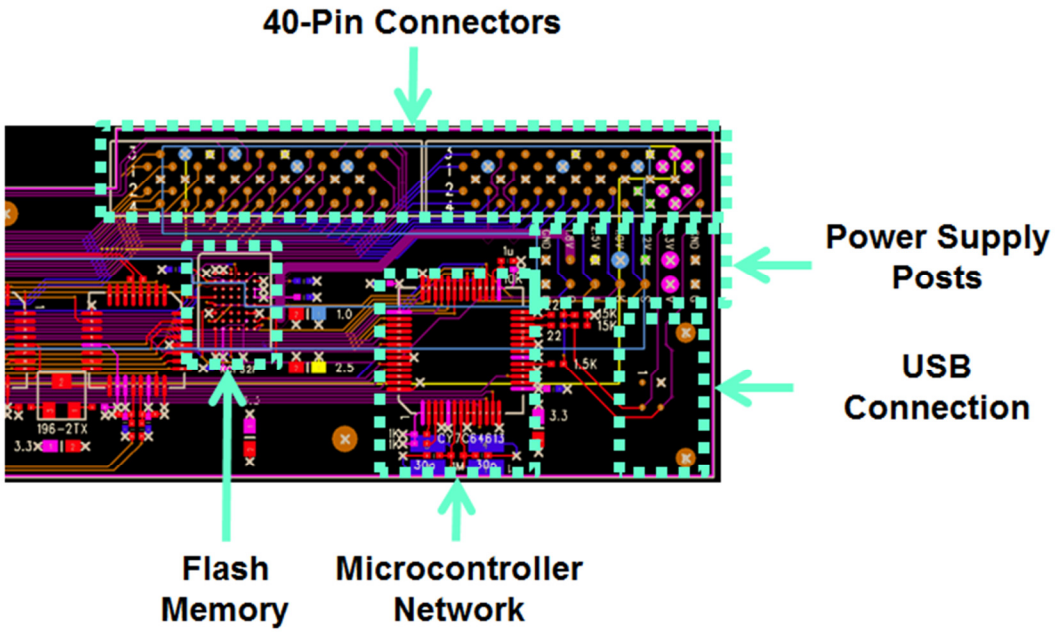


Figure 7.4 Right-most section of test module layout

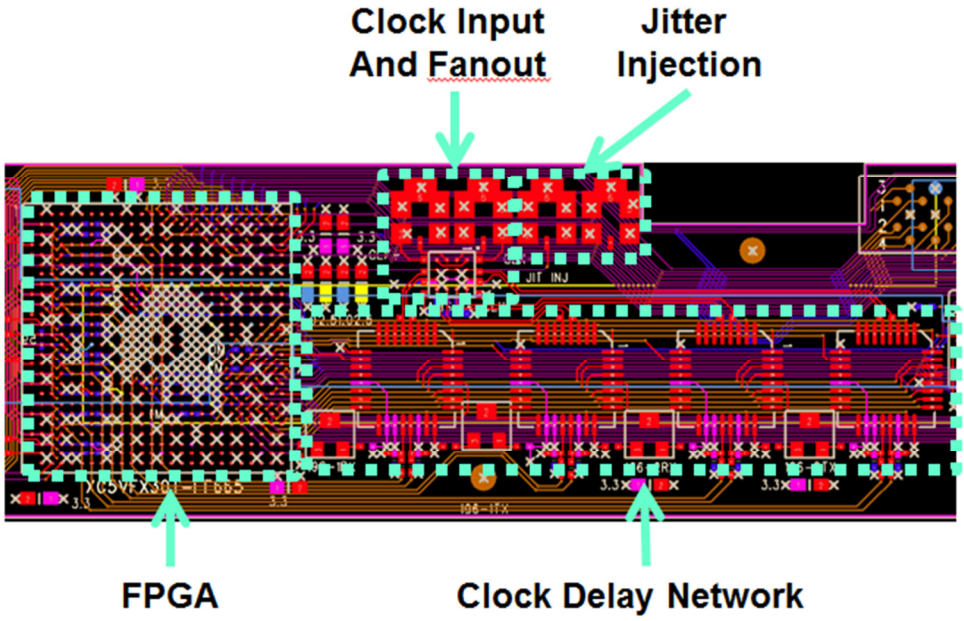


Figure 7.5 Mid-section of test module layout

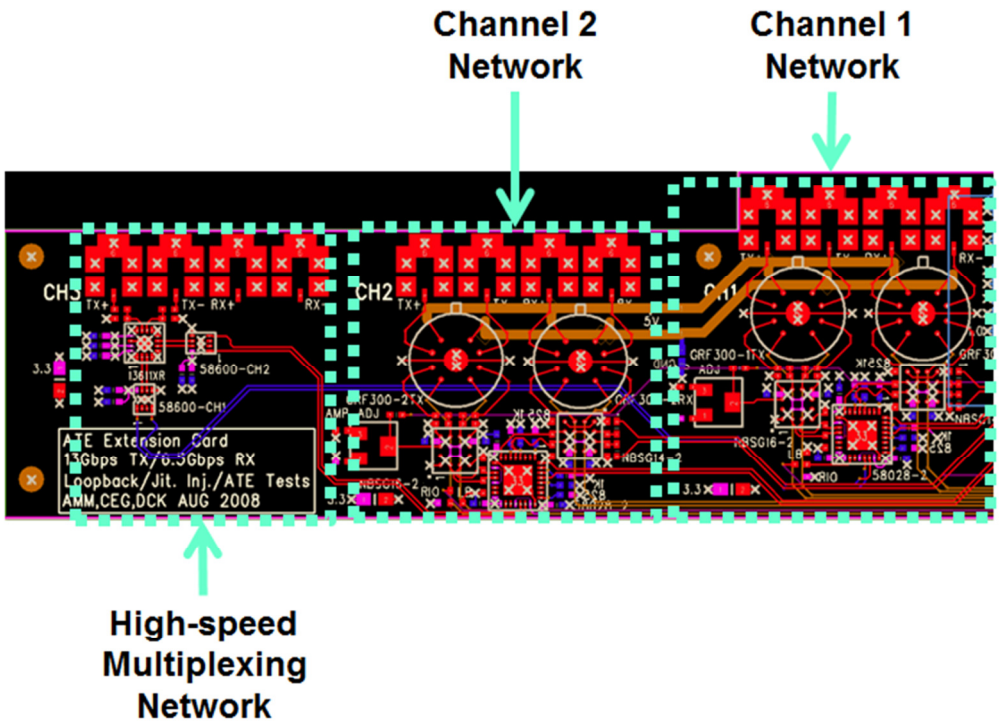


Figure 7.6 Left-most section of test module layout

Figure 7.4 shows the right-most section of the board. This section contains two 40-pin connectors which connect to an ATE through a DIB. These connectors can be used for control and communication, such as test commands, FPGA programming, USB, etc. Eight pins from the 40-pin connectors are routed directly to the RF switches (shown in Figure 7.6) for low-speed testing purposes. A few of the pins on the 40-pin connectors are designated as power input pins for the test module. For development purposes, redundant power supply posts are also present on the test module, such that it can be used on a laboratory bench. In the lower right hand-side of the figure, the USB connection can be seen. This feature is also redundant and for development purposes. Adjacent to USB connector is the microcontroller and crystal oscillator. To the left of that is the programmable flash chip. Most of the components provide an I/O interface to the test

module, and interact with each other. Thus keeping them close to each other simplifies trace routing in this section.

Figure 7.5 shows the midsection of the test module. The jitter injection subsection contains two SMP connectors. Next to it is the differential clock input to the test module. A fan-out chip supplies the clock to four delay chips in the clock delay network (discussed in Chapter 5). Clock signals are routed to the left into the FPGA. Since the clock signals are in differential pairs, the lengths of the pair must match in order for the signals to arrive in phase at the destination. The FPGA is where the first high-speed signals are generated. These signals are routed to the left into the application logic area and are routed with great care.

Figure 7.6 shows the left-most section of the board. The Channel 1 subsection contains four SMP connectors, two for differential transmit, and two for differential receive of high-speed signals. The SMP connectors are routed directly to a pair of RF relays. High-speed RIO MGT signals from the FPGA are routed directly to a MUX, where they are then passed to SiGe buffers (discussed in Chapter 5). All these signals are critical high-speed signals and routed only on the top and bottom layers for optimal performance. Furthermore, great care is taken to match the lengths of the differential high-speed signal pairs. Even a slight mismatch in length can cause one end of the signal to be slightly out of phase, and cause significant jitter. The Channel 2 subsection is similar to the Channel 1 subsection and contains all the same components. The left part of the figure consists of the components required for high-speed signal multiplexing. This subsection consists of four SMP connectors; only two are currently used for high-speed signal transmission. Two buffers in this subsection each receive a RIO MGT signal from

the Channel 1 & 2 networks. The signals are then multiplexed using a high-precision XOR gate and routed to the SMP connectors (see Chapter 5). All these signals are also critical signals, and are thus length matched and routed only on the top and bottom layers.

Routing all the required signals within the physical dimensions of the board by using only four signal planes in the PCB proved to be quite a challenge. First the critical signals were all routed only on the top and bottom layers, while trying to avoid transitions and length matching signal pairs. When a transition could not be avoided, a top layer signal was transitioned all the way to the bottom layer through a via, and vice versa. By transitioning the signal to the other side of the via, signal deterioration due to via stubs is minimized. Once the high-speed signals were all routed, the remaining signals were routed using the available area layers.

As shown in Section 7.1, high-speed signals over 10mm in length must be terminated. On the test module, all high-speed and clock signals are terminated using impedance matching resistors. In addition to resistors, numerous decoupling capacitors are strategically placed around the board to filter both high-frequency and low frequency noise. Furthermore, an elaborate network of inductors and capacitors is used to filter the power planes close to the FPGA, thus providing it with highly filtered power supplies. All this is to reap the maximum performance allowable through the system. Complete schematics for the test module design are included in Appendix C. The appendix also includes the entire layouts for all ten layers of the board.

CHAPTER 8

TEST MODULE PERFORMANCE AND CHARACTERIZATION

In this chapter the performance characteristics of the test module are presented and discussed. The core logic block is the main driver of the performance of the test module, thus its performance metrics are critical to the overall performance of the test module. In order to properly characterize the performance of the test module, the performance of the core module is presented first. Additional test functions such as high speed multiplexing, loopback testing, etc. devised within the application specific logic block utilize resources from the core logic block, thus their performance is directly related to that of the core logic block. The performance of these additional functions are discussed after the results section of the core logic block.

8.1 Core logic block –characterization

The core logic block is responsible for the control of the test module; however it also produces output signals. These output signals can be directly used for testing purposes. The output signals are produced from the RIO MGT of the FPGA within the block as described in Chapter 5. The output signal performance is mainly determined by the FPGA's RIO MGT characteristics; however the signal must pass through additional logic elements before reaching a DUT which can alter the signal's performance. The performance characteristics of these output signals are measured by connecting them to

an oscilloscope through the SMP connectors on the test board. A 50GHz oscilloscope is used that requires SMA inputs; therefore a SMP to SMA converter is used to connect the test module to the oscilloscope. An overview of the connection path is shown in Figure 8.1. High-speed test signals are generated by the FPGA through its RIO MGT ports. These signals are passed through 2-to-1 fan-out MUX, where they are selected using control logic. The signals then pass through high-performance SiGe buffers to remove any attenuation that may have occurred and produce sharper edge rates. Finally, the signals pass through an RF relay switch, which passes the signal on to the SMP connectors on the test module (TX1). A SMP to SMA cable is used to connect the signal to the oscilloscope where it is characterized.

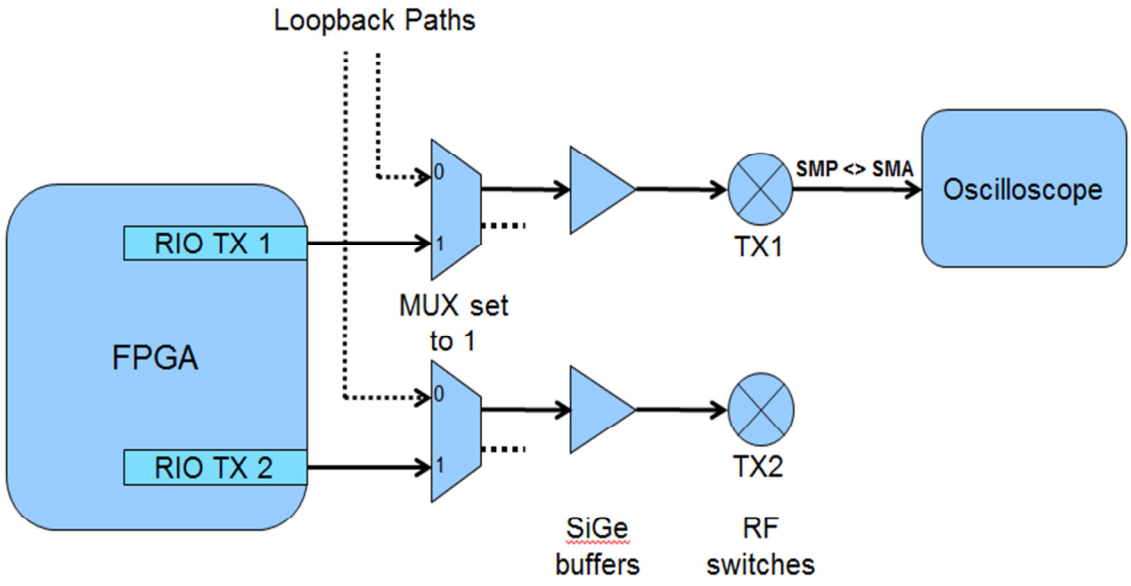


Figure 8.1 Test setup to measure core logic block performance.

Figure 8.2 shows the output of the core logic block at 5Gbps. This speed is below the maximum speed the RIO MGT blocks are specified to operate. The signals produced have relatively low jitter, are symmetric, and have wide eye openings. A rough

measurement of jitter showed the signal carried ~20ps (p-p) jitter. It should be noted that portions of the measured jitter are contributed by the additional components within the signal path in addition to the native jitter produced by the FPGA.

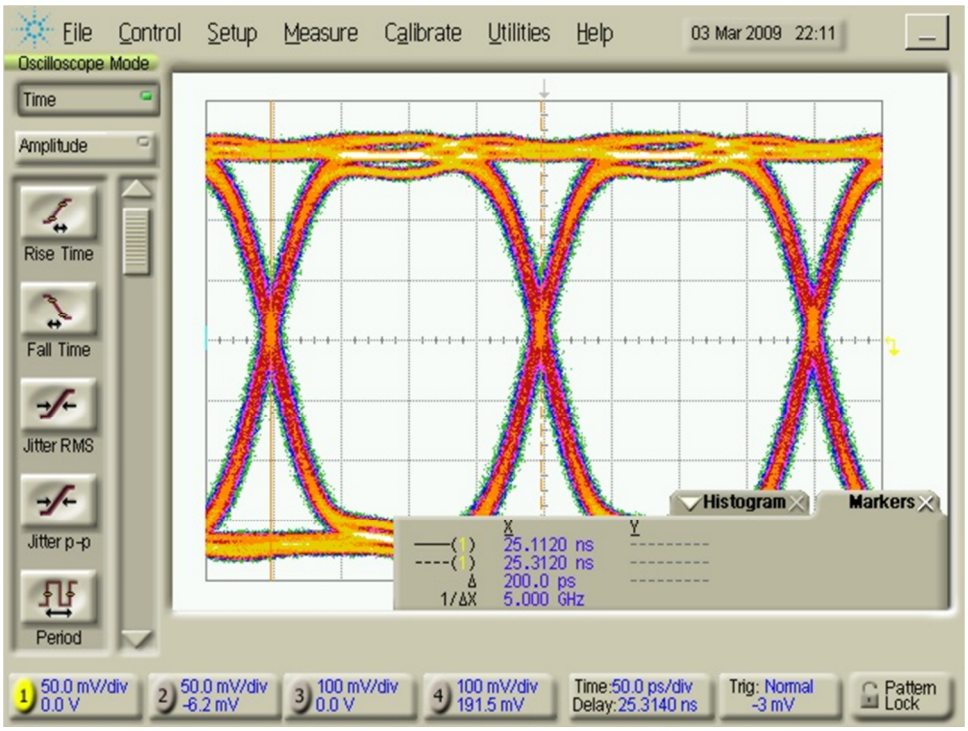


Figure 8.2. Core logic block output @ 5Gbps

Figure 8.3 shows a rise time measurement of the core logic block output signal at 5Gbps. The 20-80% rise time for this signal is measured at ~40-42ps, which is typical of SiGe technology. According to the driver's manufacturer, the maximum input data rate for this part is up to 12Gbps and it's typical rise/fall time is 40ps [117].

Figure 8.4 shows output from the core logic block at 6.25Gbps. This is a significant level as the Xilinx Virtex-5 RIO MGT is specified to operate reliably up to this speed. The results show wide open eyes. Jitter is measured to be ~22ps (p-p), and a rise-time measurement (not shown here) indicates the same SiGe driver rise-time of ~40-42ps.

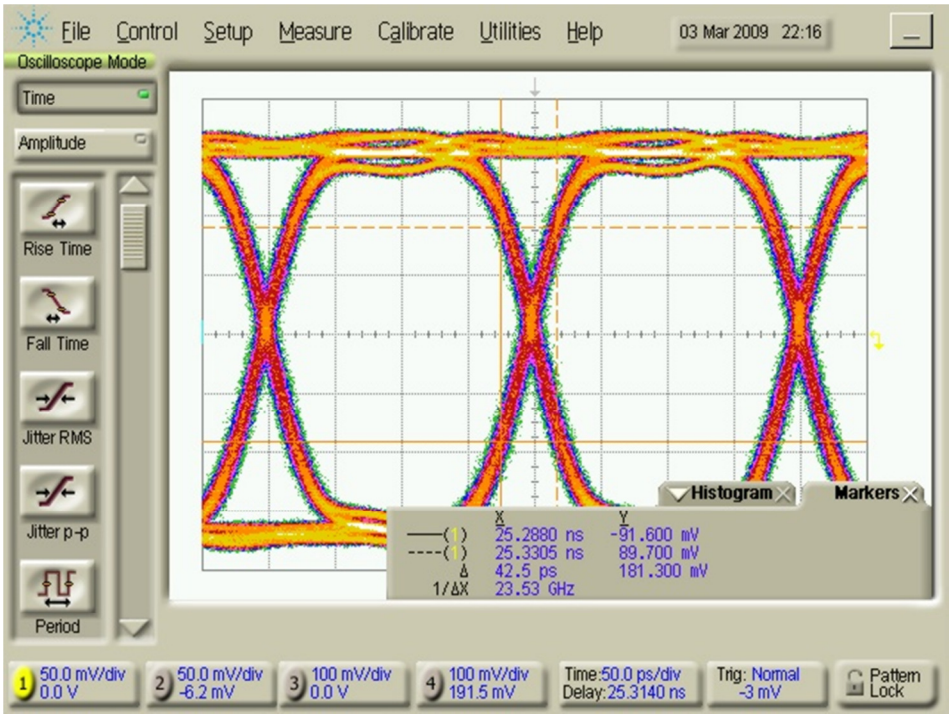


Figure 8.3 Core logic block output rise time measurement @ 5Gbps

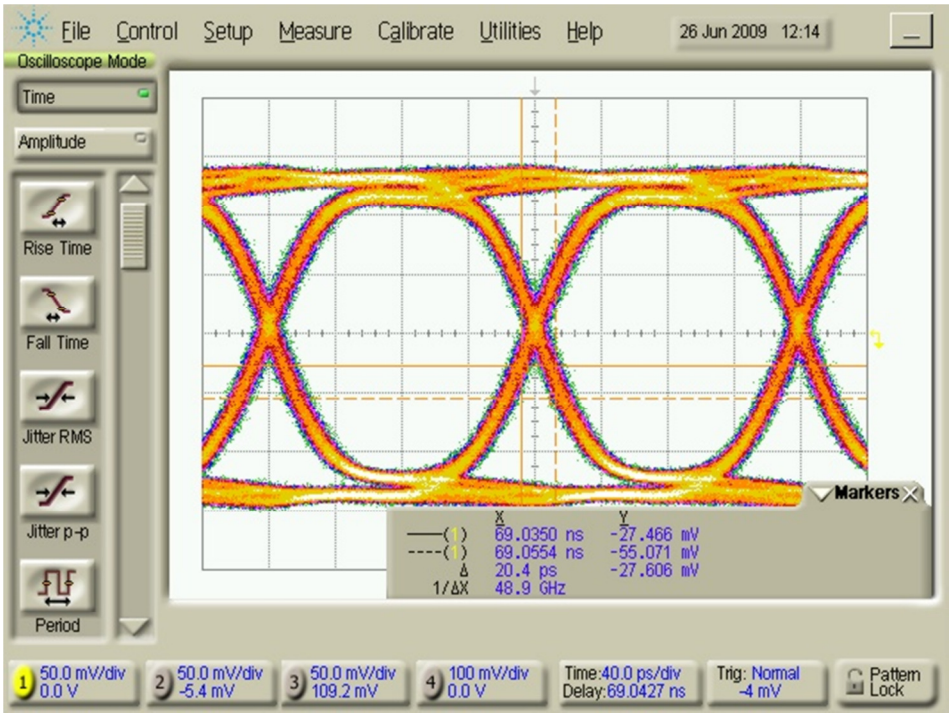


Figure 8.4 Core logic block output @ 6.25Gbps

Once operation of the RIO MGT was established at its specified maximum reliable rate, the output limits were pushed for further performance. Figure 8.5 shows output of the core logic block at 9Gbps. At this speed, wide open eyes are shown. However the p-p voltage swing of this signal was slightly limited to achieve this speed.

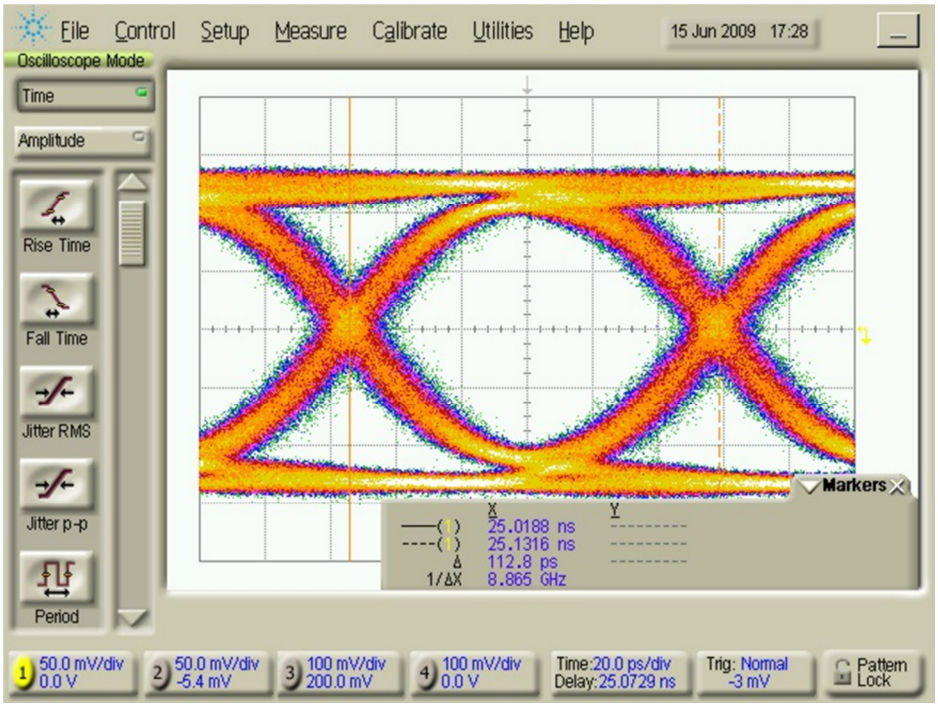


Figure 8.5 Core logic block output @ 9.00Gbps

Figure 8.6 shows a jitter measurement on the 9Gbps signal. Jitter is measured to be ~30ps (p-p). Although jitter has increased from the 6.25Gbps output signal, the added jitter is well within acceptable limits. As mentioned above, this data rate is in excess of the maximum reliable output specified by the FPGA manufacturer. However, these results were produced consistently numerous times by providing a clean low-jitter clock input and filtered low-noise (laboratory grade) power sources. Given the same quality inputs, these results should be reproducible in other environments, i.e. production testing.

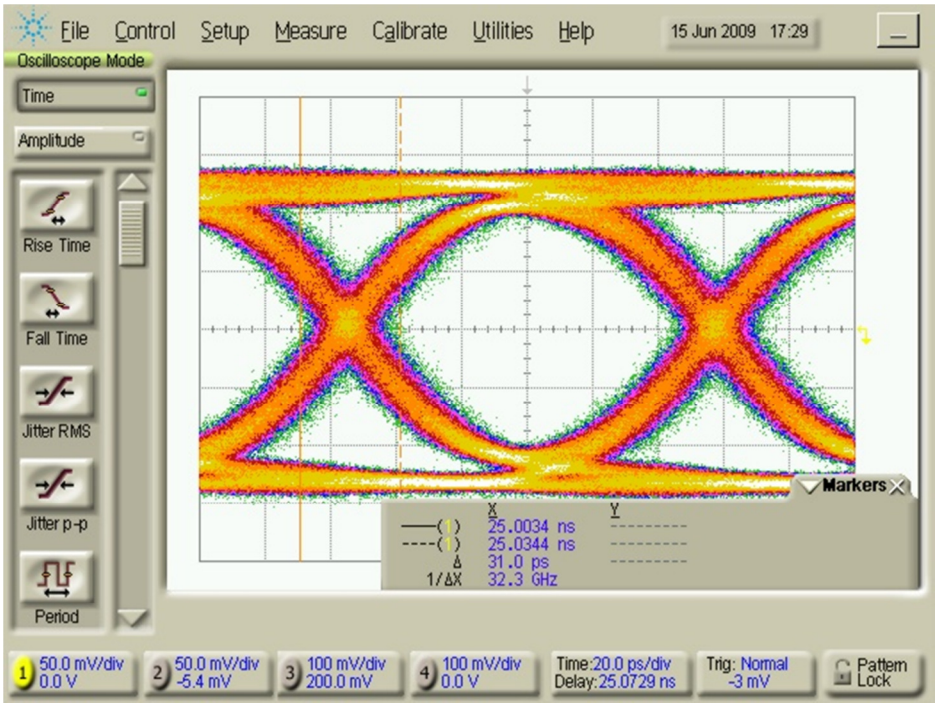


Figure 8.6 Core logic block jitter measurement @ 9.00Gbps

After functionality of the core logic block was demonstrated at 9.0Gbps, it was further pushed to determine its maximum output capabilities. Figure 8.7 shows output from the core logic block at 10Gbps. At this speed jitter is measured at 38ps (p-p). The eyes shown are open, but starting to close off. 38ps of jitter on a 10Gbps - 100ps bit period signal produces a roughly 0.60UI eye opening. This signal may be used for certain testing applications. Pushing the test module to further speeds produced unstable results.

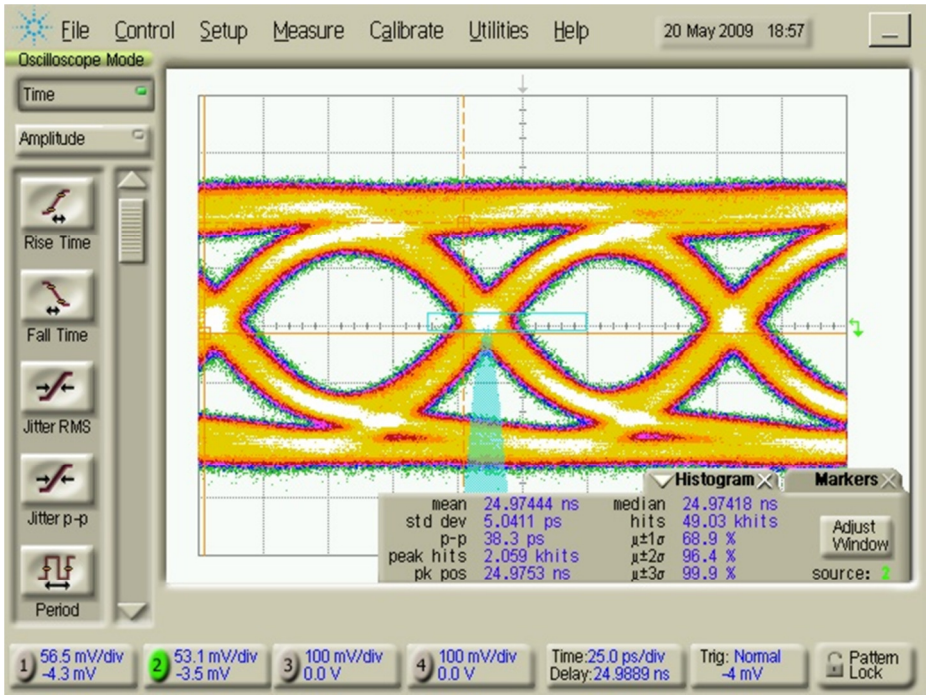


Figure 8.7 Core logic block output @ 10.00Gbps

8.2 High-speed signal multiplexing –characterization

In this section the results of the high-speed signal multiplexing application are presented. The setup to take measurements of the performance characteristics of the high-speed multiplexed signals is similar to that as described in the previous section using an oscilloscope. However, in this case, additional logic is used to synthesize signals produced by the core logic block as described in Chapter 5. The core logic block is used to produce two RIO MGT high-speed signals. These signals are available after they have passed through a 2-to-1 fan-out multiplexor. One output of the fan-out multiplexor is routed to ultra-precision drivers which sharpen the signals edges. These signals are then passed through an InP XOR gate which is used to multiplex the signals. An overview of the signal path and test setup is show in Figure 8.8. Given the signals are offset in time by

half a bit period; a double output data rate is produced by the InP XOR gate. Timing offsets of the RIO MGT signals are controlled by using clock delay chips in the core logic block, also described in Chapter 5.

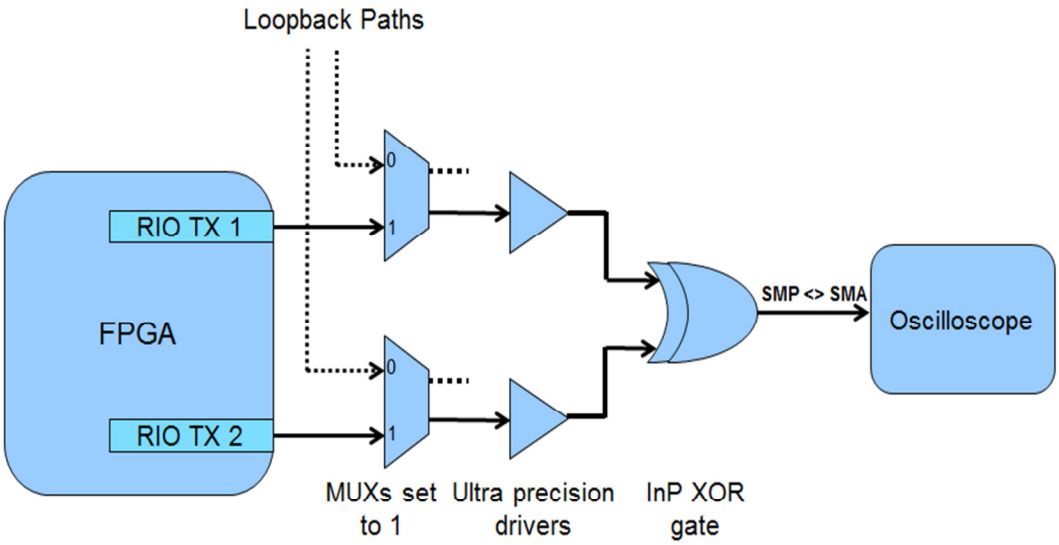


Figure 8.8 Test setup to measure high-speed signal multiplexing performance

Figure 8.9 shows a multiplexed signal at 10Gbps. This signal is produced by using two 5Gbps RIO MGT signals and appropriately offsetting them in time. The output signal exhibits wide eye openings. Jitter as measured on the signal, is 32ps. However much of the jitter can be attributed to the input signals from the FPGA. The manufacturer of the XOR gate estimates the gate adds <10ps of data dependent jitter [118]. This is an appropriate estimation as the RIO MGT input signals at 5Gbps measured 20ps (p-p) jitter. The performance demonstrated by the XOR gate at 10Gbps is within manufacturer guidelines and can be seen up to a speed of 13Gbps per manufacturer specifications.

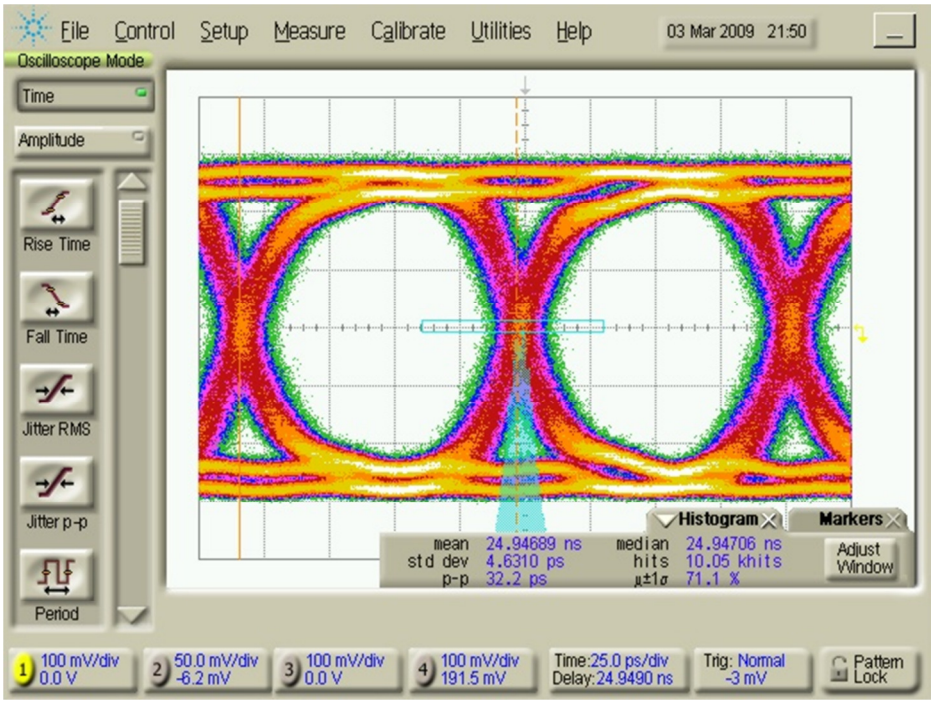


Figure 8.9 High-speed signal multiplexing output @ 10.00Gbps

Figure 8.10 shows a rise time measurement of a 10Gbps multiplexed signal. Rise time for this signal can be seen to be 24ps, which is typical for InP technology and in line with manufacturer specifications. The manufacturer also indicates a faster fall time, which is typical for InP technology [118]. Faster fall times for this signal can be seen in the figure above.

Figure 8.11 shows a multiplexed signal at 15Gbps. This signal is produced by using two 7.5Gbps RIO MGT signals and appropriately offsetting them in time. This test was done to demonstrate how far the performance of the XOR gate could be stretched. 15Gbps is beyond manufactures reliable guidelines of 13Gbps, however output is still produced. The results show data eyes are still open, however closing off. Jitter is measured around 35ps (p-p), which produces a less than 0.5UI open eye.

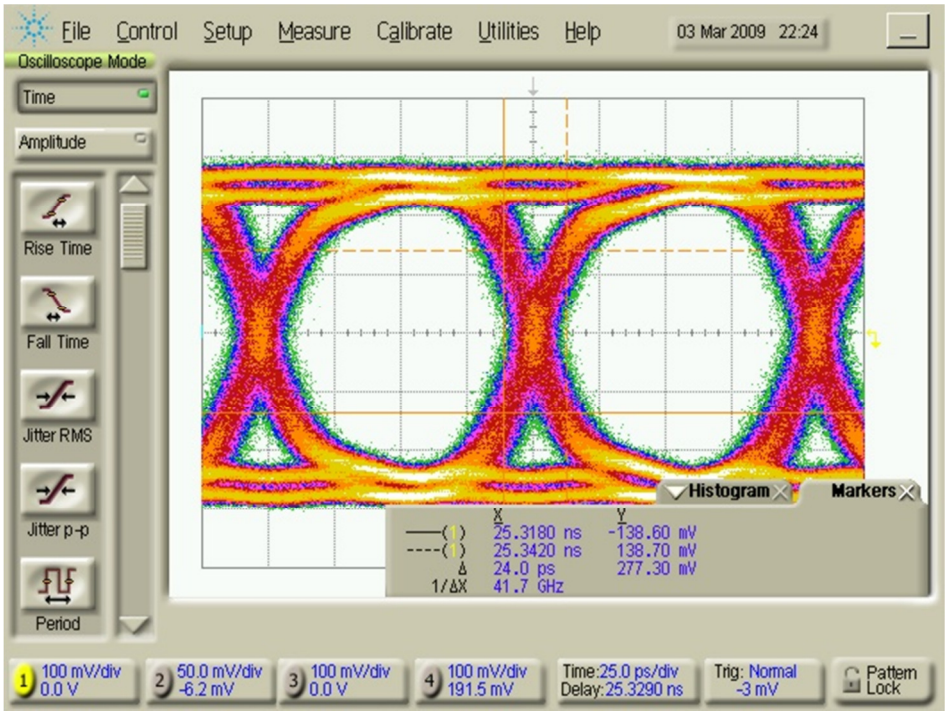


Figure 8.10 High-speed signal multiplexing rise-time measurement @ 10.00Gbps

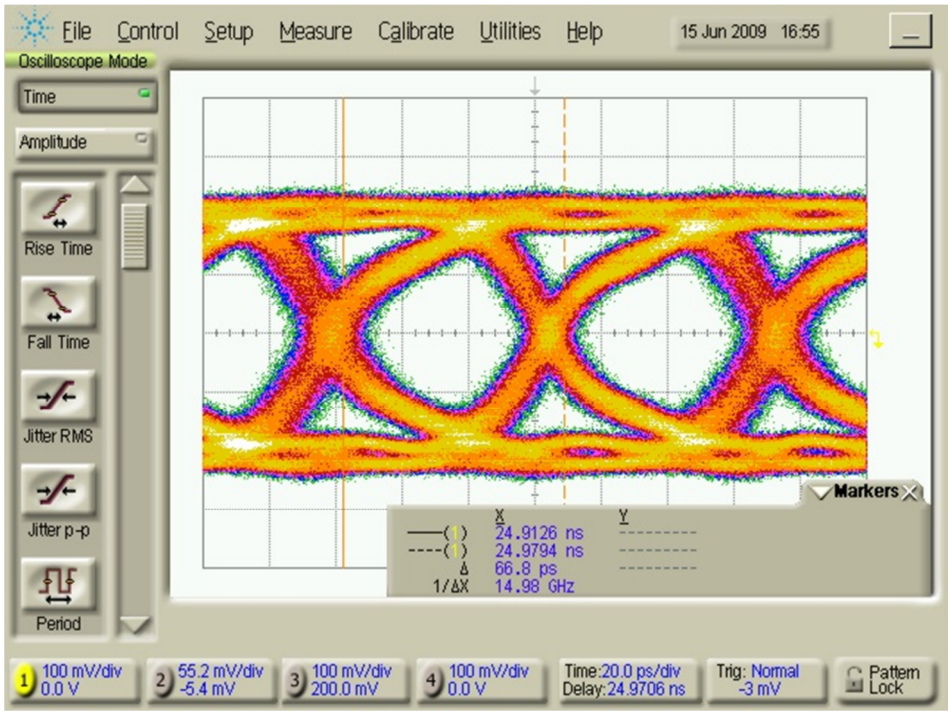


Figure 8.11 High-speed signal multiplexing output @ 15.00Gbps

Figure 8.12 shows multiplexed output from the XOR gate at 16Gbps. It is similar to the output at 15Gbps, but mainly presented to show the upper limits of the XOR gate. Jitter on this signal is measured to be approximately the same as the 15Gbps signal shown above at 35ps (p-p). The jitter is mainly dominated by the RIO MGT input signals which measured a jitter of 28ps (p-p). Furthermore, on a 62.5ps bit period signal (16Gbps), a rise time of 24ps becomes inefficient. This can be seen in the figure, as full amplitude on the high side can only be sustained for a short period of time. Pushing the XOR gate further did not produce useful results.

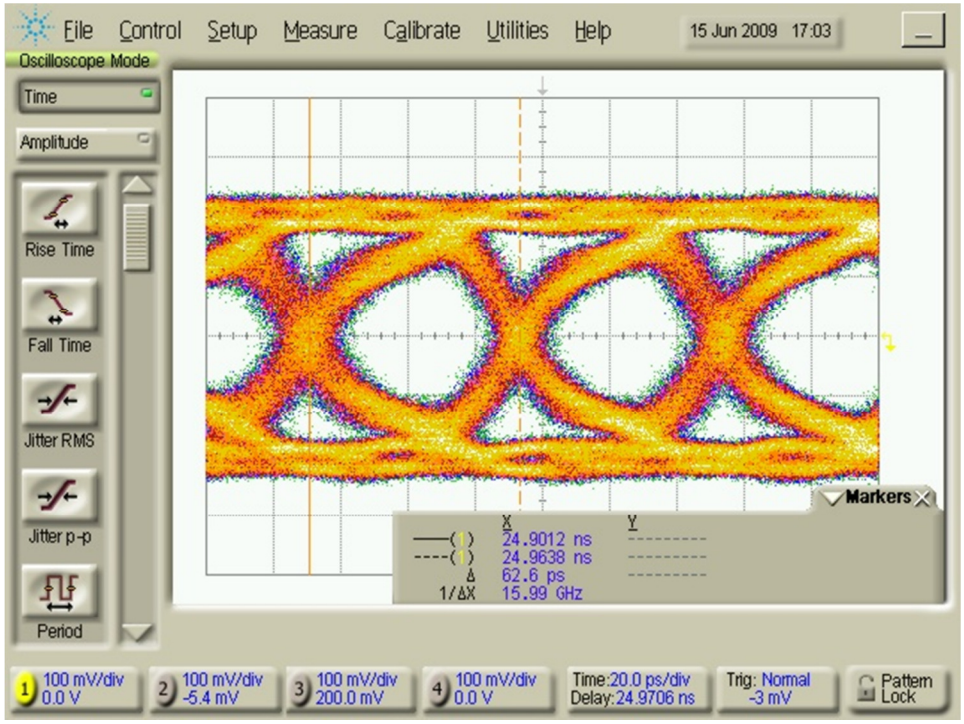


Figure 8.12 High-speed signal multiplexing output @ 16.00Gbps

8.3 Loopback Path – characterization

In this section the results of the loopback path on the test module are discussed. The test module is designed with a high-speed loopback path to allow for loopback

testing of devices. Test signals enter the test module through SMP connectors. The signals are passed through a high-speed fan-out buffer, with one set of differential outputs routed to the loopback path and the other directly to the core logic block. The loopback path continues to a 2-to-1 fan-out mux, where it can be selected as the output signal, thus creating a loopback path as discussed in detail in Chapter 5.

The loopback path is characterized using two experiments. In the first experiment, the output of the core logic block is used as shown in Figure 8.13. The output of TX1 is set to use the output signal produced by RIO MGT TX1 in the FPGA by setting the upper mux (as pictured in the figure) to 1. TX1 is then physically connected to RX2 using a SMP cable. The output of TX2 is then set to use the loopback input by setting the lower mux to 0, thus establishing a loopback path between TX1 and TX2. The output of TX2 is connected to an oscilloscope, where it is analyzed.

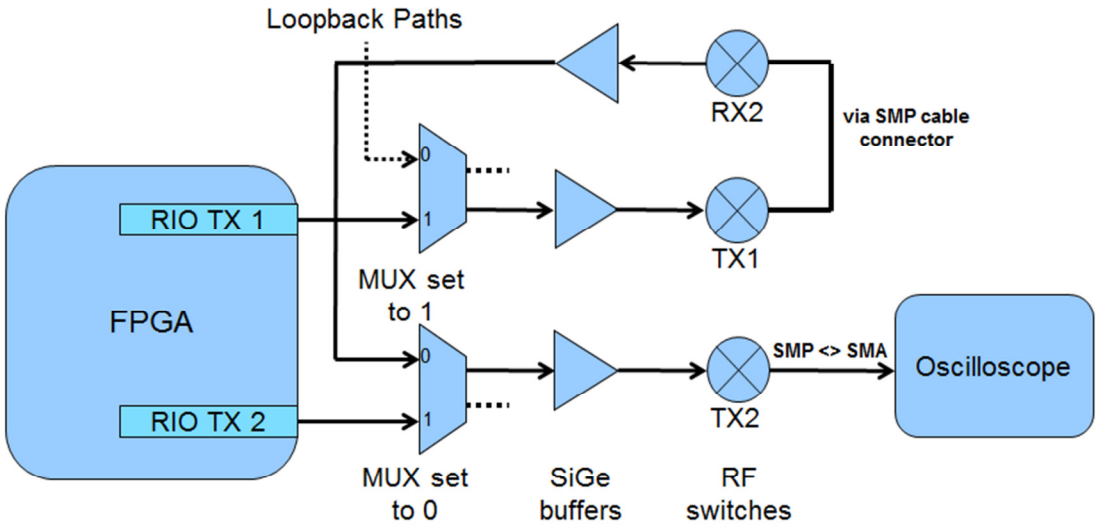


Figure 8.13 Test setup to measure loopback path using RIO MGT signals.

Figure 8.14 shows results of the loopback path using a RIO MGT signal at 6.25Gbps. The upper portion of the figure shows the signal before it enters the loopback path for reference purposes, while the lower portion of the figure shows the signal after it has travelled the loopback path. The reference signal at 6.25Gbps measures jitter at ~22pp (p-p). After going through the loopback path, jitter on the same signal is measured at ~42ps (p-p), thus the loopback path at 6.25Gbps is adding ~20ps (p-p) jitter to the signal. Nonetheless, the output signal exhibits wide open eyes with a ~0.75UI opening.

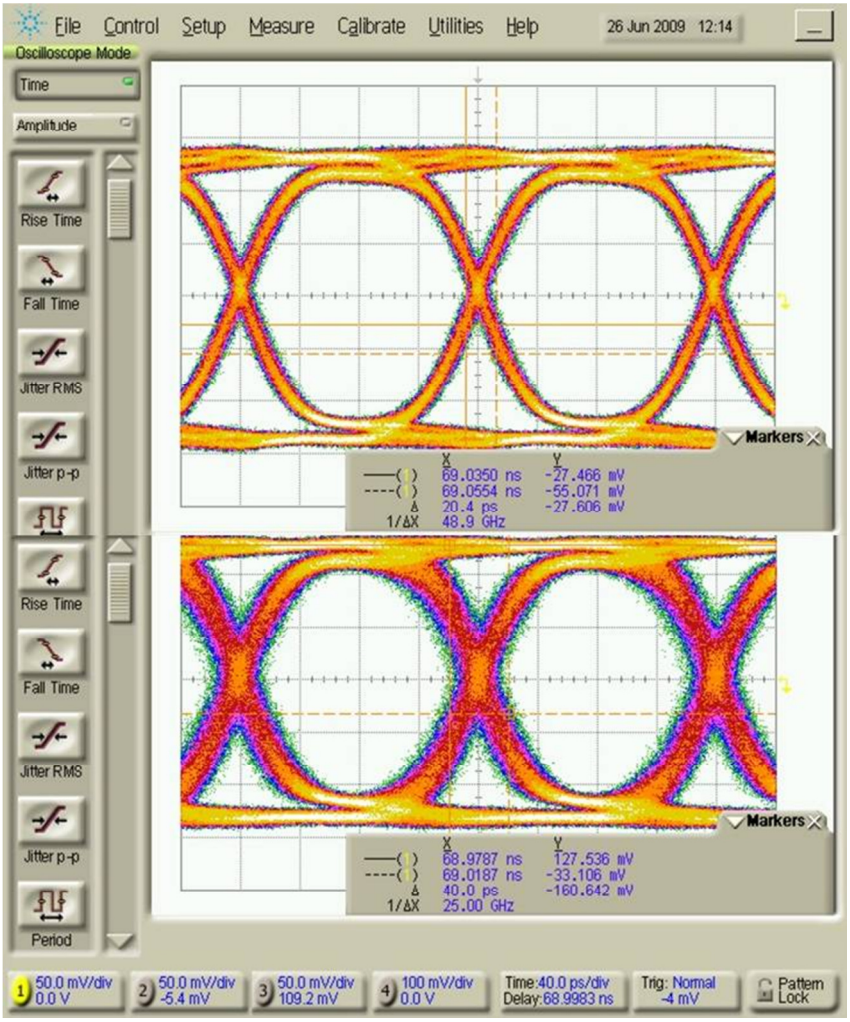


Figure 8.14 Loopback path results using RIO MGT @ 6.25Gbps

Figure 8.15 shows a similar plot of the loopback path results using RIO MGT input signal at 9.0Gbps. The reference input signal is shown above and the output signal is shown below. The input signal measures ~28ps (p-p) of jitter, whereas the output signal measures ~46ps (p-p) of jitter. The jitter added to the signal at this speed is similar to jitter added at 6.25Gbps. However since the input signal already measures jitter of 28ps (p-p), adding another ~20ps (p-p) starts to close the data eyes as seen above.

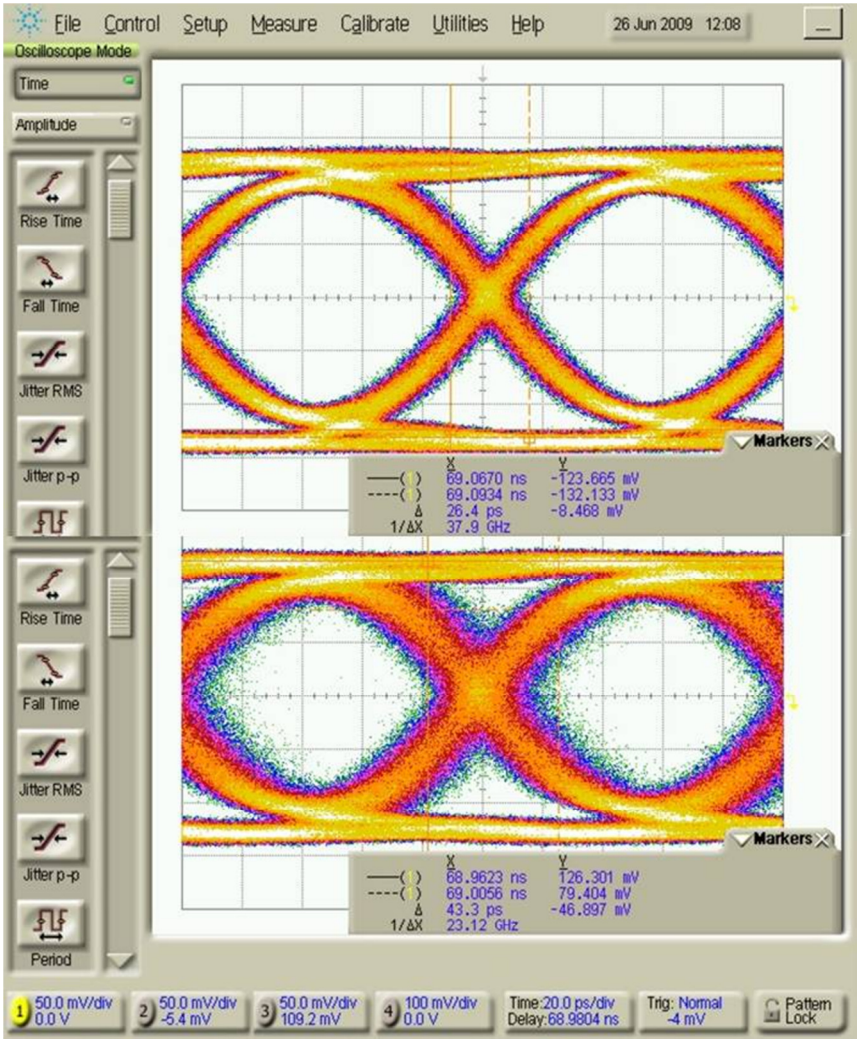


Figure 8.15 Loopback path results using RIO MGT @ 9.00Gbps

Higher speed input signals were used to determine the loopback path's limits. In Figure 8.16, a 10Gbps signal from the RIO MGT is input to the loopback path. The reference signal measures ~35ps (p-p) jitter. The lower portion of the figure displays the same signal output from the loopback. The output signal measures jitter above 50ps (p-p) with data eyes that are mostly closed (less than 0.50UI opening). Adding another 20ps of jitter in this case would not be an acceptable solution.

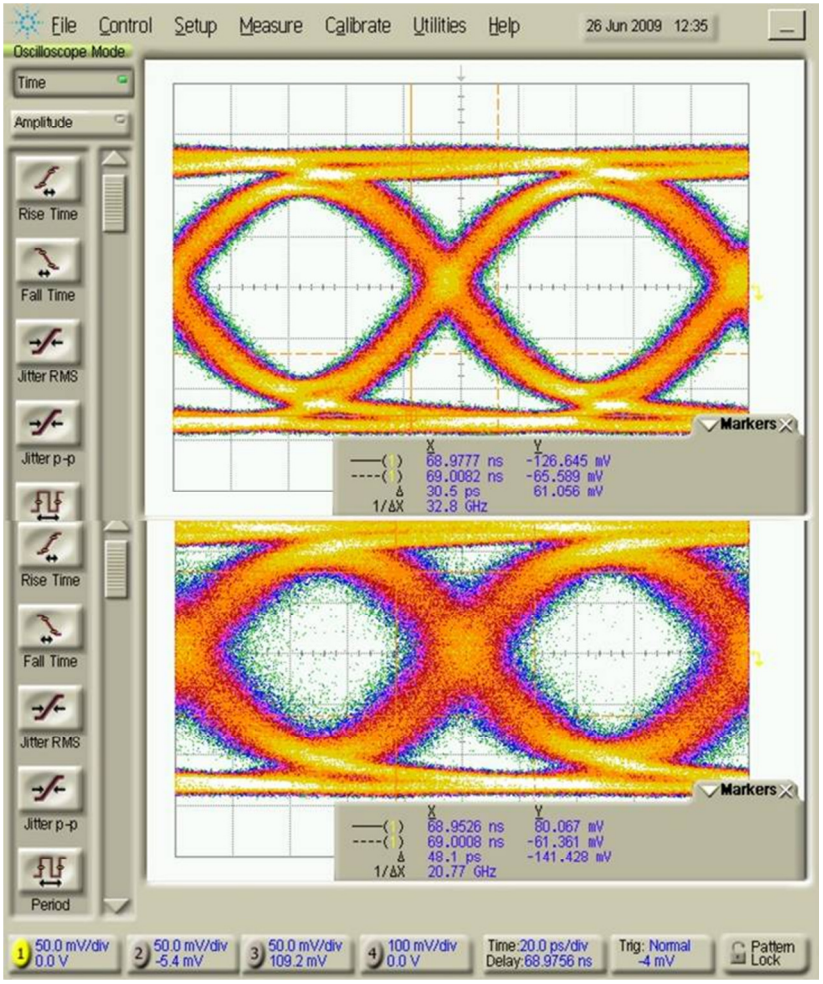


Figure 8.16 Loopback path results using RIO MGT @ 10.00Gbps

The above results indicated that using a low-jitter high speed signal would better demonstrate the performance characteristics. Since all the output signals produced by the test module measured at least 20ps (p-p) jitter, an external source was used. In [98] a serializer module is demonstrated that can output high-speed data up to 10Gbps with jitter under 20ps (p-p). This module was available to be used as an external source to test the test module's loopback path. The test setup for this experiment is similar to the above experiment and shown in Figure 8.17. Instead of the RIO MGT output from TX1, the external high-speed source from the serializer module is input into RX2. The mux on the TX2 path is set to choose the loopback path and the output signal measured with an oscilloscope. In this experiment, the mux, buffer and relay switch on TX1 path are bypassed. Each of these components has the propensity to add small amounts of jitter to the signal used, which accumulate and affect the final output signal. Thus by bypassing these components, lower jitter can be seen on the output of the loopback path.

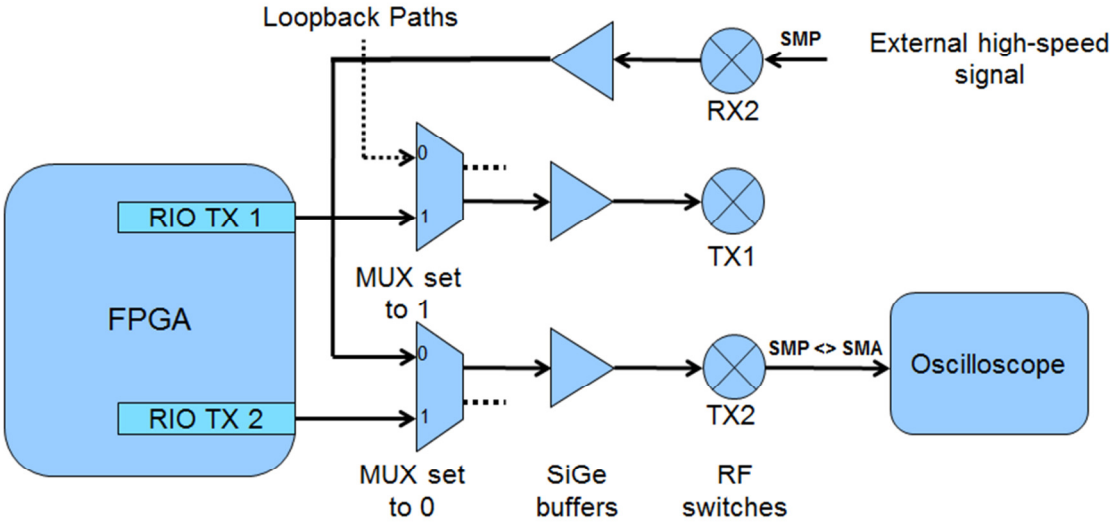


Figure 8.17 Test setup to measure loopback path using an external high-speed signal source.

Figure 8.18 shows results of the test module's loopback path using an external signal at 9.28Gbps. Jitter measured on the input signal is ~11ps (p-p), as shown in the upper portion of the figure. After this signal is output from the loopback path, jitter is measured at ~18ps (p-p). In this case, the loopback path on the test module is adding only 7ps of jitter to the input signal. The results of the loopback path using an external source at this speed are certainly promising.

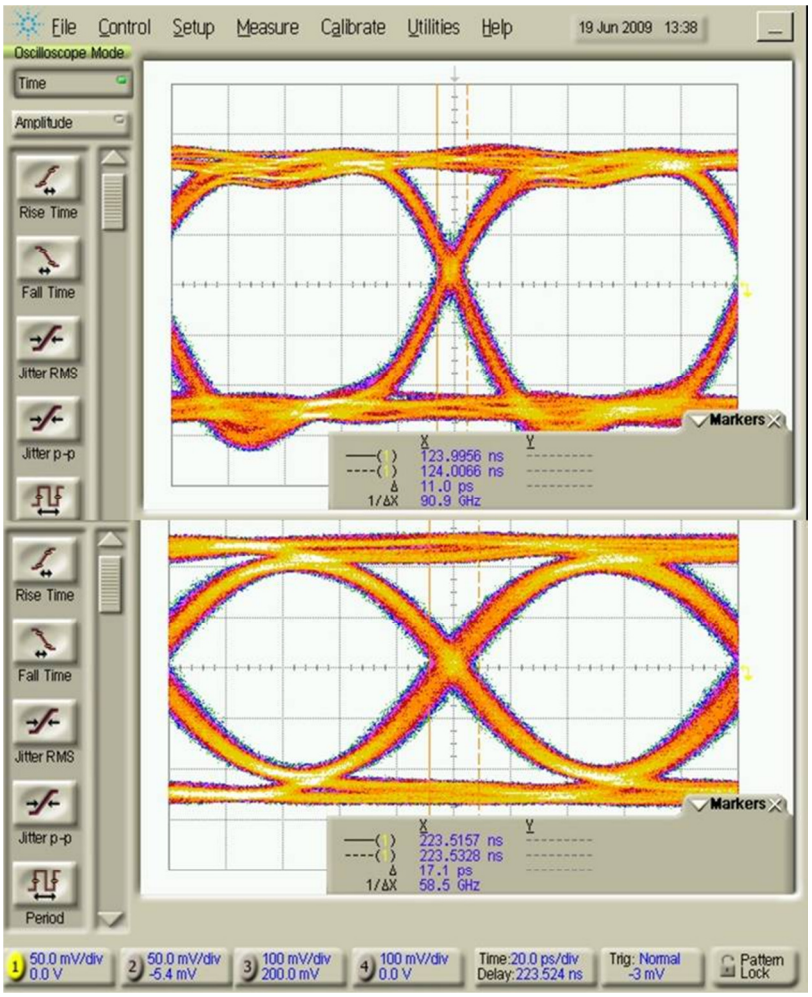


Figure 8.18 Loopback path results using external source @ 9.28Gbps

In the final loopback path experiment, an external source at 10Gbps is used as shown in the upper portion of Figure 8.19. Jitter on the input signal is measured at ~16ps (p-p). The same signal output through the loopback path measures ~32ps (p-p). At this speed, the loopback path is adding 16ps of jitter. In this case the loopback path adds lower jitter compared to using a RIO MGT input signal at the same speed. However, the output signal is distorted and not symmetrical. These results may not be ideal for most test applications, but can be used for some, such as at-speed testing.

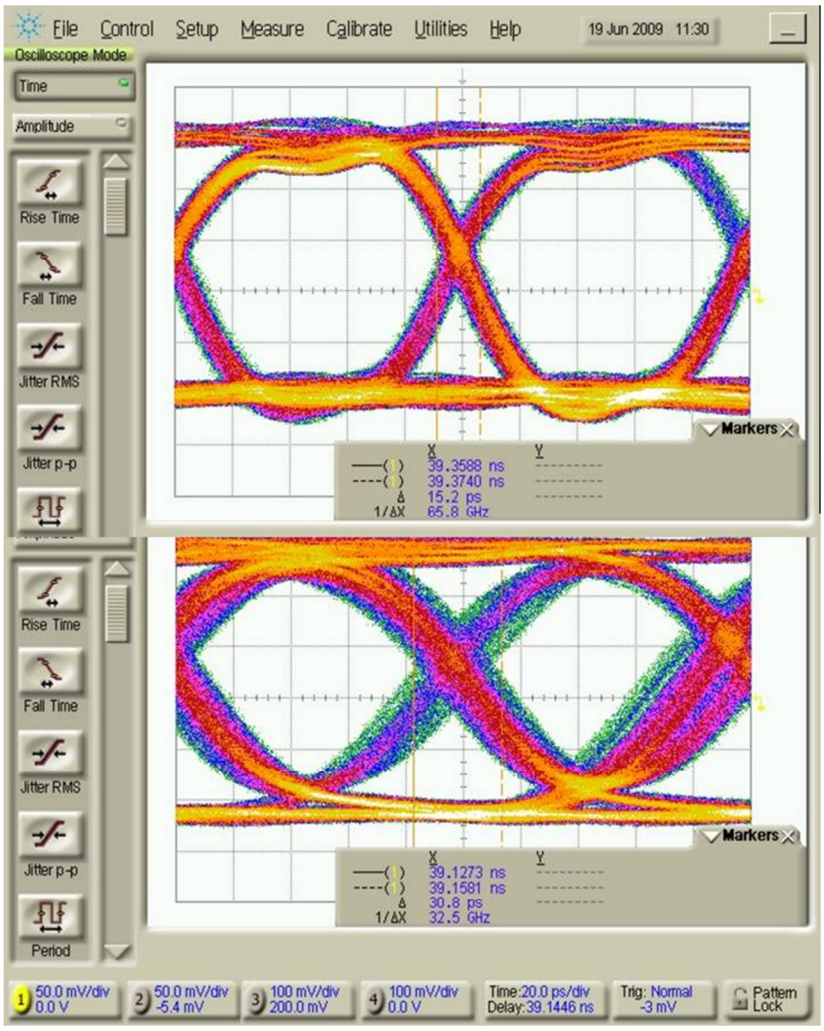


Figure 8.19 Loopback path results using external source @ 10.0Gbps

8.4 Amplitude Adjustment – characterization

Amplitude adjustment features are available on two channels of the test module – TX1 and TX2. This is achieved by using variable output drivers on the output signals and discussed in detail in Chapter 5. By adjusting a control voltage input to the driver, the output amplitude can be varied. The test module is designed with a potentiometer for the purpose of adjusting the control voltage. The output driver is capable of varying the output amplitude from 100-700mV, when input control voltage range is between 2.375-3.465V [119]. A simple lab experiment is setup to test the amplitude adjustment performance of the test module as shown in Figure 8.20. A RIO MGT signal is output onto TX1 and connected to an oscilloscope. Control voltage on the buffer in the TX1 path is systematically incremented while the output signal amplitude is monitored on the oscilloscope.

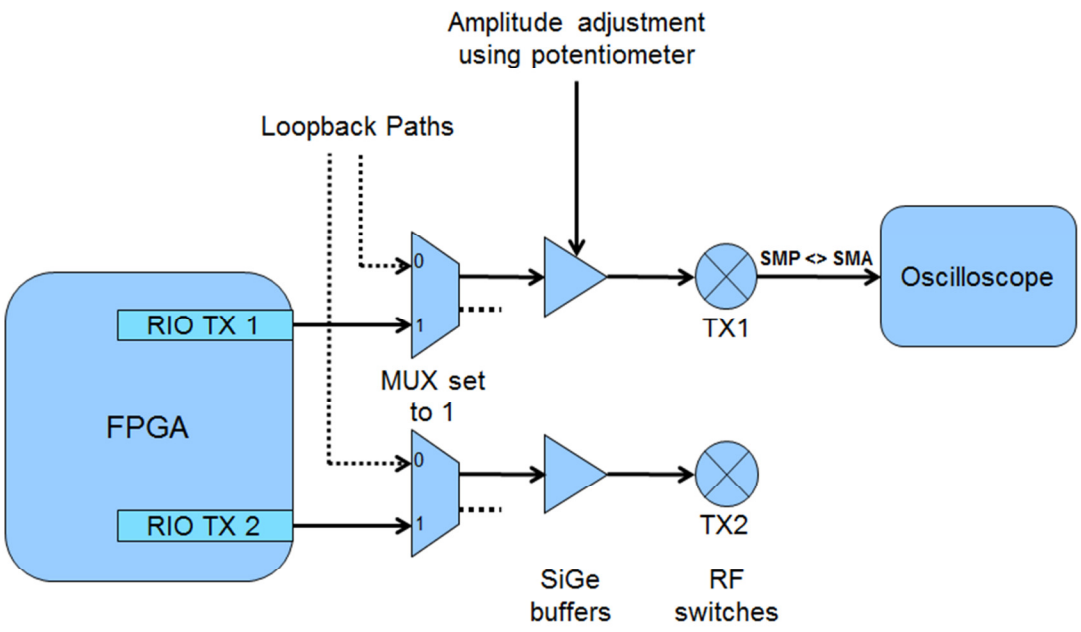


Figure 8.20 Test setup to measure amplitude adjustment performance of test module.

Figure 8.21 shows a plot of the measured amplitude of an output signal from TX1 versus the control voltage applied to the output driver. The input signal used is one produced by a RIO MGT channel at 6.25Gbps. The p-p amplitude measured varied from 70mV to 725mV and is very similar to the range specified by the device manufacturer. The driver varied the output amplitude while the control voltage was within 1.70-3.00V, which is slightly offset from the manufacturer's data sheet. Regardless, the device did provide the variable amplitude range as specified and is sufficient for this function on the test module. The control voltage offset can be noted and proper adjustments made to achieve the desired amplitude in a testing environment.

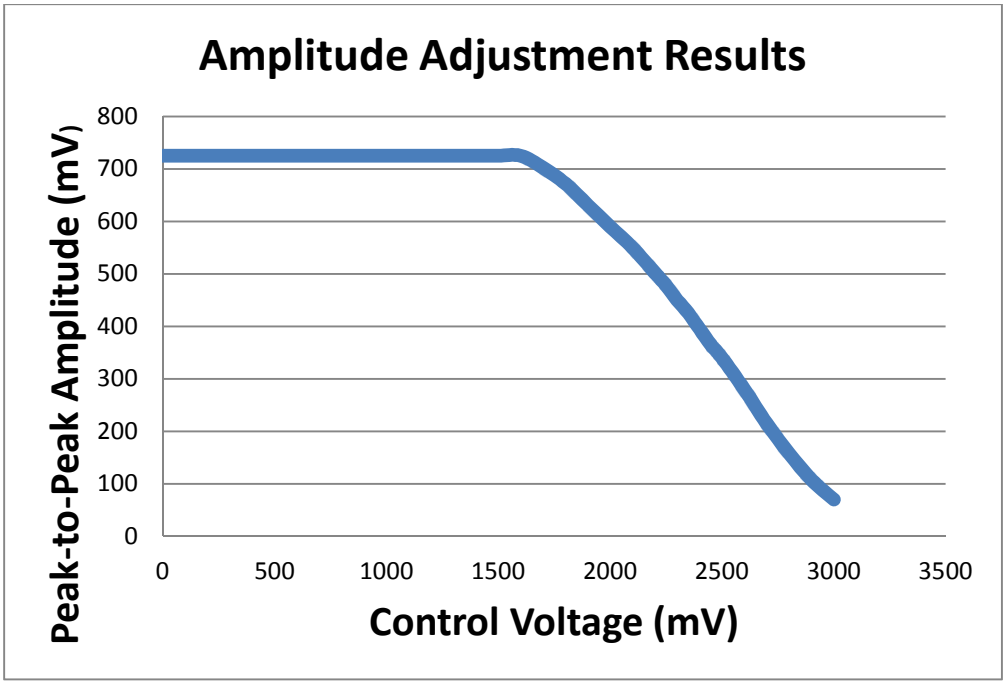


Figure 8.21 Amplitude adjustment results from TX1

8.5 Timing/Skew Adjustment – characterization

Timing/skew adjustment on the test module is available on two channels, TX1 and TX2. Adjustments are made using a programmable delay chip as described in Chapter 5. The RIO MGTs require a clock input, which also determines the output phase. Therefore, by adjusting the phase of the input clock to the RocketIO, the phase of its output data can be adjusted. This feature can be tested by setting various delay values on the programmable delay chips, and monitoring the change in phase of the output signals. An overview of the test setup is shown in Figure 8.22.

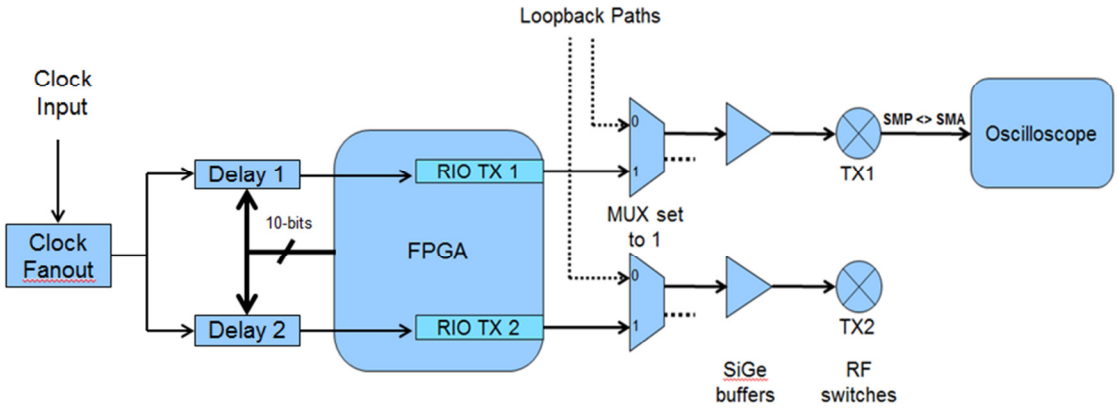


Figure 8.22 Test setup to measure timing/skew adjustment.

Figure 8.23 shows finite timing adjustment using the programmable delay chip. Edge 1 is the reference edge at 0ps. The device is then programed to delay the signal 10ps, and the actual delay is 14ps as seen in edge 2. Edge 3 delayed by 38ps is produced by programming 20ps of delay. Edge 4 is programmed to be delayed 30ps, a combination of 20+10ps. Its actual delay is 52ps (38+14ps). A programmed delay of 40ps produces an actual delay of 38ps, and overlaps edge 3. Edge 5 measured at 66ps delayed, is

programmed to be 50ps delayed. These delay non-linearities are well documented in the programmable delay data sheet [120]. Also, Chapter 5 discusses this issue in detail as well and calibration methods to obtain better results.

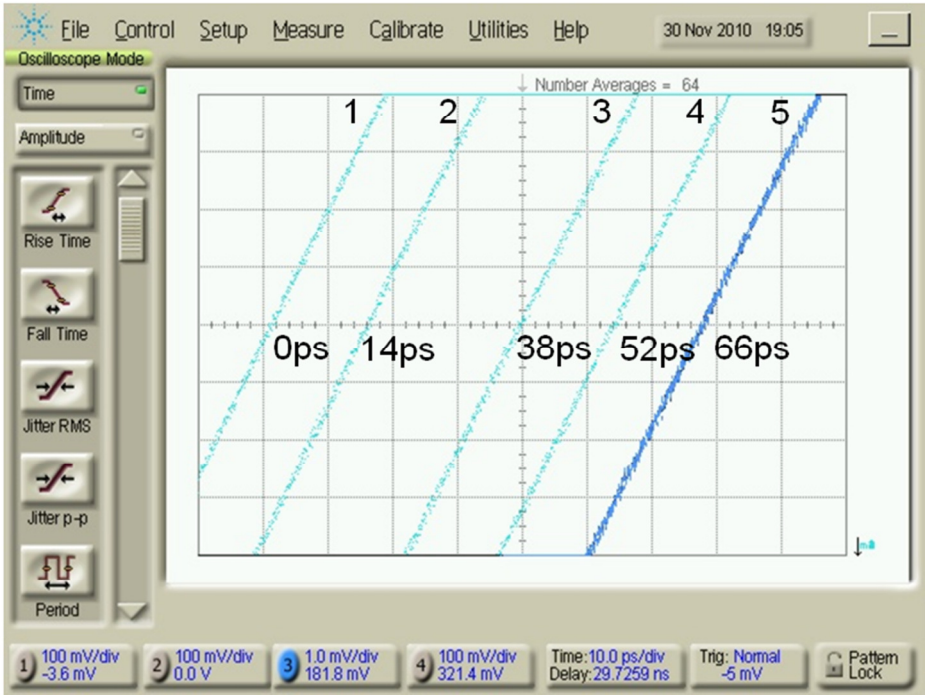


Figure 8.23 Finite timing adjustment with delay chip

In addition to finite timing adjustment, the programmable delay chips are capable of fine analog timing adjustment using its F_{TUNE} input as discussed in Chapter 5. Figure 8.24 shows the performance of the fine adjustment. A total range of about 60ps is achieved using this method. A table can be constructed measuring the voltage applied on the F_{TUNE} input and the delay measured. Using this table in conjunction with the calibrated values for the finite delay discussed in Chapter 5, accurate and finite timing adjustment can be achieved from the range of 0 to 10ns.

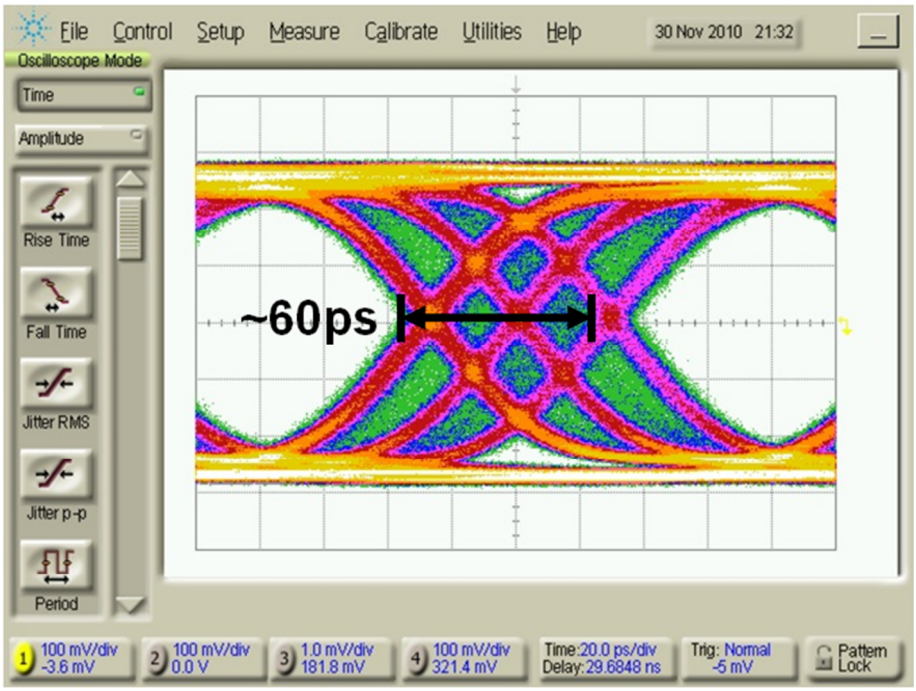


Figure 8.24 Timing adjustment using analog F_{TUNE} input on delay chip

8.6 Jitter Injection – characterization

The test module allows jitter to be injected onto its output signals through two methods. The first method by which jitter can be injected is directly into the output signal through the final stage driver. The second method to inject jitter into the system is through the clock source, i.e. the programmable delay chip. Both options are discussed in detail in Chapter 5. The second method allows a larger range of jitter injection and is preferable to the first method. In this option an external signal generator is used as a jitter injection source to input jitter directly onto the programmable delay chips as shown in Figure 8.25.

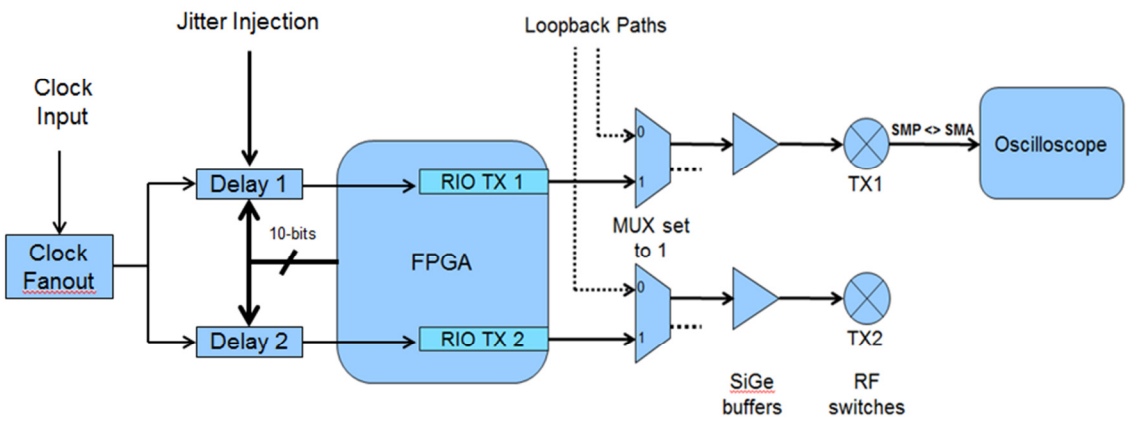


Figure 8.25 Test setup to demonstrate jitter injection.

The signal generator can produce a variety of signals with various amplitudes and frequencies. It can also produce a plain noise signal. Various signals result in different amounts of jitter injection. Figure 8.26 shows jitter injection using a 0.5V amplitude noise signal, which produces 16ps of jitter. Similarly, Figure 8.27 shows 52ps of jitter added by increasing the noise amplitude to 2.0V. Using a sine wave instead of plain noise, higher jitter can be injected. Figure 8.28 shows 30ps of jitter injected using 0.5V amplitude 20MHz sine waves. Similarly, Figure 8.29 shows 81ps of jitter injected using a 2.0V amplitude 20MHz sine wave. All these signals were applied onto to 6.25Gbps data signal that measured a base jitter of 27ps (p-p).

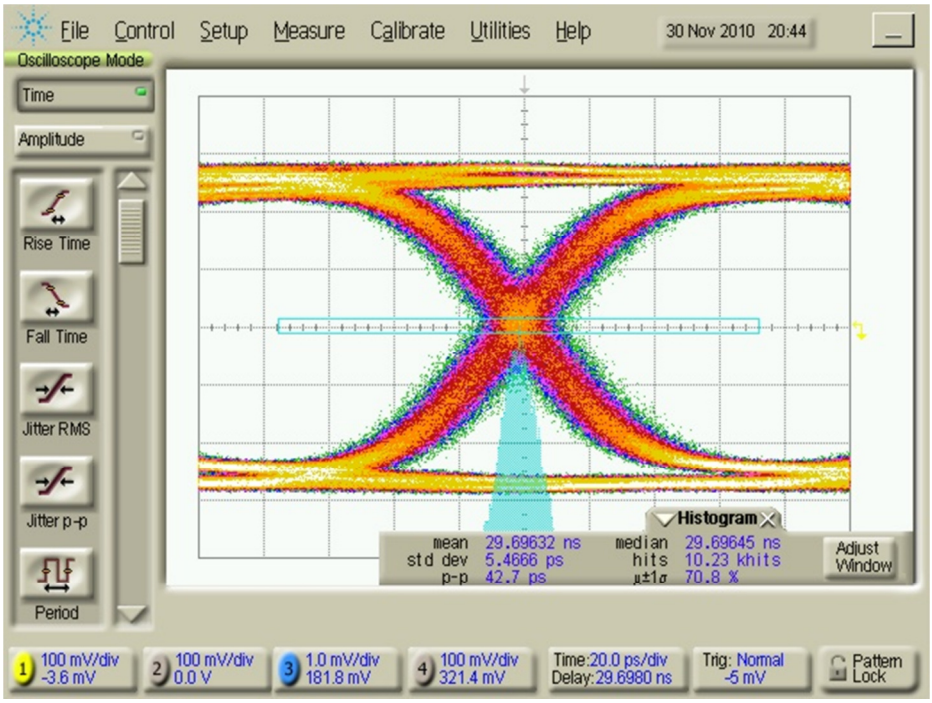


Figure 8.26 0.5V noise signal injecting 16ps (p-p) of jitter

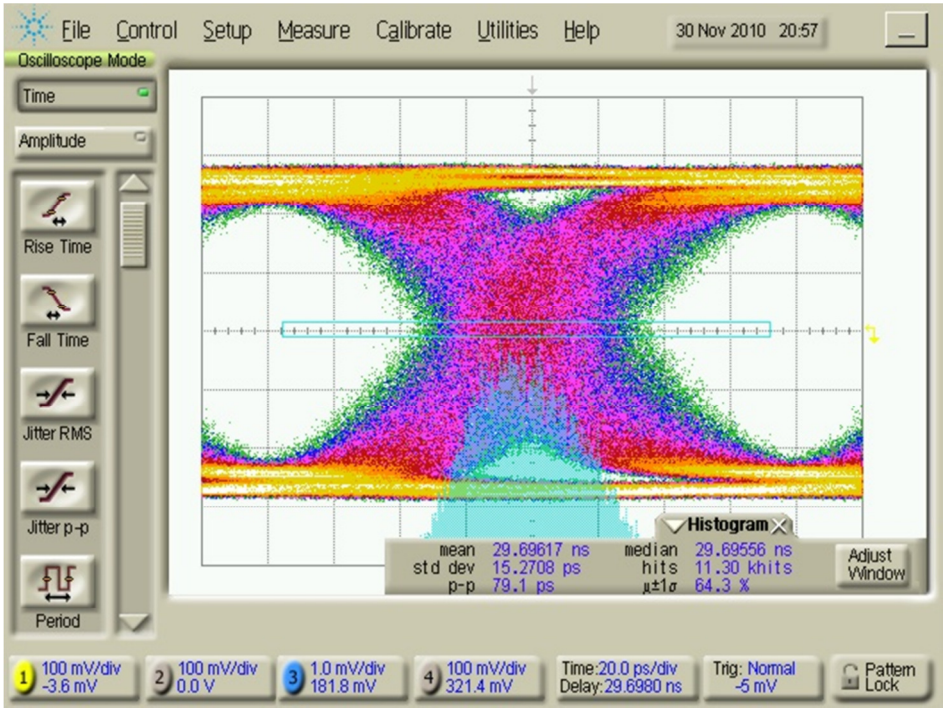


Figure 8.27 2.0V noise signal injecting 52ps (p-p) of jitter

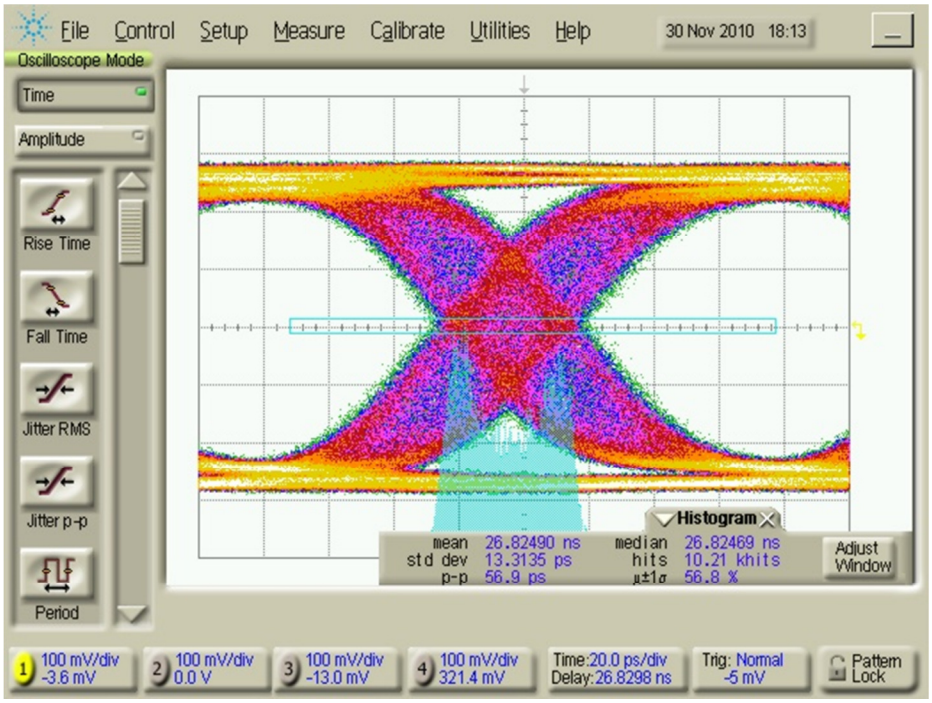


Figure 8.28 0.5V 20MHz sine signal injecting 30ps (p-p) of jitter

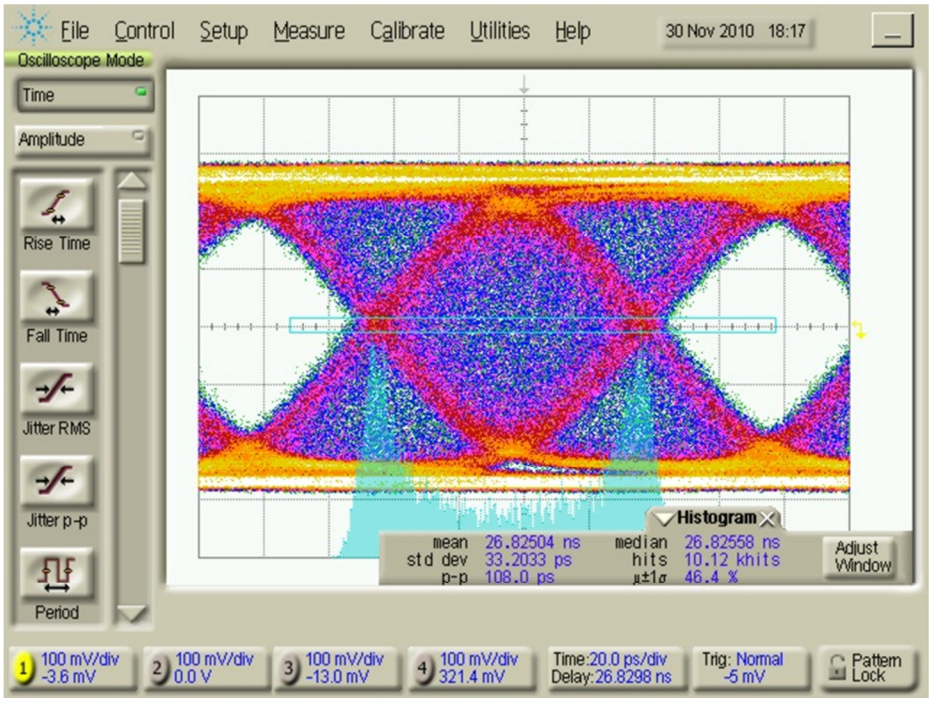


Figure 8.29 2.0V 20MHz sine signal injecting 81ps (p-p) of jitter

Injected jitter is a function of both the amplitude as well as the frequency (when not plain noise) of the added signal. Therefore signals of various amplitudes and frequencies were injected into the system, and the total jitter measured. Figure 8.30 shows the relationship between signal characteristics and total p-p jitter, while Figure 8.31 shows the same relationship with the standard deviation of the jitter. The same reference signal running at 6.25Gbps with a base jitter of 27ps (p-p) was used. Figure 8.31 shows linearly increasing jitter with signal amplitude up to about 1.25V. After this point the added jitter starts to taper off, also seen in Figure 8.30. The highest amount of jitter added is about 81ps (p-p) (shown in Figure 8.29), therefore the range of jitter injection using this method is 0-81ps (p-p). Data from Figure 8.30 and Figure 8.31 can be used to created tables that allow the finite injection of jitter close to the picosecond scale. These tables can be programmed into the software interface of the test module, making jitter injection seamless to the user.

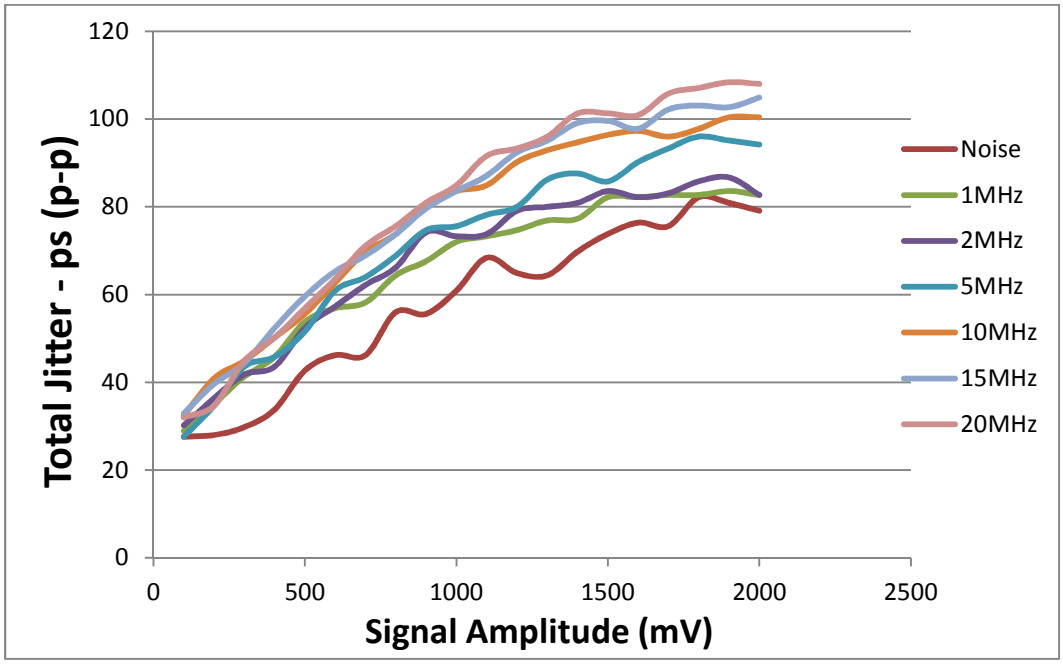


Figure 8.30 Jitter injection measurements (p-p)

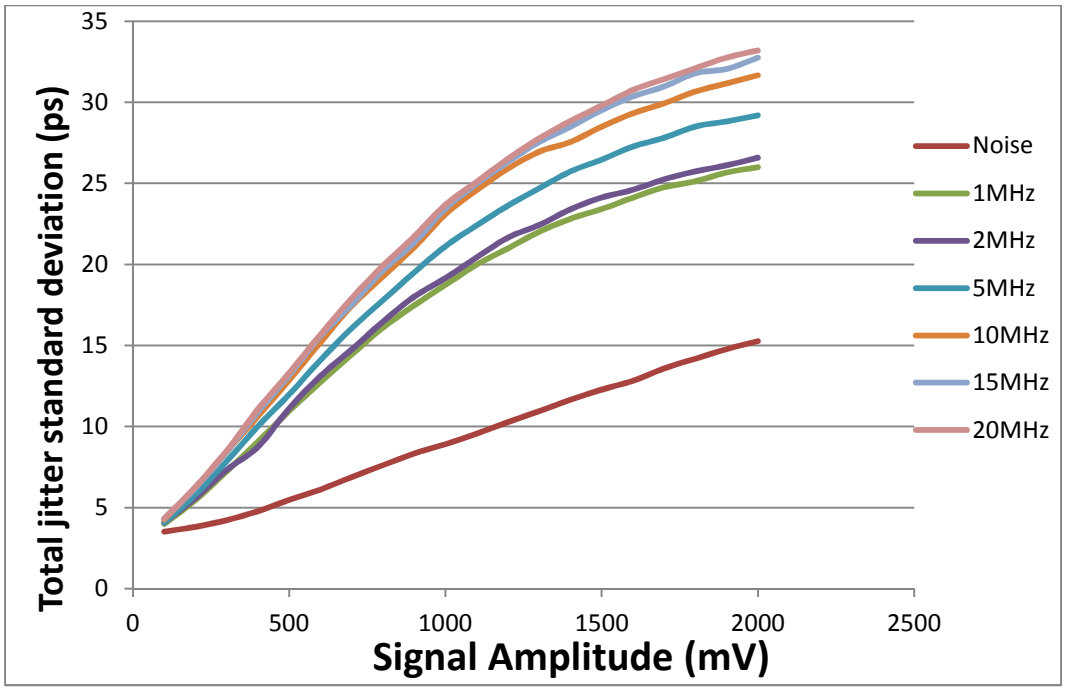


Figure 8.31 Jitter injection measurements (standard deviation)

8.7 Low-speed/parametric testing - characterization

The test module has special provisions for direct propagation of low speed signals from an ATE. This is achieved by the use of RF relays, which when are in the open position, can relay signals from an ATE to the output. This application is discussed in detail in Chapter 5.

Testing the low-speed path is a simpler experiment. The purpose of the low-speed path is to provide a connection from the 40-pin connector on the test module to the RF relays (see Figure 5.12). Therefore, essentially by checking this path for continuity, it can be asserted that the path functions properly. Furthermore, it is expected that the path will be used for parametric measurements by the ATE. Thus the low-speed path should not alter the input signal. This can be tested by measuring the current and voltage of an

input signal to the low-speed path and comparing it to the output signal at the RF relays. Given the values are within sufficient tolerance limits, it can be determined that the low-speed path performs are designed. These tests were conducted and low-speed signal path's operation was verified.

8.8 Results Summary

The results presented in this chapter demonstrate satisfactory performance of the test module. The high-speed signal generating capabilities of the core logic block greatly exceeded expectations. The main component of the core logic block, the FPGA was expected to produce reliable results up to 6.25Gbps as per the manufacturer's guidelines, however was pushed to produce results up to 10.0Gbps. The high-speed multiplexing feature produced results up to 16Gbps, at which jitter was dominated by the input signals produced by the core logic block. The test module does have the ability to multiplex external signals. Therefore, if two lower jitter external signals, which are aligned precisely for multiplexing, are provided to the test module, better results are potentially attainable. The loopback path demonstrated solid capabilities of reliably propagating high-speed signals slightly under 10Gbps. Amplitude adjustment and timing/skew adjustment results demonstrated the proper functioning of these features. Jitter injection was demonstrated with a range of 0-81ps (p-p). Furthermore, the low-speed path on the test module performed as designed. Therefore, the results presented in this chapter provide sufficient evidence that by using a multi-GHz FPGA based test module according the approach described within this research, ATE performance can indeed be extended in a feasible and useful manner.

CHAPTER 9

CONCLUSIONS

9.1 Summary

The objective of this research was to develop methodologies for extending ATE performance capabilities into the multi-GHz range using FPGAs. In Chapter 1, the motivation for extending ATE performance was discussed, and hence the purpose of this research was established. The history of ATE development and recent advancements was reviewed in Chapter 2. This chapter also examined the progression of semiconductor test and how various methods of design and test are used to test today's complex electronics. Furthermore this chapter reviewed technology roadmaps in order to better understand industry trends and anticipate future challenges.

Chapter 3 reviewed research done to extend ATE performance into the GHz range. This research mainly consisted of the development of electronic test modules that use ATE resources to synthesize higher test performance metrics. Chapter 4 presented the preliminary research of this thesis that was motivated by the need to increase high-speed testing capability. However unlike the research presented in Chapter 3, the preliminary research presented in Chapter 4 focused on the development of a FPGA-based multi-GHz miniature tester capable of operating without the need for ATE resources. The research presented in this thesis extended the earlier work (presented in Chapters 3 and 4) by combining the ideas of modular extension of ATE and stand-alone test techniques using

FPGAs. This research focused on methods to enhance ATE performance using multi-GHz FPGAs to develop test modules capable of independent operation. These methods are explored and demonstrated by the design and characterization of a high-performance test module in Chapters 5-7.

Chapter 5 discussed the logical design and methodology of the test module. This module consisted to two blocks – the core logic block and the application specific logic block. The core logic block was the central component of the module. It was designed with a FPGA that allowed it to control test functionality and also provided an interface to the outside world. A multi-GHz Xilinx FPGA was used that contained RIO MGTs. These allowed the core logic block to produce multi-GHz signals. In addition to the FPGA, the core logic block contained a flash memory chip and a microcontroller. The flash memory chip was required to program the FPGA on power up. The microcontroller provided a USB interface to the FPGA such that it could be controlled using a PC. The core logic block required a clock input that was fanned-out to four programmable delay chips. These clock delay chips were used in the core logic block to adjust the timing skew of the output signals produced by the FPGA.

The application specific logic block took signals from the core logic block and either processed them further or simply passed them through to the DUT, based upon the application. In this research six applications were developed. The first application was high-speed signal multiplexing to produce DDR signals. The second application developed a high-speed low-jitter loopback path. The third application was the ability to adjust the amplitude of output signals. Similarly, the fourth application was the ability to adjust the timing skew of the output signals. The fifth application was the ability to inject

controlled amounts of jitter onto a test signals. The final application was low speed parametric testing that was achieved using high-speed RF switches.

The FPGA used in the test module was a critical design component. Chapter 6 discussed the selection criteria for this FPGA. Since the FPGA was used to control the test module, its design and development was presented in this chapter. The test module required a communication interface to an external control computer (PC). This chapter discussed the development of this communication interface and the test module's operation.

Multi-GHz digital systems require additional design considerations to optimize signal propagation and integrity. The test module was developed on a controlled impedance multi-layer PCB. Chapter 7 discussed the physical design considerations undertaken to achieve multi-GHz speeds. Furthermore the layout of the test module was presented in this chapter.

Chapter 8 presented the experimental results of characterizing the performance of the test module. Functionality of the core logic block of the test module was shown up to 10Gbps with 38ps (p-p) jitter. High-speed signal multiplexing was shown at 10Gbps with 32ps (p-p) jitter. Multiplexing was also shown using two 8Gbps RIO MGT signals to produce a 16Gbps output signal. The loopback path developed in this research was demonstrated to work up to 10Gbps while adding only 16ps (p-p) jitter to the input signal. Amplitude adjustment of output signals from the test module is demonstrated through a range of 100-700mV. Timing control of the output signals was demonstrated in 10ps discrete increments up to 10ns. A fine scale range of 60ps was achieved using an analog control signal. Jitter injection is demonstrated up to 80ps (p-p). Finally,

functionality of a low-speed path was also demonstrated for use with signals directly from an ATE.

9.2 Contributions

The summary given above highlights the achievements of this research. Based upon these achievements, the major contributions of this thesis are:

9.2.1 Modular test enhancement framework for ATE

A modular test enhancement framework consisting of a core logic block and an application specific block is introduced in this thesis. The modular nature of the framework allows rapid development of ATE performance enhancing test modules. Since the application specific logic block can be designed independent of the core logic block, various ATE enhancements can be developed using this framework. Furthermore, since the framework is compatible with existing ATE infrastructure, full ATE functionality can be retained.

The modular approach allows for the easy upgradability when next generation devices are available. When next generation FPGAs become available, the core logic block can be redesigned to achieve higher performance. Similarly when higher-performance components, such as XOR gates, are available, the test module can be easily redesigned to include these, thus not requiring a full redesign. Therefore the framework developed in this thesis can be applied into the foreseeable future.

9.2.2 Communication and control architecture for test modules

A command and control architecture for ATE performance enhancing electronic modules is presented in this thesis. The architecture consists of software developed for use on a PC to interface to the test module through a USB port. A microcontroller firmware architecture developed on the test module translates and executes commands sent by a user. The PC software and the microcontroller software are developed independently and designed to be compatible with each other.

Since the communication architecture presented consists of two independent components, greater flexibility is achieved. When new commands are required, the microcontroller architecture can be updated to accommodate them. Similarly, the PC software can be updated accordingly to allow new commands. Therefore this approach allows for future upgradability and furthermore allows this architecture to be used in various similar applications.

9.2.3 High-speed signal multiplexing

A method for generating high-speed signals through multiplexing lower speed signals using an ultra-precision XOR gate is presented in this thesis. A data rate doubling is achieved by offsetting the input signals by half a bit-period and combining with an XOR gate. This well-established concept has proven to be very challenging, especially in cases when the input signals are in the multi-GHz ranges. Therefore a significant contribution of this thesis is the exploration, characterization and demonstration of this technique in the 5-15GHz range using FPGAs as a data source.

The results shown in Chapter 8 demonstrated that the limitations of this application were imposed by the XOR gate. Therefore this method can be used in the future to produce higher speed signals by substituting higher-speed XOR gates. When a higher-performance XOR gate is available, it is designed within the same framework to produce even faster test signals.

9.2.4 High-speed Loopback path

This thesis presents a method for designing a high-speed loopback path using active components. The loopback path is shown to add low jitter to an input signal and demonstrated up to 10Gbps. Much of the limitations on the loopback path are imposed by the active components, i.e. the multiplexor, buffer, fan-out buffer. Using next generation devices and employing the methods shown in this thesis will allow even higher performance loopback paths.

9.2.5 Jitter Injection

A method of injecting measured jitter into a high-speed test signal is presented in this thesis. Jitter injection is achieved by placing external noise on to the programmable clock delay chip that adds it to the output clock signal and ultimately to the output test signal. This thesis establishes a relationship between characteristics of the noise applied and the jitter produced. Using this relationship, controlled amounts of jitter can be injected onto a signal. This method also allows for future upgradability when high-speed serializers become available.

9.2.6 Low-speed/parametric testing path

High-performance ATE may perform many standard tests (such as parametric measurements) better than custom built test modules. Therefore retaining full functionality of an ATE when developing test modules is desirable. A method to allow full ATE functionality while using a test module us presented in this thesis. The low-speed path allows the test module to act as a passive signal transmission path, thus allowing test signals from an ATE to be used for standard testing functions. This is achieved through the use of high-frequency RF switches, whose functionality is demonstrated up to 16Gbps. When higher performance switches are available, the switches can be replaced to accommodate higher-speed signals.

9.2.7 Physical design guidelines for high-speed test module

High-speed digital systems require special physical design considerations as discussed in Chapter 7. This thesis presents guidelines for physically designing a high-speed test-module effectively. Based on these considerations a test module design is presented that is within the size and form factor limitations imposed by a target ATE infrastructure. Furthermore the choice of PCB materials makes a significant impact on high-speed signal propagation. Therefore when higher performance PCB materials are available, a new test module capable of greater speeds can be designed using the same guidelines presented in this thesis.

9.3 Conclusions

The objective of this research was to develop methodologies to extend ATE performance capabilities into the multi-GHz range using FPGAs. Experimental results presented in this thesis demonstrate enhancing six test applications within an ATE framework. Therefore these results provide evidence that the methods presented by this thesis can be used to effectively enhance ATE performance into the multi-GHz range using economical FPGAs. Although specific example test modules are shown, the methods presented in this thesis have a broad application to future test scenarios.

The methods presented in this thesis offer seven distinct contributions discussed in the previous section. These contributions extend the applicability of this research to future test requirements. As new technological advancements are made, the same methods presented in this thesis can be used to extend ATE performance. Chapter 2 discusses that FPGA performance in terms of speed has increased much faster than ATE speeds have increased. Xilinx is introducing a Virtex-7 family of FPGA devices with the capability of 28Gbps transceivers very soon [12]. Altera has recently released a line of its Stratix V FPGAs equipped with transceivers capable of transmitting data up to 28Gbps as well [121]. The test module design presented in this thesis can utilize these new FPGAs employing the same principals to achieve 28Gbps data rates, which are far beyond the capability of available ATE.

Similarly when faster components are available, they can be incorporated in to the application specific block of the test module. Inphi Corp currently offers a 25Gbps precision XOR device [122], other manufacturers offer faster components. These XOR devices can be designed in to the application specific logic block to take two signals from

the core logic block and multiplex them to produce a double date rate using the same methods presented in this thesis. Therefore the contributions made in this thesis have the potential to be used into the foreseeable future to enhance semiconductor testing applications.

9.4 Future Work

This thesis demonstrated several functions of a test module used to enhance ATE performance. Much of this work was required in order to demonstrate proof of concept. Therefore, there is much immediate work that can be done to further improve current performance and usability.

The first improvements that can be made are to the software interface. The software interface was designed as a minimal interface to demonstrate communication to the test module. However, the software interface can readily be designed to do much more. For example, the delay non-linearities of the programmable delay chips can be programmed into the interface. This would allow the software to correct the desired delay based on calibration tables. Furthermore, potentiometers on the test module can be replaced with DACs which can be programed automatically by the core logic block. This will make delay calibration much more efficient.

Jitter injection can also be automated by the software interface. Jitter is injected using an Agilent signal source which is controlled through a computer link. The software interface can be developed to control the signal source and inject noise according to the amount of jitter desired. For example, when 20ps of jitter is required, the software

interface would be pre-programed to set the signal source to produce a 600mV noise signal to produce the required amount of jitter.

The FPGA firmware can also be developed to handle more commands, such as cycling through test patterns. Memory structures in the FPGA already exist to accommodate pre-determined test patterns. Implementing a command in the FPGA to sequentially cycle through a range of memory address is simply a matter of software development.

All the tasks discussed above will improve the performance of the generic test module design. Once specific applications have been targeted, these applications can also be built into the application specific logic if necessary and a more targeted test module developed. Furthermore, future work on the test module should also be aimed at redesigning the core logic block with the new Xilinx Virtex 7 FPGA capable of transmit speeds of 28Gbps. Since another Xilinx FPGA is to be used, much of the internal firmware can be reused, and most of the communication software will remain unchanged. The physical layout of the test module would have to be redone as the new FPGA will have to be connected to the other components. However, this entire process would entail much less than a complete redesign. Therefore, as discussed above, the modular design of the test module greatly simplifies the upgrade process.

APPENDIX A

FPGA Firmware

In this appendix details of the FPGA firmware used to control the test module are presented. The FPGA firmware is developed using Xilinx ISE software suite. A mixture of schematic entry and VHDL entry is used to produce the final firmware. The firmware is broken up in to four distinct functions. The first function is the interface to the USB port, via the microcontroller. It contains a memory block to read and write to the microcontroller, and controller logic to execute the read and write commands. It also has pin connections to the microcontroller. This function is developed using the schematic editor and shown in Figure A.1.

The second function deals with loading delay values to the four programmable delay chips. It contains pin outs to the delay chips. It also contains logic to read delay values from predefined memory addresses and sequentially load them on to the delay chips. This function is also developed using a schematic editor and shown in Figure A.2.

The third function is the state machine that controls the test module. This function is slightly more complex and developed in VHDL. Much of this code has been developed with assistance from Carl Gray at the High Speed Digital Test Lab at Georgia Tech. This code operates the state machine and executes commands based on inputs to the instruction register as discussed in Chapter 6. The code is included after Figure A.1 & Figure A.2.

The final function of FPGA firmware is to control the RIO MGTs. The RIO MGTs are complex structure with over fifty variables. Xilinx provides a RIO MGT wizard in its ISE software suite and recommends its use when designing with RIO MGTs. Using this wizard, the RIO MGT function was developed in VHDL. The RIO MGT code is included after the controller code.

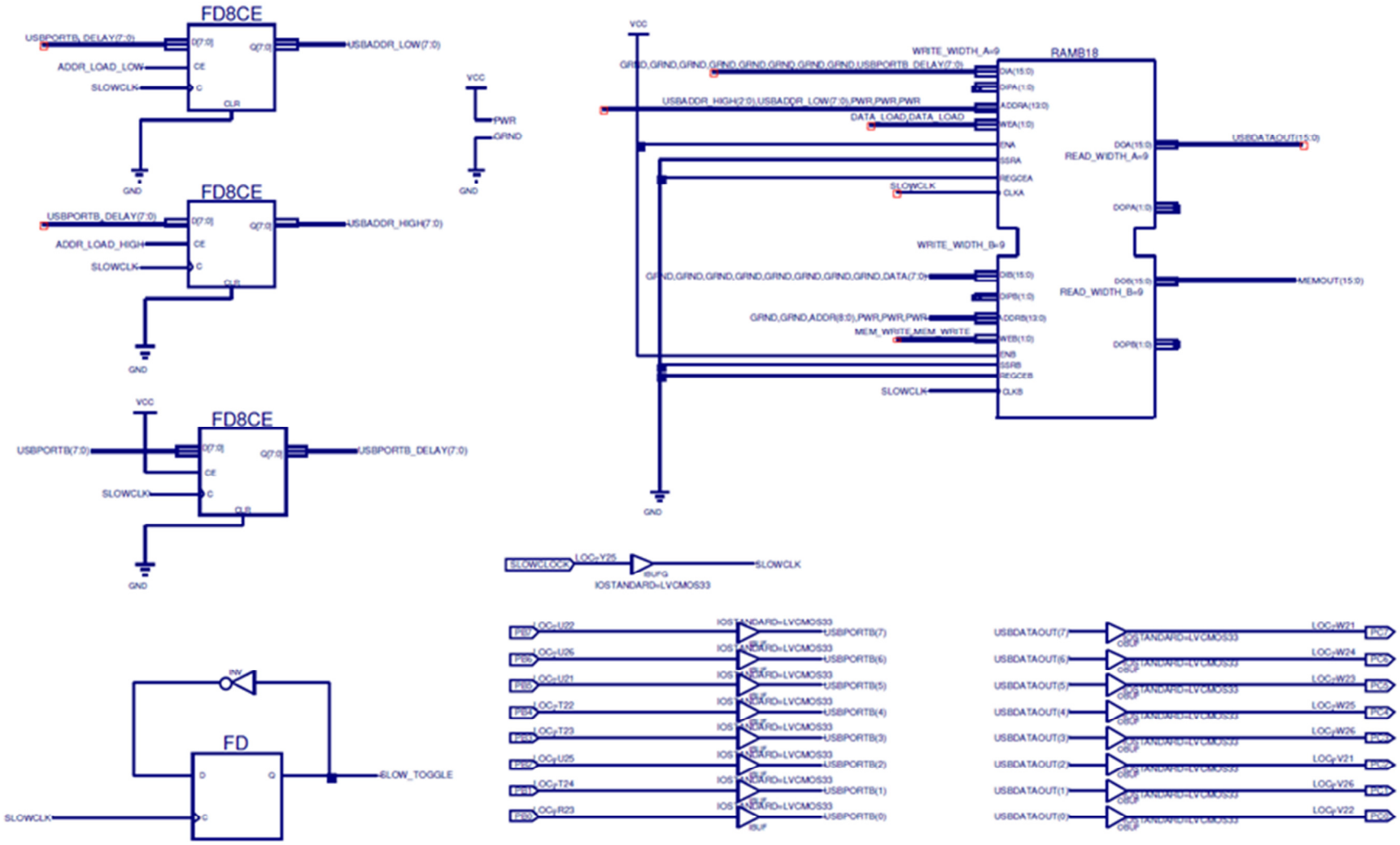


Figure A.1 USB Communication logic

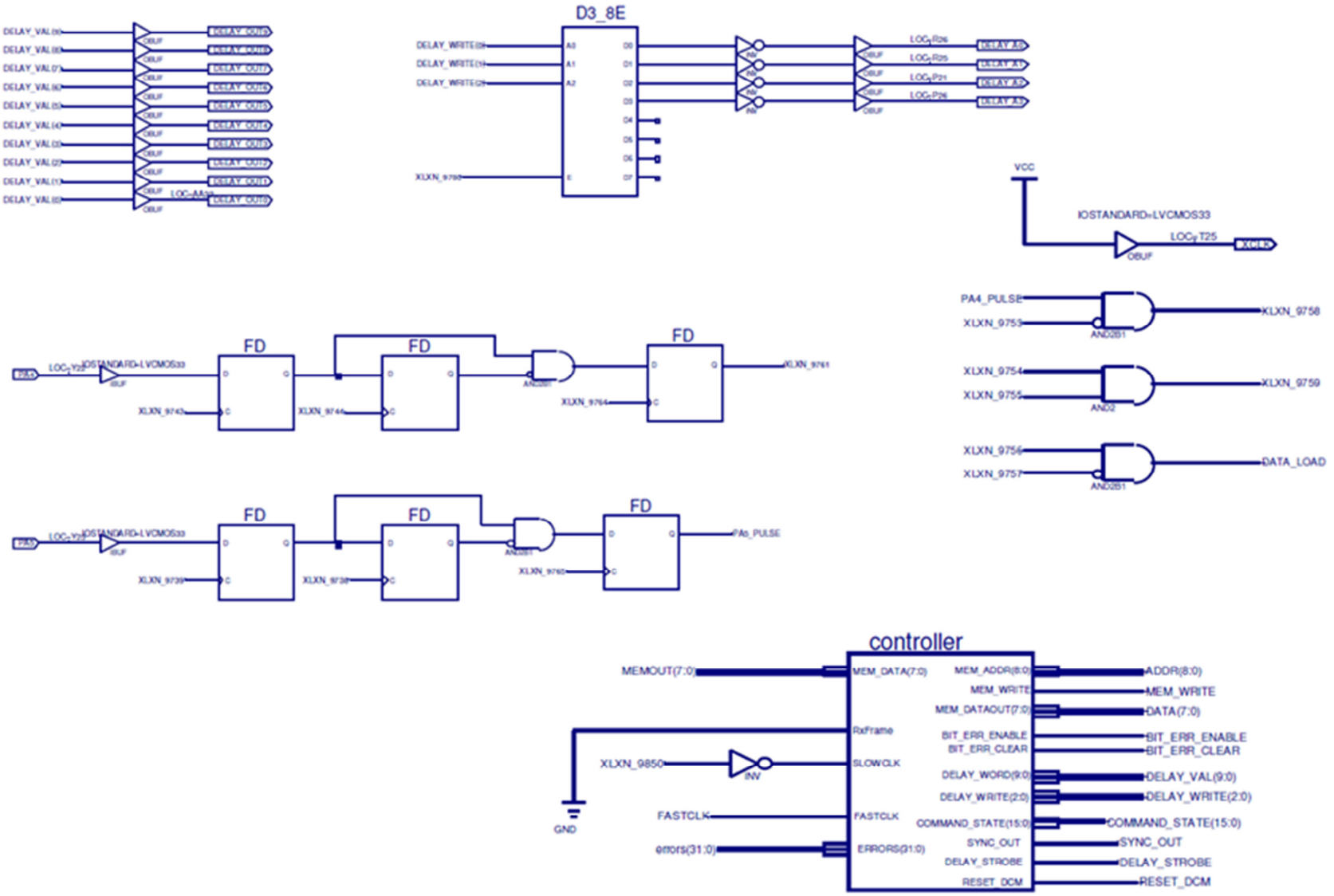


Figure A.2 Delay load logic

Controller.vhd

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

--  Uncomment the following lines to use the declarations that are
--  provided for instantiating Xilinx primitive components.
--library UNISIM;
--use UNISIM.VComponents.all;

entity controller is
    Port ( MEM_DATA : in std_logic_vector(7 downto 0);
           MEM_ADDR : out std_logic_vector(8 downto 0);
           MEM_WRITE : out std_logic;
           SLOWCLK : in std_logic;
           MEM_DATAOUT : out std_logic_vector(7 downto 0);
                                      DELAY_WORD : out std_logic_vector(9 downto 0);
                                      DELAY_WRITE : out std_logic_vector(2 downto 0);
                                      COMMAND_STATE : out std_logic_vector(15 downto 0);
                                      DELAY_STROBE : out std_logic;
                                      FASTCLK : in std_logic;
                                      SYNC_OUT : out std_logic;
                                      ERRORS : in std_logic_vector(31 downto 0);
                                      RxFrame : in std_logic;
                                      BIT_ERR_CLEAR : out std_logic;
                                      BIT_ERR_ENABLE : out std_logic;
                                      RESET_DCM : out std_logic);
end controller;

architecture Behavioral of controller is


type state_type is (RESET, STANDBY,
DELAY_LOAD,SYNCHRONIZE,SYNCHRONIZE_DCM,COMMAND_CLR,READ_ERRORS,RUN_TEST);
signal state : state_type;

signal delay_count : std_logic_vector (2 downto 0);
signal low_word : std_logic;
signal sub_state : std_logic_vector(2 downto 0);

signal sync : std_logic;
signal sync_dcm : std_logic;

signal temp_memaddr : std_logic_vector(7 downto 0);
signal delay_strobe_out : std_logic;

signal dumb_counter : std_logic_vector(7 downto 0);

signal bit_error_enable : std_logic;
signal bit_error_done : std_logic;
signal bit_error_reset : std_logic;
signal bit_error_temp : std_logic;

signal error0 : std_logic_vector(15 downto 0);
signal error1 : std_logic_vector(15 downto 0);
signal error2 : std_logic_vector(15 downto 0);
signal error3 : std_logic_vector(15 downto 0);
signal error4 : std_logic_vector(15 downto 0);
signal error5 : std_logic_vector(15 downto 0);
signal error6 : std_logic_vector(15 downto 0);
signal error7 : std_logic_vector(15 downto 0);
signal error8 : std_logic_vector(15 downto 0);
signal error9 : std_logic_vector(15 downto 0);
signal error10 : std_logic_vector(15 downto 0);
signal error11 : std_logic_vector(15 downto 0);
signal error12 : std_logic_vector(15 downto 0);
signal error13 : std_logic_vector(15 downto 0);
```

```
signal error14 : std_logic_vector(15 downto 0);
signal error15 : std_logic_vector(15 downto 0);
signal error16 : std_logic_vector(15 downto 0);
signal error17 : std_logic_vector(15 downto 0);
signal error18 : std_logic_vector(15 downto 0);
signal error19 : std_logic_vector(15 downto 0);
signal error20 : std_logic_vector(15 downto 0);
signal error21 : std_logic_vector(15 downto 0);
signal error22 : std_logic_vector(15 downto 0);
signal error23 : std_logic_vector(15 downto 0);
signal error24 : std_logic_vector(15 downto 0);
signal error25 : std_logic_vector(15 downto 0);
signal error26 : std_logic_vector(15 downto 0);
signal error27 : std_logic_vector(15 downto 0);
signal error28 : std_logic_vector(15 downto 0);
signal error29 : std_logic_vector(15 downto 0);
signal error30 : std_logic_vector(15 downto 0);
signal error31 : std_logic_vector(15 downto 0);

signal total_error : std_logic_vector(31 downto 0);

signal count : std_logic_vector(1 downto 0);

signal eight_count : std_logic_vector(2 downto 0);
signal packet_count : std_logic_vector(31 downto 0);

signal global_reset : std_logic;

signal toggle : std_logic;


begin

total_error <= error0 + error1 + error2 + error3 + error4 + error5 + error6 + error7 +
                                                error8 + error9 + error10 + error11 + error12 + error13 + error14 + error15 +
                                                error16 + error17 + error18 + error19 + error20 + error21 + error22 + error23 +
                                                error24 + error25 + error26 + error27 + error28 + error29 + error30 + error31;

MEM_ADDR <= "0" & temp_memaddr;
delay_strobe <= delay_strobe_out;

BIT_ERR_CLEAR <= bit_error_temp;
BIT_ERR_ENABLE <= toggle;

sync_out <= sync;
reset_dcm <= sync_dcm;

--main state update
process(SLOWCLK,MEM_DATA,global_reset)
begin
                        if(global_reset = '1') then
                                global_reset <= '0';
                        elsif(rising_edge(SLOWCLK)) then
                                case state is
                                        when STANDBY =>
                                                MEM_WRITE <= '0';
                                                case sub_state is
                                                        when "000" =>
                                                                temp_memaddr <= "00000010";
                                                                sub_state <= "001";
                                                        when "001" =>
                                                                COMMAND_STATE(7 downto 0) <= MEM_DATA;
                                                                temp_memaddr <= "00000011";
                                                                sub_state <= "010";
                                                        when "010"=>
                                                                COMMAND_STATE(15 downto 8) <= MEM_DATA;
                                                                sub_state <= "011";
                                                        when "011" =>
                                                                temp_memaddr <= "00000000";
                                                                sub_state <= "100";
```

```
                        when "100" =>
                                if(MEM_DATA = "00000010") then
                                        state <= DELAY_LOAD;
                                        delay_count <= "000";
                                        temp_memaddr <= "00000110";
                                        delay_strobe_out <= '0';
                                        low_word <= '0';
                                elsif(MEM_DATA = "00000011") then
                                  state       <= SYNCHRONIZE;
                                elsif(MEM_DATA = "00000100") then
                                  state       <= SYNCHRONIZE_DCM;

                                elsif(MEM_DATA = "00000101") then
                                        temp_memaddr <= "00110000";
                                        MEM_WRITE <= '1';
                                        MEM_DATAOUT <= ERRORS(7 downto
0);
                                        count <= "01";
                                        state <= READ_ERRORS;
                                elsif(MEM_DATA = "00000110") then
                                        bit_error_reset <= '1';
                                        low_word <= '0';
                                        state <= RUN_TEST;
                                end if;
                                sub_state <= "000";
                        when others =>
                                sub_state <= "000";
                        end case; -- case sub_sate
                ------------------
                when DELAY_LOAD =>
                        if(delay_count = "111") then
                                state <= COMMAND_CLR;
                                delay_strobe_out <= '0';
                                --delay_write <= "00000";
                        else
                                if(low_word = '0') then
                                        delay_word(7 downto 0) <= mem_data(7
downto 0);
                                        low_word <= '1';
                                        delay_strobe_out <= '0';
                                        temp_memaddr <= temp_memaddr + 1;
                                        delay_write <= "111";
                                elsif(delay_strobe_out = '1') then
                                        delay_strobe_out <= '0';
                                        low_word <= '0';
                                        --delay_count <= "00111";

                                        delay_count <= delay_count + 1;
                                else
                                        delay_word(9 downto 8) <= mem_data(1
downto 0);
                                        temp_memaddr <= temp_memaddr + 1;
                                        delay_strobe_out <= '1';
                                        delay_write <= delay_count;

                                end if;
                        end if;
                ------------------
                when SYNCHRONIZE =>
                                bit_error_enable <= '0';
                                bit_error_reset <= '1';
                                global_reset <= '1';
                                sync <= '1';
                                state <= COMMAND_CLR;
                ------------------
                when SYNCHRONIZE_DCM =>
                                sync_dcm <= '1';
                                state <= COMMAND_CLR;
                ------------------
                when READ_ERRORS =>
```

```
                                                if(temp_memaddr = "00110100") then
                                                        state <= COMMAND_CLR;
                                                else
                                                        case count is
                                                                WHEN "01" =>
                                                                        MEM_DATAOUT <=
ERRORS(15 downto 8);
                                                                WHEN "10" =>
                                                                        MEM_DATAOUT <=
ERRORS(23 downto 16);
                                                                WHEN "11" =>
                                                                        MEM_DATAOUT <=
ERRORS(31 downto 24);
                                                                WHEN others =>
                                                        end case;
                                                        count <= count + 1;
                                                        temp_memaddr <= temp_memaddr + 1;
                                                        MEM_WRITE <= '1';
                                                end if;
                                        --------------------
                                        when RUN_TEST =>
                                                bit_error_enable <= '1';
                                                MEM_WRITE <= '1';
                                                if(low_word = '0') then
                                                        temp_memaddr <= "00111100";
                                                        MEM_DATAOUT <= packet_count(31 downto 24);
                                                        low_word <= '1';
                                                else
                                                        temp_memaddr <= "00111101";
                                                        MEM_DATAOUT <= total_error(15 downto 8);
                                                        low_word <= '0';
                                                end if;

                                                if(bit_error_done = '1') then
                                                        temp_memaddr <= "00111100";
                                                        MEM_DATAOUT <= total_error(7 downto 0);

                                                        state <= COMMAND_CLR;
                                                end if;
                                        --------------------
                                        when COMMAND_CLR =>
                                                        MEM_WRITE <= '1';
                                                        sync <= '0';
                                                        sync_dcm <= '0';
                                                        MEM_DATAOUT <= "00000000";
                                                        temp_memaddr <= "00000000";
                                                        state <= STANDBY;
                                                        sub_state <= "000";
                                        when others =>
                                                state <= STANDBY;
                                                sub_state <= "000";
                                        end case;
                        end if;

end process;

process(RxFrame, bit_error_enable, global_reset)
begin
        if(global_reset = '1') then
                        bit_error_temp <= '0';
                        bit_error_done <= '0';
                        packet_count <= "00000000000000000000000000000000";

                        error0 <= "0000000000000000";
                        error1 <= "0000000000000000";
                        error2 <= "0000000000000000";
                        error3 <= "0000000000000000";
                        error4 <= "0000000000000000";
                        error5 <= "0000000000000000";
                        error6 <= "0000000000000000";
```

```
                        error7 <= "0000000000000000";
                        error8 <= "0000000000000000";
                        error9 <= "0000000000000000";
                        error10 <= "0000000000000000";
                        error11 <= "0000000000000000";
                        error12 <= "0000000000000000";
                        error13 <= "0000000000000000";
                        error14 <= "0000000000000000";
                        error15 <= "0000000000000000";
                        error16 <= "0000000000000000";
                        error17 <= "0000000000000000";
                        error18 <= "0000000000000000";
                        error19 <= "0000000000000000";
                        error20 <= "0000000000000000";
                        error21 <= "0000000000000000";
                        error22 <= "0000000000000000";
                        error23 <= "0000000000000000";
                        error24 <= "0000000000000000";
                        error25 <= "0000000000000000";
                        error26 <= "0000000000000000";
                        error27 <= "0000000000000000";
                        error28 <= "0000000000000000";
                        error29 <= "0000000000000000";
                        error30 <= "0000000000000000";
                        error31 <= "0000000000000000";

                elsif(rising_edge(RxFrame)) then

                        if(packet_count = "11111111111111111111111111111111") then
                                bit_error_done <= '1';
                        end if;
                         ------------------------------------baaaaaad code here

if (errors(0) = '1') then
                error0 <= error0  + 1;
end if;
if (errors(1) = '1') then
                error1 <= error1  + 1;
end if;
if (errors(2) = '1') then
                error2 <= error2  + 1;
end if;
if (errors(3) = '1') then
                error3 <= error3  + 1;
end if;
if (errors(4) = '1') then
                error4 <= error4  + 1;
end if;
if (errors(5) = '1') then
                error5 <= error5  + 1;
end if;
if (errors(6) = '1') then
                error6 <= error6  + 1;
end if;
if (errors(7) = '1') then
                error7 <= error7  + 1;
end if;
if (errors(8) = '1') then
                error8 <= error8  + 1;
end if;
if (errors(9) = '1') then
                error9 <= error9  + 1;
end if;
if (errors(10) = '1') then
                error10 <= error10  + 1;
end if;
if (errors(11) = '1') then
                error11 <= error11  + 1;
end if;
if (errors(12) = '1') then
```

```
                error12 <= error12  + 1;
end if;
if (errors(13) = '1') then
                error13 <= error13  + 1;
end if;
if (errors(14) = '1') then
                error14 <= error14  + 1;
end if;
if (errors(15) = '1') then
                error15 <= error15  + 1;
end if;
if (errors(16) = '1') then
                error16 <= error16  + 1;
end if;
if (errors(17) = '1') then
                error17 <= error17  + 1;
end if;
if (errors(18) = '1') then
                error18 <= error18  + 1;
end if;
if (errors(19) = '1') then
                error19 <= error19  + 1;
end if;
if (errors(20) = '1') then
                error20 <= error20  + 1;
end if;
if (errors(21) = '1') then
                error21 <= error21  + 1;
end if;
if (errors(22) = '1') then
                error22 <= error22  + 1;
end if;
if (errors(23) = '1') then
                error23 <= error23  + 1;
end if;
if (errors(24) = '1') then
                error24 <= error24  + 1;
end if;
if (errors(25) = '1') then
                error25 <= error25  + 1;
end if;
if (errors(26) = '1') then
                error26 <= error26  + 1;
end if;
if (errors(27) = '1') then
                error27 <= error27  + 1;
end if;
if (errors(28) = '1') then
                error28 <= error28  + 1;
end if;
if (errors(29) = '1') then
                error29 <= error29  + 1;
end if;
if (errors(30) = '1') then
                error30 <= error30  + 1;
end if;
if (errors(31) = '1') then
                error31 <= error31  + 1;
end if;


-------------------------------------------------

                        packet_count <= packet_count + 1;

                        if(bit_error_temp = '0') then
                                bit_error_temp <= '1';
                        else
                                bit_error_temp <= '0';
                        end if;
```

```
            end if;
    end process;
    end Behavioral;
```

RocketIO.vhd

```
--------------------------------------------------------------------------------
--$Date: 2008/07/23 00:16:39 $
--$Revision: 1.1.2.7 $
--------------------------------------------------------------------------------
--   ____  ____
-- /   /\/   /
-- /___/  \  /    Vendor: Xilinx
-- \   \   \/     Version : 1.5
-- \   \         Application : RocketIO GTX Wizard
-- /   /         Filename : example_mgt_top.vhd
-- /___/   /\     Timestamp :
-- \   \  / \
-- \___\/\___\
--
--
-- Module EXAMPLE_MGT_TOP
-- Generated by Xilinx RocketIO GTX Wizard

library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
library UNISIM;
use UNISIM.VCOMPONENTS.ALL;



--***********************************Entity Declaration************************

entity EXAMPLE_MGT_TOP is
generic
(
   EXAMPLE_CONFIG_INDEPENDENT_LANES       : integer   := 1;
   EXAMPLE_LANE_WITH_START_CHAR           : integer   := 0;
   EXAMPLE_WORDS_IN_BRAM                  : integer   := 512;
   EXAMPLE_SIM_MODE                       : string    := "FAST";
   EXAMPLE_SIM_GTXRESET_SPEEDUP           : integer   := 1;
   EXAMPLE_SIM_PLL_PERDIV2                : bit_vector:= x"0a0";
   EXAMPLE_USE_CHIPSCOPE                  : integer   := 0     -- Set to 1 to use Chipscope to drive resets
);
port
(
   TILE0_REFCLK_PAD_N_IN                  : in   std_logic;
   TILE0_REFCLK_PAD_P_IN                  : in   std_logic;
   TILE1_REFCLK_PAD_N_IN                  : in   std_logic;
   TILE1_REFCLK_PAD_P_IN                  : in   std_logic;
   DRP_CLK_IN                         : in   std_logic;
   GTXRESET_IN                        : in   std_logic;
   TILE0_PLLLKDET_OUT                     : out  std_logic;
   TILE1_PLLLKDET_OUT                     : out  std_logic;
   RXN_IN                           : in   std_logic_vector(3 downto 0);
   RXP_IN                           : in   std_logic_vector(3 downto 0);
   TXN_OUT                            : out  std_logic_vector(3 downto 0);
   TXP_OUT                            : out  std_logic_vector(3 downto 0)

);

   attribute X_CORE_INFO : string;
   attribute X_CORE_INFO of EXAMPLE_MGT_TOP : entity is "gtxwizard_v1_5, Coregen v10.1_ip3";

end EXAMPLE_MGT_TOP;

architecture RTL of EXAMPLE_MGT_TOP is

--***************************Component Declarations*****************************


component ROCKETIO_WRAPPER
generic
```

```
(
   -- Simulation attributes
   WRAPPER_SIM_MODE              : string    := "FAST"; -- Set to Fast Functional Simulation Model
   WRAPPER_SIM_GTXRESET_SPEEDUP  : integer   := 0; -- Set to 1 to speed up sim reset
   WRAPPER_SIM_PLL_PERDIV2       : bit_vector:= x"0a0" -- Set to the VCO Unit Interval time
);
port
(
   ------------------------ Loopback and Powerdown Ports ----------------------
   TILE0_LOOPBACK0_IN                  : in   std_logic_vector(2 downto 0);
   TILE0_LOOPBACK1_IN                  : in   std_logic_vector(2 downto 0);
   ----------------------- Receive Ports - 8b10b Decoder ----------------------
   TILE0_RXDISPERR0_OUT                : out  std_logic_vector(1 downto 0);
   TILE0_RXDISPERR1_OUT                : out  std_logic_vector(1 downto 0);
   TILE0_RXNOTINTABLE0_OUT             : out  std_logic_vector(1 downto 0);
   TILE0_RXNOTINTABLE1_OUT             : out  std_logic_vector(1 downto 0);
   --------------- Receive Ports - Comma Detection and Alignment --------------
   TILE0_RXENMCOMMAALIGN0_IN           : in   std_logic;
   TILE0_RXENMCOMMAALIGN1_IN           : in   std_logic;
   TILE0_RXENPCOMMAALIGN0_IN           : in   std_logic;
   TILE0_RXENPCOMMAALIGN1_IN           : in   std_logic;
   ------------------- Receive Ports - RX Data Path interface -----------------
   TILE0_RXDATA0_OUT                   : out  std_logic_vector(15 downto 0);
   TILE0_RXDATA1_OUT                   : out  std_logic_vector(15 downto 0);
   TILE0_RXRESET0_IN                   : in   std_logic;
   TILE0_RXRESET1_IN                   : in   std_logic;
   TILE0_RXUSRCLK0_IN                  : in   std_logic;
   TILE0_RXUSRCLK1_IN                  : in   std_logic;
   TILE0_RXUSRCLK20_IN                 : in   std_logic;
   TILE0_RXUSRCLK21_IN                 : in   std_logic;
   ------- Receive Ports - RX Driver,OOB signalling,Coupling and Eq.,CDR ------
   TILE0_RXEQMIX0_IN                   : in   std_logic_vector(1 downto 0);
   TILE0_RXEQMIX1_IN                   : in   std_logic_vector(1 downto 0);
   TILE0_RXN0_IN                       : in   std_logic;
   TILE0_RXN1_IN                       : in   std_logic;
   TILE0_RXP0_IN                       : in   std_logic;
   TILE0_RXP1_IN                       : in   std_logic;
   --------------- Receive Ports - RX Loss-of-sync State Machine --------------
   TILE0_RXLOSSOFSYNC0_OUT             : out  std_logic_vector(1 downto 0);
   TILE0_RXLOSSOFSYNC1_OUT             : out  std_logic_vector(1 downto 0);
   ------------ Shared Ports - Dynamic Reconfiguration Port (DRP) ------------
   TILE0_DADDR_IN                      : in   std_logic_vector(6 downto 0);
   --TILE0_DCLK_IN                     : in   std_logic;
   TILE0_DEN_IN                        : in   std_logic;
   TILE0_DI_IN                         : in   std_logic_vector(15 downto 0);
   TILE0_DO_OUT                        : out  std_logic_vector(15 downto 0);
   TILE0_DRDY_OUT                      : out  std_logic;
   TILE0_DWE_IN                        : in   std_logic;
   -------------------- Shared Ports - Tile and PLL Ports --------------------
   TILE0_CLKIN_IN                      : in   std_logic;
   TILE0_GTXRESET_IN                   : in   std_logic;
   TILE0_PLLLKDET_OUT                  : out  std_logic;
   TILE0_REFCLKOUT_OUT                 : out  std_logic;
   TILE0_RESETDONE0_OUT                : out  std_logic;
   TILE0_RESETDONE1_OUT                : out  std_logic;
   ---------------- Transmit Ports - 8b10b Encoder Control Ports --------------
   TILE0_TXCHARISK0_IN                 : in   std_logic_vector(1 downto 0);
   TILE0_TXCHARISK1_IN                 : in   std_logic_vector(1 downto 0);
   ------------------ Transmit Ports - TX Data Path interface -----------------
   TILE0_TXDATA0_IN                    : in   std_logic_vector(15 downto 0);
   TILE0_TXDATA1_IN                    : in   std_logic_vector(15 downto 0);
   TILE0_TXRESET0_IN                   : in   std_logic;
   TILE0_TXRESET1_IN                   : in   std_logic;
   TILE0_TXUSRCLK0_IN                  : in   std_logic;
   TILE0_TXUSRCLK1_IN                  : in   std_logic;
   TILE0_TXUSRCLK20_IN                 : in   std_logic;
   TILE0_TXUSRCLK21_IN                 : in   std_logic;
   ---------------- Transmit Ports - TX Driver and OOB signalling -------------
   TILE0_TXDIFFCTRL0_IN                : in   std_logic_vector(2 downto 0);
   TILE0_TXDIFFCTRL1_IN                : in   std_logic_vector(2 downto 0);
```

```
TILE0_TXN0_OUT                  : out  std_logic;
TILE0_TXN1_OUT                  : out  std_logic;
TILE0_TXP0_OUT                  : out  std_logic;
TILE0_TXP1_OUT                  : out  std_logic;
TILE0_TXPREEMPHASIS0_IN         : in   std_logic_vector(2 downto 0);
TILE0_TXPREEMPHASIS1_IN         : in   std_logic_vector(2 downto 0);
---------------------- Transmit Ports - TX PRBS Generator -------------------
TILE0_TXENPRBSTST0_IN           : in   std_logic_vector(1 downto 0);
TILE0_TXENPRBSTST1_IN           : in   std_logic_vector(1 downto 0);
----------------------- Loopback and Powerdown Ports ----------------------
TILE1_LOOPBACK0_IN              : in   std_logic_vector(2 downto 0);
TILE1_LOOPBACK1_IN              : in   std_logic_vector(2 downto 0);
----------------------- Receive Ports - 8b10b Decoder ----------------------
TILE1_RXDISPERR0_OUT            : out  std_logic_vector(1 downto 0);
TILE1_RXDISPERR1_OUT            : out  std_logic_vector(1 downto 0);
TILE1_RXNOTINTABLE0_OUT         : out  std_logic_vector(1 downto 0);
TILE1_RXNOTINTABLE1_OUT         : out  std_logic_vector(1 downto 0);
--------------- Receive Ports - Comma Detection and Alignment --------------
TILE1_RXENMCOMMAALIGN0_IN       : in   std_logic;
TILE1_RXENMCOMMAALIGN1_IN       : in   std_logic;
TILE1_RXENPCOMMAALIGN0_IN       : in   std_logic;
TILE1_RXENPCOMMAALIGN1_IN       : in   std_logic;
------------------ Receive Ports - RX Data Path interface -----------------
TILE1_RXDATA0_OUT               : out  std_logic_vector(15 downto 0);
TILE1_RXDATA1_OUT               : out  std_logic_vector(15 downto 0);
TILE1_RXRESET0_IN               : in   std_logic;
TILE1_RXRESET1_IN               : in   std_logic;
TILE1_RXUSRCLK0_IN              : in   std_logic;
TILE1_RXUSRCLK1_IN              : in   std_logic;
TILE1_RXUSRCLK20_IN             : in   std_logic;
TILE1_RXUSRCLK21_IN             : in   std_logic;
------- Receive Ports - RX Driver,OOB signalling,Coupling and Eq.,CDR ------
TILE1_RXEQMIX0_IN               : in   std_logic_vector(1 downto 0);
TILE1_RXEQMIX1_IN               : in   std_logic_vector(1 downto 0);
TILE1_RXN0_IN                   : in   std_logic;
TILE1_RXN1_IN                   : in   std_logic;
TILE1_RXP0_IN                   : in   std_logic;
TILE1_RXP1_IN                   : in   std_logic;
--------------- Receive Ports - RX Loss-of-sync State Machine --------------
TILE1_RXLOSSOFSYNC0_OUT         : out  std_logic_vector(1 downto 0);
TILE1_RXLOSSOFSYNC1_OUT         : out  std_logic_vector(1 downto 0);
------------- Shared Ports - Dynamic Reconfiguration Port (DRP) ------------
TILE1_DADDR_IN                  : in   std_logic_vector(6 downto 0);
--TILE1_DCLK_IN                 : in   std_logic;
TILE1_DEN_IN                    : in   std_logic;
TILE1_DI_IN                     : in   std_logic_vector(15 downto 0);
TILE1_DO_OUT                    : out  std_logic_vector(15 downto 0);
TILE1_DRDY_OUT                  : out  std_logic;
TILE1_DWE_IN                    : in   std_logic;
---------------------- Shared Ports - Tile and PLL Ports ---------------------
TILE1_CLKIN_IN                  : in   std_logic;
TILE1_GTXRESET_IN               : in   std_logic;
TILE1_PLLLKDET_OUT              : out  std_logic;
TILE1_REFCLKOUT_OUT             : out  std_logic;
TILE1_RESETDONE0_OUT            : out  std_logic;
TILE1_RESETDONE1_OUT            : out  std_logic;
---------------- Transmit Ports - 8b10b Encoder Control Ports --------------
TILE1_TXCHARISK0_IN             : in   std_logic_vector(1 downto 0);
TILE1_TXCHARISK1_IN             : in   std_logic_vector(1 downto 0);
------------------ Transmit Ports - TX Data Path interface -----------------
TILE1_TXDATA0_IN                : in   std_logic_vector(15 downto 0);
TILE1_TXDATA1_IN                : in   std_logic_vector(15 downto 0);
TILE1_TXRESET0_IN               : in   std_logic;
TILE1_TXRESET1_IN               : in   std_logic;
TILE1_TXUSRCLK0_IN              : in   std_logic;
TILE1_TXUSRCLK1_IN              : in   std_logic;
TILE1_TXUSRCLK20_IN             : in   std_logic;
TILE1_TXUSRCLK21_IN             : in   std_logic;
--------------- Transmit Ports - TX Driver and OOB signalling --------------
TILE1_TXDIFFCTRL0_IN            : in   std_logic_vector(2 downto 0);
```

```
    TILE1_TXDIFFCTRL1_IN                : in   std_logic_vector(2 downto 0);
    TILE1_TXN0_OUT                      : out  std_logic;
    TILE1_TXN1_OUT                      : out  std_logic;
    TILE1_TXP0_OUT                      : out  std_logic;
    TILE1_TXP1_OUT                      : out  std_logic;
    TILE1_TXPREEMPHASIS0_IN             : in   std_logic_vector(2 downto 0);
    TILE1_TXPREEMPHASIS1_IN             : in   std_logic_vector(2 downto 0);
    ---------------------- Transmit Ports - TX PRBS Generator -------------------
    TILE1_TXENPRBSTST0_IN               : in   std_logic_vector(1 downto 0);
    TILE1_TXENPRBSTST1_IN               : in   std_logic_vector(1 downto 0)
);
end component;

component MGT_USRCLK_SOURCE
generic
(
    FREQUENCY_MODE   : string   := "LOW";
    PERFORMANCE_MODE : string   := "MAX_SPEED"
);
port
(
    DIV1_OUT              : out std_logic;
    DIV2_OUT              : out std_logic;
    DCM_LOCKED_OUT        : out std_logic;
    CLK_IN                : in  std_logic;
    DCM_RESET_IN          : in  std_logic

);
end component;

component FRAME_GEN
generic
(
    WORDS_IN_BRAM : integer    :=   256;
    MEM_00      : bit_vector :=   X"0000000000000000000000000000000000000000000000000000000000000000";
    MEM_01      : bit_vector :=   X"0000000000000000000000000000000000000000000000000000000000000000";
    MEM_02      : bit_vector :=   X"0000000000000000000000000000000000000000000000000000000000000000";
    MEM_03      : bit_vector :=   X"0000000000000000000000000000000000000000000000000000000000000000";
    MEM_04      : bit_vector :=   X"0000000000000000000000000000000000000000000000000000000000000000";
    MEM_05      : bit_vector :=   X"0000000000000000000000000000000000000000000000000000000000000000";
    MEM_06      : bit_vector :=   X"0000000000000000000000000000000000000000000000000000000000000000";
    MEM_07      : bit_vector :=   X"0000000000000000000000000000000000000000000000000000000000000000";
    MEM_08      : bit_vector :=   X"0000000000000000000000000000000000000000000000000000000000000000";
    MEM_09      : bit_vector :=   X"0000000000000000000000000000000000000000000000000000000000000000";
    MEM_0A      : bit_vector :=   X"0000000000000000000000000000000000000000000000000000000000000000";
    MEM_0B      : bit_vector :=   X"0000000000000000000000000000000000000000000000000000000000000000";
    MEM_0C      : bit_vector :=   X"0000000000000000000000000000000000000000000000000000000000000000";
    MEM_0D      : bit_vector :=   X"0000000000000000000000000000000000000000000000000000000000000000";
    MEM_0E      : bit_vector :=   X"0000000000000000000000000000000000000000000000000000000000000000";
    MEM_0F      : bit_vector :=   X"0000000000000000000000000000000000000000000000000000000000000000";
    MEM_10      : bit_vector :=   X"0000000000000000000000000000000000000000000000000000000000000000";
    MEM_11      : bit_vector :=   X"0000000000000000000000000000000000000000000000000000000000000000";
    MEM_12      : bit_vector :=   X"0000000000000000000000000000000000000000000000000000000000000000";
    MEM_13      : bit_vector :=   X"0000000000000000000000000000000000000000000000000000000000000000";
    MEM_14      : bit_vector :=   X"0000000000000000000000000000000000000000000000000000000000000000";
    MEM_15      : bit_vector :=   X"0000000000000000000000000000000000000000000000000000000000000000";
    MEM_16      : bit_vector :=   X"0000000000000000000000000000000000000000000000000000000000000000";
    MEM_17      : bit_vector :=   X"0000000000000000000000000000000000000000000000000000000000000000";
    MEM_18      : bit_vector :=   X"0000000000000000000000000000000000000000000000000000000000000000";
    MEM_19      : bit_vector :=   X"0000000000000000000000000000000000000000000000000000000000000000";
    MEM_1A      : bit_vector :=   X"0000000000000000000000000000000000000000000000000000000000000000";
    MEM_1B      : bit_vector :=   X"0000000000000000000000000000000000000000000000000000000000000000";
    MEM_1C      : bit_vector :=   X"0000000000000000000000000000000000000000000000000000000000000000";
    MEM_1D      : bit_vector :=   X"0000000000000000000000000000000000000000000000000000000000000000";
    MEM_1E      : bit_vector :=   X"0000000000000000000000000000000000000000000000000000000000000000";
    MEM_1F      : bit_vector :=   X"0000000000000000000000000000000000000000000000000000000000000000";
    MEM_20      : bit_vector :=   X"0000000000000000000000000000000000000000000000000000000000000000";
    MEM_21      : bit_vector :=   X"0000000000000000000000000000000000000000000000000000000000000000";
    MEM_22      : bit_vector :=   X"0000000000000000000000000000000000000000000000000000000000000000";
    MEM_23      : bit_vector :=   X"0000000000000000000000000000000000000000000000000000000000000000";
```

```
    MEM_24      : bit_vector :=  X"0000000000000000000000000000000000000000000000000000000000000000";
    MEM_25      : bit_vector :=  X"0000000000000000000000000000000000000000000000000000000000000000";
    MEM_26      : bit_vector :=  X"0000000000000000000000000000000000000000000000000000000000000000";
    MEM_27      : bit_vector :=  X"0000000000000000000000000000000000000000000000000000000000000000";
    MEM_28      : bit_vector :=  X"0000000000000000000000000000000000000000000000000000000000000000";
    MEM_29      : bit_vector :=  X"0000000000000000000000000000000000000000000000000000000000000000";
    MEM_2A      : bit_vector :=  X"0000000000000000000000000000000000000000000000000000000000000000";
    MEM_2B      : bit_vector :=  X"0000000000000000000000000000000000000000000000000000000000000000";
    MEM_2C      : bit_vector :=  X"0000000000000000000000000000000000000000000000000000000000000000";
    MEM_2D      : bit_vector :=  X"0000000000000000000000000000000000000000000000000000000000000000";
    MEM_2E      : bit_vector :=  X"0000000000000000000000000000000000000000000000000000000000000000";
    MEM_2F      : bit_vector :=  X"0000000000000000000000000000000000000000000000000000000000000000";
    MEM_30      : bit_vector :=  X"0000000000000000000000000000000000000000000000000000000000000000";
    MEM_31      : bit_vector :=  X"0000000000000000000000000000000000000000000000000000000000000000";
    MEM_32      : bit_vector :=  X"0000000000000000000000000000000000000000000000000000000000000000";
    MEM_33      : bit_vector :=  X"0000000000000000000000000000000000000000000000000000000000000000";
    MEM_34      : bit_vector :=  X"0000000000000000000000000000000000000000000000000000000000000000";
    MEM_35      : bit_vector :=  X"0000000000000000000000000000000000000000000000000000000000000000";
    MEM_36      : bit_vector :=  X"0000000000000000000000000000000000000000000000000000000000000000";
    MEM_37      : bit_vector :=  X"0000000000000000000000000000000000000000000000000000000000000000";
    MEM_38      : bit_vector :=  X"0000000000000000000000000000000000000000000000000000000000000000";
    MEM_39      : bit_vector :=  X"0000000000000000000000000000000000000000000000000000000000000000";
    MEM_3A      : bit_vector :=  X"0000000000000000000000000000000000000000000000000000000000000000";
    MEM_3B      : bit_vector :=  X"0000000000000000000000000000000000000000000000000000000000000000";
    MEM_3C      : bit_vector :=  X"0000000000000000000000000000000000000000000000000000000000000000";
    MEM_3D      : bit_vector :=  X"0000000000000000000000000000000000000000000000000000000000000000";
    MEM_3E      : bit_vector :=  X"0000000000000000000000000000000000000000000000000000000000000000";
    MEM_3F      : bit_vector :=  X"0000000000000000000000000000000000000000000000000000000000000000";
    MEMP_00     : bit_vector :=  X"0000000000000000000000000000000000000000000000000000000000000000";
    MEMP_01     : bit_vector :=  X"0000000000000000000000000000000000000000000000000000000000000000";
    MEMP_02     : bit_vector :=  X"0000000000000000000000000000000000000000000000000000000000000000";
    MEMP_03     : bit_vector :=  X"0000000000000000000000000000000000000000000000000000000000000000";
    MEMP_04     : bit_vector :=  X"0000000000000000000000000000000000000000000000000000000000000000";
    MEMP_05     : bit_vector :=  X"0000000000000000000000000000000000000000000000000000000000000000";
    MEMP_06     : bit_vector :=  X"0000000000000000000000000000000000000000000000000000000000000000";
    MEMP_07     : bit_vector :=  X"0000000000000000000000000000000000000000000000000000000000000000"
);
port
(
    -- User Interface
    TX_DATA           : out   std_logic_vector(39 downto 0);
    TX_CHARISK        : out   std_logic_vector(3 downto 0);

    -- System Interface
    USER_CLK          : in    std_logic;
    SYSTEM_RESET      : in    std_logic
);
end component;

component FRAME_CHECK
generic
(
    RX_DATA_WIDTH           : integer := 16;
    USE_COMMA               : integer := 1;
    NONE_MSB_FIRST_DEC      : integer := 0;
    COMMA_DOUBLE_DEC        : integer := 0;
    CHANBOND_SEQ_LEN        : integer := 1;
    WORDS_IN_BRAM           : integer := 256;
    CONFIG_INDEPENDENT_LANES : integer := 0;
    START_OF_PACKET_CHAR    : std_logic_vector := x"55fb";
    COMMA_DOUBLE_CHAR       : std_logic_vector := x"f628";
    MEM_00      : bit_vector :=  X"0000000000000000000000000000000000000000000000000000000000000000";
    MEM_01      : bit_vector :=  X"0000000000000000000000000000000000000000000000000000000000000000";
    MEM_02      : bit_vector :=  X"0000000000000000000000000000000000000000000000000000000000000000";
    MEM_03      : bit_vector :=  X"0000000000000000000000000000000000000000000000000000000000000000";
    MEM_04      : bit_vector :=  X"0000000000000000000000000000000000000000000000000000000000000000";
    MEM_05      : bit_vector :=  X"0000000000000000000000000000000000000000000000000000000000000000";
    MEM_06      : bit_vector :=  X"0000000000000000000000000000000000000000000000000000000000000000";
    MEM_07      : bit_vector :=  X"0000000000000000000000000000000000000000000000000000000000000000";
    MEM_08      : bit_vector :=  X"0000000000000000000000000000000000000000000000000000000000000000";
```

```
    MEM_09      : bit_vector :=  X"0000000000000000000000000000000000000000000000000000000000000000";
    MEM_0A      : bit_vector :=  X"0000000000000000000000000000000000000000000000000000000000000000";
    MEM_0B      : bit_vector :=  X"0000000000000000000000000000000000000000000000000000000000000000";
    MEM_0C      : bit_vector :=  X"0000000000000000000000000000000000000000000000000000000000000000";
    MEM_0D      : bit_vector :=  X"0000000000000000000000000000000000000000000000000000000000000000";
    MEM_0E      : bit_vector :=  X"0000000000000000000000000000000000000000000000000000000000000000";
    MEM_0F      : bit_vector :=  X"0000000000000000000000000000000000000000000000000000000000000000";
    MEM_10      : bit_vector :=  X"0000000000000000000000000000000000000000000000000000000000000000";
    MEM_11      : bit_vector :=  X"0000000000000000000000000000000000000000000000000000000000000000";
    MEM_12      : bit_vector :=  X"0000000000000000000000000000000000000000000000000000000000000000";
    MEM_13      : bit_vector :=  X"0000000000000000000000000000000000000000000000000000000000000000";
    MEM_14      : bit_vector :=  X"0000000000000000000000000000000000000000000000000000000000000000";
    MEM_15      : bit_vector :=  X"0000000000000000000000000000000000000000000000000000000000000000";
    MEM_16      : bit_vector :=  X"0000000000000000000000000000000000000000000000000000000000000000";
    MEM_17      : bit_vector :=  X"0000000000000000000000000000000000000000000000000000000000000000";
    MEM_18      : bit_vector :=  X"0000000000000000000000000000000000000000000000000000000000000000";
    MEM_19      : bit_vector :=  X"0000000000000000000000000000000000000000000000000000000000000000";
    MEM_1A      : bit_vector :=  X"0000000000000000000000000000000000000000000000000000000000000000";
    MEM_1B      : bit_vector :=  X"0000000000000000000000000000000000000000000000000000000000000000";
    MEM_1C      : bit_vector :=  X"0000000000000000000000000000000000000000000000000000000000000000";
    MEM_1D      : bit_vector :=  X"0000000000000000000000000000000000000000000000000000000000000000";
    MEM_1E      : bit_vector :=  X"0000000000000000000000000000000000000000000000000000000000000000";
    MEM_1F      : bit_vector :=  X"0000000000000000000000000000000000000000000000000000000000000000";
    MEM_20      : bit_vector :=  X"0000000000000000000000000000000000000000000000000000000000000000";
    MEM_21      : bit_vector :=  X"0000000000000000000000000000000000000000000000000000000000000000";
    MEM_22      : bit_vector :=  X"0000000000000000000000000000000000000000000000000000000000000000";
    MEM_23      : bit_vector :=  X"0000000000000000000000000000000000000000000000000000000000000000";
    MEM_24      : bit_vector :=  X"0000000000000000000000000000000000000000000000000000000000000000";
    MEM_25      : bit_vector :=  X"0000000000000000000000000000000000000000000000000000000000000000";
    MEM_26      : bit_vector :=  X"0000000000000000000000000000000000000000000000000000000000000000";
    MEM_27      : bit_vector :=  X"0000000000000000000000000000000000000000000000000000000000000000";
    MEM_28      : bit_vector :=  X"0000000000000000000000000000000000000000000000000000000000000000";
    MEM_29      : bit_vector :=  X"0000000000000000000000000000000000000000000000000000000000000000";
    MEM_2A      : bit_vector :=  X"0000000000000000000000000000000000000000000000000000000000000000";
    MEM_2B      : bit_vector :=  X"0000000000000000000000000000000000000000000000000000000000000000";
    MEM_2C      : bit_vector :=  X"0000000000000000000000000000000000000000000000000000000000000000";
    MEM_2D      : bit_vector :=  X"0000000000000000000000000000000000000000000000000000000000000000";
    MEM_2E      : bit_vector :=  X"0000000000000000000000000000000000000000000000000000000000000000";
    MEM_2F      : bit_vector :=  X"0000000000000000000000000000000000000000000000000000000000000000";
    MEM_30      : bit_vector :=  X"0000000000000000000000000000000000000000000000000000000000000000";
    MEM_31      : bit_vector :=  X"0000000000000000000000000000000000000000000000000000000000000000";
    MEM_32      : bit_vector :=  X"0000000000000000000000000000000000000000000000000000000000000000";
    MEM_33      : bit_vector :=  X"0000000000000000000000000000000000000000000000000000000000000000";
    MEM_34      : bit_vector :=  X"0000000000000000000000000000000000000000000000000000000000000000";
    MEM_35      : bit_vector :=  X"0000000000000000000000000000000000000000000000000000000000000000";
    MEM_36      : bit_vector :=  X"0000000000000000000000000000000000000000000000000000000000000000";
    MEM_37      : bit_vector :=  X"0000000000000000000000000000000000000000000000000000000000000000";
    MEM_38      : bit_vector :=  X"0000000000000000000000000000000000000000000000000000000000000000";
    MEM_39      : bit_vector :=  X"0000000000000000000000000000000000000000000000000000000000000000";
    MEM_3A      : bit_vector :=  X"0000000000000000000000000000000000000000000000000000000000000000";
    MEM_3B      : bit_vector :=  X"0000000000000000000000000000000000000000000000000000000000000000";
    MEM_3C      : bit_vector :=  X"0000000000000000000000000000000000000000000000000000000000000000";
    MEM_3D      : bit_vector :=  X"0000000000000000000000000000000000000000000000000000000000000000";
    MEM_3E      : bit_vector :=  X"0000000000000000000000000000000000000000000000000000000000000000";
    MEM_3F      : bit_vector :=  X"0000000000000000000000000000000000000000000000000000000000000000";
    MEMP_00     : bit_vector :=  X"0000000000000000000000000000000000000000000000000000000000000000";
    MEMP_01     : bit_vector :=  X"0000000000000000000000000000000000000000000000000000000000000000";
    MEMP_02     : bit_vector :=  X"0000000000000000000000000000000000000000000000000000000000000000";
    MEMP_03     : bit_vector :=  X"0000000000000000000000000000000000000000000000000000000000000000";
    MEMP_04     : bit_vector :=  X"0000000000000000000000000000000000000000000000000000000000000000";
    MEMP_05     : bit_vector :=  X"0000000000000000000000000000000000000000000000000000000000000000";
    MEMP_06     : bit_vector :=  X"0000000000000000000000000000000000000000000000000000000000000000";
    MEMP_07     : bit_vector :=  X"0000000000000000000000000000000000000000000000000000000000000000"
);
port
(
    -- User Interface
    RX_DATA                 : in  std_logic_vector((RX_DATA_WIDTH-1) downto 0);
    RX_ENMCOMMA_ALIGN       : out std_logic;
    RX_ENPCOMMA_ALIGN       : out std_logic;
```

```
    RX_ENCHAN_SYNC          : out std_logic;
    RX_CHANBOND_SEQ         : in  std_logic;
    -- Control Interface
    INC_IN                  : in std_logic;
    INC_OUT                 : out std_logic;
    PATTERN_MATCH_N         : out std_logic;
    RESET_ON_ERROR          : in std_logic;
    -- Error Monitoring
    ERROR_COUNT             : out std_logic_vector(7 downto 0);
    -- System Interface
    USER_CLK                : in std_logic;
    SYSTEM_RESET            : in std_logic
);
end component;

component MGT_USRCLK_SOURCE_PLL
generic
(
    MULT                : integer         := 2;
    DIVIDE              : integer         := 2;
    CLK_PERIOD          : real            := 1.6;
    OUT0_DIVIDE         : integer         := 2;
    OUT1_DIVIDE         : integer         := 2;
    OUT2_DIVIDE         : integer         := 2;
    OUT3_DIVIDE         : integer         := 2;
    SIMULATION_P        : integer         := 1;
    LOCK_WAIT_COUNT     : std_logic_vector := "1000001000110101"
);
port
(
    CLK0_OUT            : out std_logic;
    CLK1_OUT            : out std_logic;
    CLK2_OUT            : out std_logic;
    CLK3_OUT            : out std_logic;
    CLK_IN              : in  std_logic;
    PLL_LOCKED_OUT      : out std_logic;
    PLL_RESET_IN        : in  std_logic
);
end component;

--***********************************Parameter Declarations********************
    constant DLY : time := 1 ns;
--*************************** Register Declarations ****************************
    signal   tile0_tx_resetdone0_r          : std_logic;
    signal   tile0_tx_resetdone0_r2         : std_logic;
    signal   tile0_rx_resetdone0_r          : std_logic;
    signal   tile0_rx_resetdone0_r2         : std_logic;
    signal   tile0_tx_resetdone1_r          : std_logic;
    signal   tile0_tx_resetdone1_r2         : std_logic;
    signal   tile0_rx_resetdone1_r          : std_logic;
    signal   tile0_rx_resetdone1_r2         : std_logic;
    signal   tile1_tx_resetdone0_r          : std_logic;
    signal   tile1_tx_resetdone0_r2         : std_logic;
    signal   tile1_rx_resetdone0_r          : std_logic;
    signal   tile1_rx_resetdone0_r2         : std_logic;
    signal   tile1_tx_resetdone1_r          : std_logic;
    signal   tile1_tx_resetdone1_r2         : std_logic;
    signal   tile1_rx_resetdone1_r          : std_logic;
    signal   tile1_rx_resetdone1_r2         : std_logic;
    signal   async_mux0_sel_i               : std_logic;
    signal   not_async_mux0_sel_i           : std_logic;
    signal   async_mux1_sel_i               : std_logic;
    signal   not_async_mux1_sel_i           : std_logic;
 --***************************** Wire Declarations ****************************
    ---------------------- MGT Wrapper Wires ------------------------------
    ------------------------ Loopback and Powerdown Ports ----------------------
    signal  tile0_loopback0_i               : std_logic_vector(2 downto 0);
    signal  tile0_loopback1_i               : std_logic_vector(2 downto 0);
    ---------------------- Receive Ports - 8b10b Decoder ----------------------
    signal  tile0_rxdisperr0_i              : std_logic_vector(1 downto 0);
```

```
signal  tile0_rxdisperr1_i              : std_logic_vector(1 downto 0);
signal  tile0_rxnotintable0_i           : std_logic_vector(1 downto 0);
signal  tile0_rxnotintable1_i           : std_logic_vector(1 downto 0);
---------- Receive Ports - Comma Detection and Alignment ----------
signal  tile0_rxenmcommaalign0_i        : std_logic;
signal  tile0_rxenmcommaalign1_i        : std_logic;
signal  tile0_rxenpcommaalign0_i        : std_logic;
signal  tile0_rxenpcommaalign1_i        : std_logic;
---------- Receive Ports - RX Data Path interface ----------
signal  tile0_rxdata0_i                 : std_logic_vector(15 downto 0);
signal  tile0_rxdata1_i                 : std_logic_vector(15 downto 0);
signal  tile0_rxreset0_i                : std_logic;
signal  tile0_rxreset1_i                : std_logic;
------- Receive Ports - RX Driver,OOB signalling,Coupling and Eq.,CDR ------
signal  tile0_rxeqmix0_i                : std_logic_vector(1 downto 0);
signal  tile0_rxeqmix1_i                : std_logic_vector(1 downto 0);
---------- Receive Ports - RX Loss-of-sync State Machine ----------
signal  tile0_rxlossofsync0_i           : std_logic_vector(1 downto 0);
signal  tile0_rxlossofsync1_i           : std_logic_vector(1 downto 0);
---------- Shared Ports - Tile and PLL Ports ----------
signal  tile0_gtxreset_i                : std_logic;
signal  tile0_plllkdet_i                : std_logic;
signal  tile0_refclkout_i               : std_logic;
signal  tile0_resetdone0_i              : std_logic;
signal  tile0_resetdone1_i              : std_logic;
---------- Transmit Ports - 8b10b Encoder Control Ports ----------
signal  tile0_txcharisk0_i              : std_logic_vector(1 downto 0);
signal  tile0_txcharisk1_i              : std_logic_vector(1 downto 0);
---------- Transmit Ports - TX Data Path interface ----------
signal  tile0_txdata0_i                 : std_logic_vector(15 downto 0);
signal  tile0_txdata1_i                 : std_logic_vector(15 downto 0);
signal  tile0_txreset0_i                : std_logic;
signal  tile0_txreset1_i                : std_logic;
---------- Transmit Ports - TX Driver and OOB signalling ----------
signal  tile0_txdiffctrl0_i             : std_logic_vector(2 downto 0);
signal  tile0_txdiffctrl1_i             : std_logic_vector(2 downto 0);
signal  tile0_txpreemphasis0_i          : std_logic_vector(2 downto 0);
signal  tile0_txpreemphasis1_i          : std_logic_vector(2 downto 0);
---------- Transmit Ports - TX PRBS Generator ----------
signal  tile0_txenprbstst0_i            : std_logic_vector(1 downto 0);
signal  tile0_txenprbstst1_i            : std_logic_vector(1 downto 0);
---------- Loopback and Powerdown Ports ----------
signal  tile1_loopback0_i               : std_logic_vector(2 downto 0);
signal  tile1_loopback1_i               : std_logic_vector(2 downto 0);
---------- Receive Ports - 8b10b Decoder ----------
signal  tile1_rxdisperr0_i              : std_logic_vector(1 downto 0);
signal  tile1_rxdisperr1_i              : std_logic_vector(1 downto 0);
signal  tile1_rxnotintable0_i           : std_logic_vector(1 downto 0);
signal  tile1_rxnotintable1_i           : std_logic_vector(1 downto 0);
---------- Receive Ports - Comma Detection and Alignment ----------
signal  tile1_rxenmcommaalign0_i        : std_logic;
signal  tile1_rxenmcommaalign1_i        : std_logic;
signal  tile1_rxenpcommaalign0_i        : std_logic;
signal  tile1_rxenpcommaalign1_i        : std_logic;
---------- Receive Ports - RX Data Path interface ----------
signal  tile1_rxdata0_i                 : std_logic_vector(15 downto 0);
signal  tile1_rxdata1_i                 : std_logic_vector(15 downto 0);
signal  tile1_rxreset0_i                : std_logic;
signal  tile1_rxreset1_i                : std_logic;
------- Receive Ports - RX Driver,OOB signalling,Coupling and Eq.,CDR ------
signal  tile1_rxeqmix0_i                : std_logic_vector(1 downto 0);
signal  tile1_rxeqmix1_i                : std_logic_vector(1 downto 0);
---------- Receive Ports - RX Loss-of-sync State Machine ----------
signal  tile1_rxlossofsync0_i           : std_logic_vector(1 downto 0);
signal  tile1_rxlossofsync1_i           : std_logic_vector(1 downto 0);
---------- Shared Ports - Tile and PLL Ports ----------
signal  tile1_gtxreset_i                : std_logic;
signal  tile1_plllkdet_i                : std_logic;
signal  tile1_refclkout_i               : std_logic;
signal  tile1_resetdone0_i              : std_logic;
```

```
signal  tile1_resetdone1_i              : std_logic;
---------------- Transmit Ports - 8b10b Encoder Control Ports --------------
signal  tile1_txcharisk0_i              : std_logic_vector(1 downto 0);
signal  tile1_txcharisk1_i              : std_logic_vector(1 downto 0);
------------------ Transmit Ports - TX Data Path interface -----------------
signal  tile1_txdata0_i                 : std_logic_vector(15 downto 0);
signal  tile1_txdata1_i                 : std_logic_vector(15 downto 0);
signal  tile1_txreset0_i                : std_logic;
signal  tile1_txreset1_i                : std_logic;
--------------- Transmit Ports - TX Driver and OOB signalling --------------
signal  tile1_txdiffctrl0_i             : std_logic_vector(2 downto 0);
signal  tile1_txdiffctrl1_i             : std_logic_vector(2 downto 0);
signal  tile1_txpreemphasis0_i          : std_logic_vector(2 downto 0);
signal  tile1_txpreemphasis1_i          : std_logic_vector(2 downto 0);
--------------------- Transmit Ports - TX PRBS Generator -------------------
signal  tile1_txenprbstst0_i            : std_logic_vector(1 downto 0);
signal  tile1_txenprbstst1_i            : std_logic_vector(1 downto 0);
------------------------------- Global Signals -----------------------------
signal  tile0_tx_system_reset0_c        : std_logic;
signal  tile0_rx_system_reset0_c        : std_logic;
signal  tile0_tx_system_reset1_c        : std_logic;
signal  tile0_rx_system_reset1_c        : std_logic;
signal  tile1_tx_system_reset0_c        : std_logic;
signal  tile1_rx_system_reset0_c        : std_logic;
signal  tile1_tx_system_reset1_c        : std_logic;
signal  tile1_rx_system_reset1_c        : std_logic;
signal  tied_to_ground_i                : std_logic;
signal  tied_to_ground_vec_i            : std_logic_vector(63 downto 0);
signal  tied_to_vcc_i                   : std_logic;
signal  tied_to_vcc_vec_i               : std_logic_vector(7 downto 0);
signal  tile0_refclkout_bufg_i          : std_logic;
---------------------------- User Clocks ---------------------------------
signal  tile0_txusrclk0_i               : std_logic;
signal  tile1_txusrclk0_i               : std_logic;
signal  refclkout_pll0_locked_i         : std_logic;
signal  refclkout_pll0_reset_i          : std_logic;
signal  tile0_refclkout_to_cmt_i        : std_logic;
signal  refclkout_pll1_locked_i         : std_logic;
signal  refclkout_pll1_reset_i          : std_logic;
signal  tile1_refclkout_to_cmt_i        : std_logic;
---------------------- Frame check/gen Module Signals --------------------
signal  tile0_refclk_i                  : std_logic;
signal  tile0_matchn0_i                 : std_logic;
signal  tile0_txcharisk0_float_i        : std_logic_vector(1 downto 0);
signal  tile0_txdata0_float_i           : std_logic_vector(23 downto 0);
signal  tile0_block_sync0_i             : std_logic;
signal  tile0_error_count0_i            : std_logic_vector(7 downto 0);
signal  tile0_frame_check0_reset_i      : std_logic;
signal  tile0_inc_in0_i                 : std_logic;
signal  tile0_inc_out0_i                : std_logic;
signal  tile0_unscrambled_data0_i       : std_logic_vector(15 downto 0);
signal  tile0_matchn1_i                 : std_logic;
signal  tile0_txcharisk1_float_i        : std_logic_vector(1 downto 0);
signal  tile0_txdata1_float_i           : std_logic_vector(23 downto 0);
signal  tile0_block_sync1_i             : std_logic;
signal  tile0_error_count1_i            : std_logic_vector(7 downto 0);
signal  tile0_frame_check1_reset_i      : std_logic;
signal  tile0_inc_in1_i                 : std_logic;
signal  tile0_inc_out1_i                : std_logic;
signal  tile0_unscrambled_data1_i       : std_logic_vector(15 downto 0);
signal  tile1_refclk_i                  : std_logic;
signal  tile1_matchn0_i                 : std_logic;
signal  tile1_txcharisk0_float_i        : std_logic_vector(1 downto 0);
signal  tile1_txdata0_float_i           : std_logic_vector(23 downto 0);
signal  tile1_block_sync0_i             : std_logic;
signal  tile1_error_count0_i            : std_logic_vector(7 downto 0);
signal  tile1_frame_check0_reset_i      : std_logic;
signal  tile1_inc_in0_i                 : std_logic;
signal  tile1_inc_out0_i                : std_logic;
signal  tile1_unscrambled_data0_i       : std_logic_vector(15 downto 0);
```

```
    signal  tile1_matchn1_i                : std_logic;
    signal  tile1_txcharisk1_float_i       : std_logic_vector(1 downto 0);
    signal  tile1_txdata1_float_i          : std_logic_vector(23 downto 0);
    signal  tile1_block_sync1_i            : std_logic;
    signal  tile1_error_count1_i           : std_logic_vector(7 downto 0);
    signal  tile1_frame_check1_reset_i     : std_logic;
    signal  tile1_inc_in1_i                : std_logic;
    signal  tile1_inc_out1_i               : std_logic;
    signal  tile1_unscrambled_data1_i      : std_logic_vector(15 downto 0);
    signal  reset_on_data_error_i          : std_logic;


--****************************** Main Body of Code *******************************
begin
    --  Static signal Assigments
    tied_to_ground_i                    <= '0';
    tied_to_ground_vec_i                <= x"0000000000000000";
    tied_to_vcc_i                       <= '1';
    tied_to_vcc_vec_i                   <= x"ff";

    tile0_refclk_ibufds_i : IBUFDS
    port map
    (
       O                        =>     tile0_refclk_i,
       I                        =>     TILE0_REFCLK_PAD_P_IN,
       IB                       =>     TILE0_REFCLK_PAD_N_IN
    );

    tile1_refclk_ibufds_i : IBUFDS
    port map
    (
       O                        =>     tile1_refclk_i,
       I                        =>     TILE1_REFCLK_PAD_P_IN,
       IB                       =>     TILE1_REFCLK_PAD_N_IN
    );

    ---------------------------------- User Clocks ---------------------------
    -- The clock resources in this section were added based on userclk source selections on
    -- the Latency, Buffering, and Clocking page of the GUI. A few notes about user clocks:
    -- * The userclk and userclk2 for each GTX datapath (TX and RX) must be phase aligned to
    --   avoid data errors in the fabric interface whenever the datapath is wider than 10 bits
    -- * To minimize clock resources, you can share clocks between GTXs. GTXs using the same frequency
    --   or multiples of the same frequency can be accomadated using DCMs and PLLs. Use caution when
    --   using RXRECCLK as a clock source, however - these clocks can typically only be shared if all
    --   the channels using the clock are receiving data from TX channels that share a reference clock
    --   source with each other.

    refclkout_pll0_bufg_i : BUFG
    port map
    (
       I                        =>     tile0_refclkout_i,
       O                        =>     tile0_refclkout_to_cmt_i
    );

    refclkout_pll0_reset_i                <= not tile0_plllkdet_i;
    refclkout_pll0_i : MGT_USRCLK_SOURCE_PLL
    generic map
    (
       MULT                     =>     1,
       DIVIDE                   =>     1,
       CLK_PERIOD               =>     1.6,
       OUT0_DIVIDE              =>     2,
       OUT1_DIVIDE              =>     1,
       OUT2_DIVIDE              =>     1,
       OUT3_DIVIDE              =>     1,
       SIMULATION_P             =>     EXAMPLE_USE_CHIPSCOPE,
       LOCK_WAIT_COUNT          =>     "1111010000100100"
    )
    port map
    (
```

```
   CLK0_OUT                    =>     tile0_txusrclk0_i,
   CLK1_OUT                    =>     open,
   CLK2_OUT                    =>     open,
   CLK3_OUT                    =>     open,
   CLK_IN                   =>     tile0_refclkout_to_cmt_i,
   PLL_LOCKED_OUT              =>     refclkout_pll0_locked_i,
   PLL_RESET_IN             =>     refclkout_pll0_reset_i
);


refclkout_pll1_bufg_i : BUFG
port map
(
   I                     =>     tile1_refclkout_i,
   O                      =>     tile1_refclkout_to_cmt_i
);

refclkout_pll1_reset_i              <= not tile1_plllkdet_i;
refclkout_pll1_i : MGT_USRCLK_SOURCE_PLL
generic map
(
   MULT                    =>     1,
   DIVIDE                  =>     1,
   CLK_PERIOD                 =>     1.6,
   OUT0_DIVIDE                =>     2,
   OUT1_DIVIDE                =>     1,
   OUT2_DIVIDE                =>     1,
   OUT3_DIVIDE                =>     1,
   SIMULATION_P               =>     EXAMPLE_USE_CHIPSCOPE,
   LOCK_WAIT_COUNT            =>     "1111010000100100"
)
port map
(
   CLK0_OUT                    =>     tile1_txusrclk0_i,
   CLK1_OUT                    =>     open,
   CLK2_OUT                    =>     open,
   CLK3_OUT                    =>     open,
   CLK_IN                   =>     tile1_refclkout_to_cmt_i,
   PLL_LOCKED_OUT              =>     refclkout_pll1_locked_i,
   PLL_RESET_IN             =>     refclkout_pll1_reset_i
);
----------------------------- The GTX Wrapper -----------------------------

-- Use the instantiation template in the examples directory to add the GTX wrapper to your design.
-- In this example, the wrapper is wired up for basic operation with a frame generator and frame
-- checker. The GTXs will reset, then attempt to align and transmit data. If channel bonding is
-- enabled, bonding should occur after alignment.
-- Wire all PLLLKDET signals to the top level as output ports
TILE0_PLLLKDET_OUT              <= tile0_plllkdet_i;
TILE1_PLLLKDET_OUT              <= tile1_plllkdet_i;

-- Hold the TX in reset till the TX user clocks are stable
tile0_txreset0_i            <= not refclkout_pll0_locked_i;
tile0_txreset1_i            <= not refclkout_pll0_locked_i;
tile1_txreset0_i            <= not refclkout_pll1_locked_i;
tile1_txreset1_i            <= not refclkout_pll1_locked_i;

-- Hold the RX in reset till the RX user clocks are stable
tile0_rxreset0_i            <= not refclkout_pll0_locked_i;
tile0_rxreset1_i            <= not refclkout_pll0_locked_i;
tile1_rxreset0_i            <= not refclkout_pll1_locked_i;
tile1_rxreset1_i            <= not refclkout_pll1_locked_i;

rocketio_wrapper_i : ROCKETIO_WRAPPER
generic map
(
   WRAPPER_SIM_MODE            =>     EXAMPLE_SIM_MODE,
   WRAPPER_SIM_GTXRESET_SPEEDUP    =>     EXAMPLE_SIM_GTXRESET_SPEEDUP,
   WRAPPER_SIM_PLL_PERDIV2        =>     EXAMPLE_SIM_PLL_PERDIV2
)
```

```
port map
(
    ------------------------ Loopback and Powerdown Ports ----------------------
    TILE0_LOOPBACK0_IN              =>     tile0_loopback0_i,
    TILE0_LOOPBACK1_IN              =>     tile0_loopback1_i,
    ------------------------ Receive Ports - 8b10b Decoder ----------------------
    TILE0_RXDISPERR0_OUT            =>     tile0_rxdisperr0_i,
    TILE0_RXDISPERR1_OUT            =>     tile0_rxdisperr1_i,
    TILE0_RXNOTINTABLE0_OUT         =>     tile0_rxnotintable0_i,
    TILE0_RXNOTINTABLE1_OUT         =>     tile0_rxnotintable1_i,
    --------------- Receive Ports - Comma Detection and Alignment --------------
    TILE0_RXENMCOMMAALIGN0_IN       =>     tile0_rxenmcommaalign0_i,
    TILE0_RXENMCOMMAALIGN1_IN       =>     tile0_rxenmcommaalign1_i,
    TILE0_RXENPCOMMAALIGN0_IN       =>     tile0_rxenpcommaalign0_i,
    TILE0_RXENPCOMMAALIGN1_IN       =>     tile0_rxenpcommaalign1_i,
    ------------------ Receive Ports - RX Data Path interface -----------------
    TILE0_RXDATA0_OUT               =>     tile0_rxdata0_i,
    TILE0_RXDATA1_OUT               =>     tile0_rxdata1_i,
    TILE0_RXRESET0_IN               =>     tile0_rxreset0_i,
    TILE0_RXRESET1_IN               =>     tile0_rxreset1_i,
    TILE0_RXUSRCLK0_IN              =>     tile0_txusrclk0_i,
    TILE0_RXUSRCLK1_IN              =>     tile0_txusrclk0_i,
    TILE0_RXUSRCLK20_IN             =>     tile0_txusrclk0_i,
    TILE0_RXUSRCLK21_IN             =>     tile0_txusrclk0_i,
    ------- Receive Ports - RX Driver,OOB signalling,Coupling and Eq.,CDR ------
    TILE0_RXEQMIX0_IN               =>     tile0_rxeqmix0_i,
    TILE0_RXEQMIX1_IN               =>     tile0_rxeqmix1_i,
    TILE0_RXN0_IN                   =>     RXN_IN(0),
    TILE0_RXN1_IN                   =>     RXN_IN(1),
    TILE0_RXP0_IN                   =>     RXP_IN(0),
    TILE0_RXP1_IN                   =>     RXP_IN(1),
    --------------- Receive Ports - RX Loss-of-sync State Machine --------------
    TILE0_RXLOSSOFSYNC0_OUT         =>     tile0_rxlossofsync0_i,
    TILE0_RXLOSSOFSYNC1_OUT         =>     tile0_rxlossofsync1_i,
    ------------- Shared Ports - Dynamic Reconfiguration Port (DRP) ------------
    TILE0_DADDR_IN                  =>     tied_to_ground_vec_i(6 downto 0),
    --TILE0_DCLK_IN                 =>     drp_clk_in_i,
    TILE0_DEN_IN                    =>     tied_to_ground_i,
    TILE0_DI_IN                     =>     tied_to_ground_vec_i(15 downto 0),
    TILE0_DO_OUT                    =>     open,
    TILE0_DRDY_OUT                  =>     open,
    TILE0_DWE_IN                    =>     tied_to_ground_i,
    ---------------------- Shared Ports - Tile and PLL Ports --------------------
    TILE0_CLKIN_IN                  =>     tile0_refclk_i,
    TILE0_GTXRESET_IN               =>     tile0_gtxreset_i,
    TILE0_PLLLKDET_OUT              =>     tile0_plllkdet_i,
    TILE0_REFCLKOUT_OUT             =>     tile0_refclkout_i,
    TILE0_RESETDONE0_OUT            =>     tile0_resetdone0_i,
    TILE0_RESETDONE1_OUT            =>     tile0_resetdone1_i,
    ---------------- Transmit Ports - 8b10b Encoder Control Ports --------------
    TILE0_TXCHARISK0_IN             =>     tile0_txcharisk0_i,
    TILE0_TXCHARISK1_IN             =>     tile0_txcharisk1_i,
    ------------------ Transmit Ports - TX Data Path interface -----------------
    TILE0_TXDATA0_IN                =>     tile0_txdata0_i,
    TILE0_TXDATA1_IN                =>     tile0_txdata1_i,
    TILE0_TXRESET0_IN               =>     tile0_txreset0_i,
    TILE0_TXRESET1_IN               =>     tile0_txreset1_i,
    TILE0_TXUSRCLK0_IN              =>     tile0_txusrclk0_i,
    TILE0_TXUSRCLK1_IN              =>     tile0_txusrclk0_i,
    TILE0_TXUSRCLK20_IN             =>     tile0_txusrclk0_i,
    TILE0_TXUSRCLK21_IN             =>     tile0_txusrclk0_i,
    --------------- Transmit Ports - TX Driver and OOB signalling --------------
    TILE0_TXDIFFCTRL0_IN            =>     tile0_txdiffctrl0_i,
    TILE0_TXDIFFCTRL1_IN            =>     tile0_txdiffctrl1_i,
    TILE0_TXN0_OUT                  =>     TXN_OUT(0),
    TILE0_TXN1_OUT                  =>     TXN_OUT(1),
    TILE0_TXP0_OUT                  =>     TXP_OUT(0),
    TILE0_TXP1_OUT                  =>     TXP_OUT(1),
    TILE0_TXPREEMPHASIS0_IN         =>     tile0_txpreemphasis0_i,
```

```
TILE0_TXPREEMPHASIS1_IN         =>      tile0_txpreemphasis1_i,
---------------------- Transmit Ports - TX PRBS Generator -------------------
TILE0_TXENPRBSTST0_IN           =>      tile0_txenprbstst0_i,
TILE0_TXENPRBSTST1_IN           =>      tile0_txenprbstst1_i,
------------------------ Loopback and Powerdown Ports ----------------------
TILE1_LOOPBACK0_IN              =>      tile1_loopback0_i,
TILE1_LOOPBACK1_IN              =>      tile1_loopback1_i,
----------------------- Receive Ports - 8b10b Decoder ----------------------
TILE1_RXDISPERR0_OUT            =>      tile1_rxdisperr0_i,
TILE1_RXDISPERR1_OUT            =>      tile1_rxdisperr1_i,
TILE1_RXNOTINTABLE0_OUT         =>      tile1_rxnotintable0_i,
TILE1_RXNOTINTABLE1_OUT         =>      tile1_rxnotintable1_i,
---------------- Receive Ports - Comma Detection and Alignment --------------
TILE1_RXENMCOMMAALIGN0_IN       =>      tile1_rxenmcommaalign0_i,
TILE1_RXENMCOMMAALIGN1_IN       =>      tile1_rxenmcommaalign1_i,
TILE1_RXENPCOMMAALIGN0_IN       =>      tile1_rxenpcommaalign0_i,
TILE1_RXENPCOMMAALIGN1_IN       =>      tile1_rxenpcommaalign1_i,
------------------- Receive Ports - RX Data Path interface -----------------
TILE1_RXDATA0_OUT               =>      tile1_rxdata0_i,
TILE1_RXDATA1_OUT               =>      tile1_rxdata1_i,
TILE1_RXRESET0_IN               =>      tile1_rxreset0_i,
TILE1_RXRESET1_IN               =>      tile1_rxreset1_i,
TILE1_RXUSRCLK0_IN              =>      tile1_txusrclk0_i,
TILE1_RXUSRCLK1_IN              =>      tile1_txusrclk0_i,
TILE1_RXUSRCLK20_IN             =>      tile1_txusrclk0_i,
TILE1_RXUSRCLK21_IN             =>      tile1_txusrclk0_i,
------- Receive Ports - RX Driver,OOB signalling,Coupling and Eq.,CDR ------
TILE1_RXEQMIX0_IN               =>      tile1_rxeqmix0_i,
TILE1_RXEQMIX1_IN               =>      tile1_rxeqmix1_i,
TILE1_RXN0_IN                   =>      RXN_IN(2),
TILE1_RXN1_IN                   =>      RXN_IN(3),
TILE1_RXP0_IN                   =>      RXP_IN(2),
TILE1_RXP1_IN                   =>      RXP_IN(3),
---------------- Receive Ports - RX Loss-of-sync State Machine --------------
TILE1_RXLOSSOFSYNC0_OUT         =>      tile1_rxlossofsync0_i,
TILE1_RXLOSSOFSYNC1_OUT         =>      tile1_rxlossofsync1_i,
-------------- Shared Ports - Dynamic Reconfiguration Port (DRP) ------------
TILE1_DADDR_IN                  =>      tied_to_ground_vec_i(6 downto 0),
--TILE1_DCLK_IN                 =>      drp_clk_in_i,
TILE1_DEN_IN                    =>      tied_to_ground_i,
TILE1_DI_IN                     =>      tied_to_ground_vec_i(15 downto 0),
TILE1_DO_OUT                    =>      open,
TILE1_DRDY_OUT                  =>      open,
TILE1_DWE_IN                    =>      tied_to_ground_i,
---------------------- Shared Ports - Tile and PLL Ports --------------------
TILE1_CLKIN_IN                  =>      tile1_refclk_i,
TILE1_GTXRESET_IN               =>      tile1_gtxreset_i,
TILE1_PLLLKDET_OUT              =>      tile1_plllkdet_i,
TILE1_REFCLKOUT_OUT             =>      tile1_refclkout_i,
TILE1_RESETDONE0_OUT            =>      tile1_resetdone0_i,
TILE1_RESETDONE1_OUT            =>      tile1_resetdone1_i,
---------------- Transmit Ports - 8b10b Encoder Control Ports --------------
TILE1_TXCHARISK0_IN             =>      tile1_txcharisk0_i,
TILE1_TXCHARISK1_IN             =>      tile1_txcharisk1_i,
------------------ Transmit Ports - TX Data Path interface -----------------
TILE1_TXDATA0_IN                =>      tile1_txdata0_i,
TILE1_TXDATA1_IN                =>      tile1_txdata1_i,
TILE1_TXRESET0_IN               =>      tile1_txreset0_i,
TILE1_TXRESET1_IN               =>      tile1_txreset1_i,
TILE1_TXUSRCLK0_IN              =>      tile1_txusrclk0_i,
TILE1_TXUSRCLK1_IN              =>      tile1_txusrclk0_i,
TILE1_TXUSRCLK20_IN             =>      tile1_txusrclk0_i,
TILE1_TXUSRCLK21_IN             =>      tile1_txusrclk0_i,
---------------- Transmit Ports - TX Driver and OOB signalling --------------
TILE1_TXDIFFCTRL0_IN            =>      tile1_txdiffctrl0_i,
TILE1_TXDIFFCTRL1_IN            =>      tile1_txdiffctrl1_i,
TILE1_TXN0_OUT                  =>      TXN_OUT(2),
TILE1_TXN1_OUT                  =>      TXN_OUT(3),
TILE1_TXP0_OUT                  =>      TXP_OUT(2),
TILE1_TXP1_OUT                  =>      TXP_OUT(3),
```

```
    TILE1_TXPREEMPHASIS0_IN         =>      tile1_txpreemphasis0_i,
    TILE1_TXPREEMPHASIS1_IN         =>      tile1_txpreemphasis1_i,
    ---------------------- Transmit Ports - TX PRBS Generator -------------------
    TILE1_TXENPRBSTST0_IN           =>      tile1_txenprbstst0_i,
    TILE1_TXENPRBSTST1_IN           =>      tile1_txenprbstst1_i
);

--------------------------- User Module Resets -----------------------------
-- All the User Modules i.e. FRAME_GEN, FRAME_CHECK and the sync modules
-- are held in reset till the RESETDONE goes high.
-- The RESETDONE is registered a couple of times on USRCLK2 and connected
-- to the reset of the modules
process( tile0_txusrclk0_i,tile0_resetdone0_i)
begin
    if(tile0_resetdone0_i = '0') then
        tile0_rx_resetdone0_r  <= '0'   after DLY;
        tile0_rx_resetdone0_r2 <= '0'   after DLY;
    elsif(tile0_txusrclk0_i'event and tile0_txusrclk0_i = '1') then
        tile0_rx_resetdone0_r  <= tile0_resetdone0_i   after DLY;
        tile0_rx_resetdone0_r2 <= tile0_rx_resetdone0_r   after DLY;
    end if;
end process;
process( tile0_txusrclk0_i,tile0_resetdone0_i)
begin
    if(tile0_resetdone0_i = '0') then
        tile0_tx_resetdone0_r  <= '0'   after DLY;
        tile0_tx_resetdone0_r2 <= '0'   after DLY;
    elsif(tile0_txusrclk0_i'event and tile0_txusrclk0_i = '1') then
        tile0_tx_resetdone0_r  <= tile0_resetdone0_i   after DLY;
        tile0_tx_resetdone0_r2 <= tile0_tx_resetdone0_r   after DLY;
    end if;
end process;
process( tile0_txusrclk0_i,tile0_resetdone1_i)
begin
    if(tile0_resetdone1_i = '0') then
        tile0_rx_resetdone1_r  <= '0'   after DLY;
        tile0_rx_resetdone1_r2 <= '0'   after DLY;
    elsif(tile0_txusrclk0_i'event and tile0_txusrclk0_i = '1') then
        tile0_rx_resetdone1_r  <= tile0_resetdone1_i   after DLY;
        tile0_rx_resetdone1_r2 <= tile0_rx_resetdone1_r   after DLY;
    end if;
end process;
process( tile0_txusrclk0_i,tile0_resetdone1_i)
begin
    if(tile0_resetdone1_i = '0') then
        tile0_tx_resetdone1_r  <= '0'   after DLY;
        tile0_tx_resetdone1_r2 <= '0'   after DLY;
    elsif(tile0_txusrclk0_i'event and tile0_txusrclk0_i = '1') then
        tile0_tx_resetdone1_r  <= tile0_resetdone1_i   after DLY;
        tile0_tx_resetdone1_r2 <= tile0_tx_resetdone1_r   after DLY;
    end if;
end process;
process( tile1_txusrclk0_i,tile1_resetdone0_i)
begin
    if(tile1_resetdone0_i = '0') then
        tile1_rx_resetdone0_r  <= '0'   after DLY;
        tile1_rx_resetdone0_r2 <= '0'   after DLY;
    elsif(tile1_txusrclk0_i'event and tile1_txusrclk0_i = '1') then
        tile1_rx_resetdone0_r  <= tile1_resetdone0_i   after DLY;
        tile1_rx_resetdone0_r2 <= tile1_rx_resetdone0_r   after DLY;
    end if;
end process;
process( tile1_txusrclk0_i,tile1_resetdone0_i)
begin
    if(tile1_resetdone0_i = '0') then
        tile1_tx_resetdone0_r  <= '0'   after DLY;
        tile1_tx_resetdone0_r2 <= '0'   after DLY;
    elsif(tile1_txusrclk0_i'event and tile1_txusrclk0_i = '1') then
        tile1_tx_resetdone0_r  <= tile1_resetdone0_i   after DLY;
        tile1_tx_resetdone0_r2 <= tile1_tx_resetdone0_r   after DLY;
```

```
    end if;
end process;
process( tile1_txusrclk0_i,tile1_resetdone1_i)
begin
    if(tile1_resetdone1_i = '0') then
        tile1_rx_resetdone1_r  <= '0'   after DLY;
        tile1_rx_resetdone1_r2 <= '0'   after DLY;
    elsif(tile1_txusrclk0_i'event and tile1_txusrclk0_i = '1') then
        tile1_rx_resetdone1_r  <= tile1_resetdone1_i   after DLY;
        tile1_rx_resetdone1_r2 <= tile1_rx_resetdone1_r   after DLY;
    end if;
end process;
process( tile1_txusrclk0_i,tile1_resetdone1_i)
begin
    if(tile1_resetdone1_i = '0') then
        tile1_tx_resetdone1_r  <= '0'   after DLY;
        tile1_tx_resetdone1_r2 <= '0'   after DLY;
    elsif(tile1_txusrclk0_i'event and tile1_txusrclk0_i = '1') then
        tile1_tx_resetdone1_r  <= tile1_resetdone1_i   after DLY;
        tile1_tx_resetdone1_r2 <= tile1_tx_resetdone1_r   after DLY;
    end if;
end process;
------------------------------ Frame Generators ---------------------------
-- The example design uses Block RAM based frame generators to provide test
-- data to the GTXs for transmission. By default the frame generators are
-- loaded with an incrementing data sequence that includes commas/alignment
-- characters for alignment. If your protocol uses channel bonding, the
-- frame generator will also be preloaded with a channel bonding sequence.

-- You can modify the data transmitted by changing the INIT values of the frame
-- generator in this file. Pay careful attention to bit order and the spacing
-- of your control and alignment characters.

tile0_frame_gen0 : FRAME_GEN
generic map
(
    WORDS_IN_BRAM                =>     EXAMPLE_WORDS_IN_BRAM,
    MEM_00              =>  x"00000e0d00000c0b00000a0900000807000006050000040300000 2bc00000100",
    MEM_01              =>  x"00001e1d00001c1b00001a1900001817000016150000141300001 2110000100f",
    MEM_02              =>  x"00002e2d00002c2b00002a2900002827000026250000242300002 2210000201f",
    MEM_03              =>  x"00003e3d00003c3b00003a3900003837000036350000343300003 2310000302f",
    MEM_04              =>  x"00004e4d00004c4b00004a4900004847000046450000444300004 2410000403f",
    MEM_05              =>  x"00005e5d00005c5b00005a5900005857000056550000545300005 2510000504f",
    MEM_06              =>  x"00006e6d00006c6b00006a6900006867000066650000646300006 2610000605f",
    MEM_07              =>  x"00007e7d00007c7b00007a7900007877000076750000747300007 2710000706f",
    MEM_08              =>  x"00000e0d00000c0b00000a0900000807000006050000040300000 2bc00000100",
    MEM_09              =>  x"00001e1d00001c1b00001a1900001817000016150000141300001 2110000100f",
    MEM_0A               =>  x"00002e2d00002c2b00002a2900002827000026250000242300002 2210000201f",
    MEM_0B               =>  x"00003e3d00003c3b00003a3900003837000036350000343300003 2310000302f",
    MEM_0C               =>  x"00004e4d00004c4b00004a4900004847000046450000444300004 2410000403f",
    MEM_0D               =>  x"00005e5d00005c5b00005a5900005857000056550000545300005 2510000504f",
    MEM_0E               =>  x"00006e6d00006c6b00006a6900006867000066650000646300006 2610000605f",
    MEM_0F               =>  x"00007e7d00007c7b00007a7900007877000076750000747300007 2710000706f",
    MEM_10              =>  x"00000e0d00000c0b00000a0900000807000006050000040300000 2bc00000100",
    MEM_11              =>  x"00001e1d00001c1b00001a1900001817000016150000141300001 2110000100f",
    MEM_12              =>  x"00002e2d00002c2b00002a2900002827000026250000242300002 2210000201f",
    MEM_13              =>  x"00003e3d00003c3b00003a3900003837000036350000343300003 2310000302f",
    MEM_14              =>  x"00004e4d00004c4b00004a4900004847000046450000444300004 2410000403f",
    MEM_15              =>  x"00005e5d00005c5b00005a5900005857000056550000545300005 2510000504f",
    MEM_16              =>  x"00006e6d00006c6b00006a6900006867000066650000646300006 2610000605f",
    MEM_17              =>  x"00007e7d00007c7b00007a7900007877000076750000747300007 2710000706f",
    MEM_18              =>  x"00000e0d00000c0b00000a0900000807000006050000040300000 2bc00000100",
    MEM_19              =>  x"00001e1d00001c1b00001a1900001817000016150000141300001 2110000100f",
    MEM_1A               =>  x"00002e2d00002c2b00002a2900002827000026250000242300002 2210000201f",
    MEM_1B               =>  x"00003e3d00003c3b00003a3900003837000036350000343300003 2310000302f",
    MEM_1C               =>  x"00004e4d00004c4b00004a4900004847000046450000444300004 2410000403f",
    MEM_1D               =>  x"00005e5d00005c5b00005a5900005857000056550000545300005 2510000504f",
    MEM_1E               =>  x"00006e6d00006c6b00006a6900006867000066650000646300006 2610000605f",
    MEM_1F               =>  x"00007e7d00007c7b00007a7900007877000076750000747300007 2710000706f",
    MEM_20              =>  x"00000e0d00000c0b00000a0900000807000006050000040300000 2bc00000100",
```

```
    MEM_21              => x"00001e1d00001c1b00001a1900001817000016150000141300001211000010f",
    MEM_22              => x"00002e2d00002c2b00002a2900002827000026250000242300002221000020f",
    MEM_23              => x"00003e3d00003c3b00003a3900003837000036350000343300003231000030f",
    MEM_24              => x"00004e4d00004c4b00004a4900004847000046450000444300004241000040f",
    MEM_25              => x"00005e5d00005c5b00005a5900005857000056550000545300005251000050f",
    MEM_26              => x"00006e6d00006c6b00006a6900006867000066650000646300006261000060f",
    MEM_27              => x"00007e7d00007c7b00007a7900007877000076750000747300007271000070f",
    MEM_28              => x"00000e0d00000c0b00000a0900000807000006050000040300000 2bc00000100",
    MEM_29              => x"00001e1d00001c1b00001a1900001817000016150000141300001211000010f",
    MEM_2A              => x"00002e2d00002c2b00002a2900002827000026250000242300002221000020f",
    MEM_2B              => x"00003e3d00003c3b00003a3900003837000036350000343300003231000030f",
    MEM_2C              => x"00004e4d00004c4b00004a4900004847000046450000444300004241000040f",
    MEM_2D              => x"00005e5d00005c5b00005a5900005857000056550000545300005251000050f",
    MEM_2E              => x"00006e6d00006c6b00006a6900006867000066650000646300006261000060f",
    MEM_2F              => x"00007e7d00007c7b00007a7900007877000076750000747300007271000070f",
    MEM_30              => x"00000e0d00000c0b00000a0900000807000006050000040300000 2bc00000100",
    MEM_31              => x"00001e1d00001c1b00001a1900001817000016150000141300001211000010f",
    MEM_32              => x"00002e2d00002c2b00002a2900002827000026250000242300002221000020f",
    MEM_33              => x"00003e3d00003c3b00003a3900003837000036350000343300003231000030f",
    MEM_34              => x"00004e4d00004c4b00004a4900004847000046450000444300004241000040f",
    MEM_35              => x"00005e5d00005c5b00005a5900005857000056550000545300005251000050f",
    MEM_36              => x"00006e6d00006c6b00006a6900006867000066650000646300006261000060f",
    MEM_37              => x"00007e7d00007c7b00007a7900007877000076750000747300007271000070f",
    MEM_38              => x"00000e0d00000c0b00000a0900000807000006050000040300000 2bc00000100",
    MEM_39              => x"00001e1d00001c1b00001a1900001817000016150000141300001211000010f",
    MEM_3A              => x"00002e2d00002c2b00002a2900002827000026250000242300002221000020f",
    MEM_3B              => x"00003e3d00003c3b00003a3900003837000036350000343300003231000030f",
    MEM_3C              => x"00004e4d00004c4b00004a4900004847000046450000444300004241000040f",
    MEM_3D              => x"00005e5d00005c5b00005a5900005857000056550000545300005251000050f",
    MEM_3E              => x"00006e6d00006c6b00006a6900006867000066650000646300006261000060f",
    MEM_3F              => x"00007e7d00007c7b00007a7900007877000076750000747300007271000070f",
    MEMP_00             => x"0000000000000000000000000000000000000000000000000000000000000010",
    MEMP_01             => x"0000000000000000000000000000000000000000000000000000000000000010",
    MEMP_02             => x"0000000000000000000000000000000000000000000000000000000000000010",
    MEMP_03             => x"0000000000000000000000000000000000000000000000000000000000000010",
    MEMP_04             => x"0000000000000000000000000000000000000000000000000000000000000010",
    MEMP_05             => x"0000000000000000000000000000000000000000000000000000000000000010",
    MEMP_06             => x"0000000000000000000000000000000000000000000000000000000000000010",
    MEMP_07             => x"0000000000000000000000000000000000000000000000000000000000000010"
)
port map
(
   -- User Interface
   TX_DATA(39 downto 16)        =>    tile0_txdata0_float_i,
   TX_DATA(15 downto 0)         =>    tile0_txdata0_i,

   TX_CHARISK(3 downto 2)       =>    tile0_txcharisk0_float_i,
   TX_CHARISK(1 downto 0)       =>    tile0_txcharisk0_i,
   -- System Interface
   USER_CLK                     =>    tile0_txusrclk0_i,
   SYSTEM_RESET                 =>    tile0_tx_system_reset0_c
);

tile0_frame_gen1 : FRAME_GEN
generic map
(
   WORDS_IN_BRAM               =>    EXAMPLE_WORDS_IN_BRAM,
   MEM_00              => x"00000e0d00000c0b00000a0900000807000006050000040300000 2bc00000100",
   MEM_01              => x"00001e1d00001c1b00001a1900001817000016150000141300001211000010f",
   MEM_02              => x"00002e2d00002c2b00002a2900002827000026250000242300002221000020f",
   MEM_03              => x"00003e3d00003c3b00003a3900003837000036350000343300003231000030f",
   MEM_04              => x"00004e4d00004c4b00004a4900004847000046450000444300004241000040f",
   MEM_05              => x"00005e5d00005c5b00005a5900005857000056550000545300005251000050f",
   MEM_06              => x"00006e6d00006c6b00006a6900006867000066650000646300006261000060f",
   MEM_07              => x"00007e7d00007c7b00007a7900007877000076750000747300007271000070f",
   MEM_08              => x"00000e0d00000c0b00000a0900000807000006050000040300000 2bc00000100",
   MEM_09              => x"00001e1d00001c1b00001a1900001817000016150000141300001211000010f",
   MEM_0A              => x"00002e2d00002c2b00002a2900002827000026250000242300002221000020f",
   MEM_0B              => x"00003e3d00003c3b00003a3900003837000036350000343300003231000030f",
   MEM_0C              => x"00004e4d00004c4b00004a4900004847000046450000444300004241000040f",
```

```
    MEM_0D              => x"00005e5d00005c5b00005a5900005857000056550000545300005251000050 4f",
    MEM_0E              => x"00006e6d00006c6b00006a6900006867000066650000646300006261000060 5f",
    MEM_0F              => x"00007e7d00007c7b00007a7900007877000076750000747300007271000070 6f",
    MEM_10              => x"00000e0d00000c0b00000a0900000807000006050000040300000 2bc00000100",
    MEM_11              => x"00001e1d00001c1b00001a1900001817000016150000141300001211000010 0f",
    MEM_12              => x"00002e2d00002c2b00002a2900002827000026250000242300002221000020 1f",
    MEM_13              => x"00003e3d00003c3b00003a3900003837000036350000343300003231000030 2f",
    MEM_14              => x"00004e4d00004c4b00004a4900004847000046450000444300004241000040 3f",
    MEM_15              => x"00005e5d00005c5b00005a5900005857000056550000545300005251000050 4f",
    MEM_16              => x"00006e6d00006c6b00006a6900006867000066650000646300006261000060 5f",
    MEM_17              => x"00007e7d00007c7b00007a7900007877000076750000747300007271000070 6f",
    MEM_18              => x"00000e0d00000c0b00000a0900000807000006050000040300000 2bc00000100",
    MEM_19              => x"00001e1d00001c1b00001a1900001817000016150000141300001211000010 0f",
    MEM_1A              => x"00002e2d00002c2b00002a2900002827000026250000242300002221000020 1f",
    MEM_1B              => x"00003e3d00003c3b00003a3900003837000036350000343300003231000030 2f",
    MEM_1C              => x"00004e4d00004c4b00004a4900004847000046450000444300004241000040 3f",
    MEM_1D              => x"00005e5d00005c5b00005a5900005857000056550000545300005251000050 4f",
    MEM_1E              => x"00006e6d00006c6b00006a6900006867000066650000646300006261000060 5f",
    MEM_1F              => x"00007e7d00007c7b00007a7900007877000076750000747300007271000070 6f",
    MEM_20              => x"00000e0d00000c0b00000a0900000807000006050000040300000 2bc00000100",
    MEM_21              => x"00001e1d00001c1b00001a1900001817000016150000141300001211000010 0f",
    MEM_22              => x"00002e2d00002c2b00002a2900002827000026250000242300002221000020 1f",
    MEM_23              => x"00003e3d00003c3b00003a3900003837000036350000343300003231000030 2f",
    MEM_24              => x"00004e4d00004c4b00004a4900004847000046450000444300004241000040 3f",
    MEM_25              => x"00005e5d00005c5b00005a5900005857000056550000545300005251000050 4f",
    MEM_26              => x"00006e6d00006c6b00006a6900006867000066650000646300006261000060 5f",
    MEM_27              => x"00007e7d00007c7b00007a7900007877000076750000747300007271000070 6f",
    MEM_28              => x"00000e0d00000c0b00000a0900000807000006050000040300000 2bc00000100",
    MEM_29              => x"00001e1d00001c1b00001a1900001817000016150000141300001211000010 0f",
    MEM_2A              => x"00002e2d00002c2b00002a2900002827000026250000242300002221000020 1f",
    MEM_2B              => x"00003e3d00003c3b00003a3900003837000036350000343300003231000030 2f",
    MEM_2C              => x"00004e4d00004c4b00004a4900004847000046450000444300004241000040 3f",
    MEM_2D              => x"00005e5d00005c5b00005a5900005857000056550000545300005251000050 4f",
    MEM_2E              => x"00006e6d00006c6b00006a6900006867000066650000646300006261000060 5f",
    MEM_2F              => x"00007e7d00007c7b00007a7900007877000076750000747300007271000070 6f",
    MEM_30              => x"00000e0d00000c0b00000a0900000807000006050000040300000 2bc00000100",
    MEM_31              => x"00001e1d00001c1b00001a1900001817000016150000141300001211000010 0f",
    MEM_32              => x"00002e2d00002c2b00002a2900002827000026250000242300002221000020 1f",
    MEM_33              => x"00003e3d00003c3b00003a3900003837000036350000343300003231000030 2f",
    MEM_34              => x"00004e4d00004c4b00004a4900004847000046450000444300004241000040 3f",
    MEM_35              => x"00005e5d00005c5b00005a5900005857000056550000545300005251000050 4f",
    MEM_36              => x"00006e6d00006c6b00006a6900006867000066650000646300006261000060 5f",
    MEM_37              => x"00007e7d00007c7b00007a7900007877000076750000747300007271000070 6f",
    MEM_38              => x"00000e0d00000c0b00000a0900000807000006050000040300000 2bc00000100",
    MEM_39              => x"00001e1d00001c1b00001a1900001817000016150000141300001211000010 0f",
    MEM_3A              => x"00002e2d00002c2b00002a2900002827000026250000242300002221000020 1f",
    MEM_3B              => x"00003e3d00003c3b00003a3900003837000036350000343300003231000030 2f",
    MEM_3C              => x"00004e4d00004c4b00004a4900004847000046450000444300004241000040 3f",
    MEM_3D              => x"00005e5d00005c5b00005a5900005857000056550000545300005251000050 4f",
    MEM_3E              => x"00006e6d00006c6b00006a6900006867000066650000646300006261000060 5f",
    MEM_3F              => x"00007e7d00007c7b00007a7900007877000076750000747300007271000070 6f",
    MEMP_00             => x"0000000000000000000000000000000000000000000000000000000000000010",
    MEMP_01             => x"0000000000000000000000000000000000000000000000000000000000000010",
    MEMP_02             => x"0000000000000000000000000000000000000000000000000000000000000010",
    MEMP_03             => x"0000000000000000000000000000000000000000000000000000000000000010",
    MEMP_04             => x"0000000000000000000000000000000000000000000000000000000000000010",
    MEMP_05             => x"0000000000000000000000000000000000000000000000000000000000000010",
    MEMP_06             => x"0000000000000000000000000000000000000000000000000000000000000010",
    MEMP_07             => x"0000000000000000000000000000000000000000000000000000000000000010"
)
port map
(
    -- User Interface
    TX_DATA(39 downto 16)       =>    tile0_txdata1_float_i,
    TX_DATA(15 downto 0)        =>    tile0_txdata1_i,

    TX_CHARISK(3 downto 2)      =>    tile0_txcharisk1_float_i,
    TX_CHARISK(1 downto 0)      =>    tile0_txcharisk1_i,
    -- System Interface
    USER_CLK                    =>    tile0_txusrclk0_i,
```

```
    SYSTEM_RESET                =>      tile0_tx_system_reset1_c
);

tile1_frame_gen0 : FRAME_GEN
generic map
(
    WORDS_IN_BRAM               =>      EXAMPLE_WORDS_IN_BRAM,
    MEM_00              =>  x"00000e0d00000c0b00000a0900000807000006050000040300000 2bc00000100",
    MEM_01              =>  x"00001e1d00001c1b00001a1900001817000016150000141300001 2110000100f",
    MEM_02              =>  x"00002e2d00002c2b00002a2900002827000026250000242300002 2210000201f",
    MEM_03              =>  x"00003e3d00003c3b00003a3900003837000036350000343300003 2310000302f",
    MEM_04              =>  x"00004e4d00004c4b00004a4900004847000046450000444300004 2410000403f",
    MEM_05              =>  x"00005e5d00005c5b00005a5900005857000056550000545300005 2510000504f",
    MEM_06              =>  x"00006e6d00006c6b00006a6900006867000066650000646300006 2610000605f",
    MEM_07              =>  x"00007e7d00007c7b00007a7900007877000076750000747300007 2710000706f",
    MEM_08              =>  x"00000e0d00000c0b00000a0900000807000006050000040300000 2bc00000100",
    MEM_09              =>  x"00001e1d00001c1b00001a1900001817000016150000141300001 2110000100f",
    MEM_0A              =>  x"00002e2d00002c2b00002a2900002827000026250000242300002 2210000201f",
    MEM_0B              =>  x"00003e3d00003c3b00003a3900003837000036350000343300003 2310000302f",
    MEM_0C              =>  x"00004e4d00004c4b00004a4900004847000046450000444300004 2410000403f",
    MEM_0D              =>  x"00005e5d00005c5b00005a5900005857000056550000545300005 2510000504f",
    MEM_0E              =>  x"00006e6d00006c6b00006a6900006867000066650000646300006 2610000605f",
    MEM_0F              =>  x"00007e7d00007c7b00007a7900007877000076750000747300007 2710000706f",
    MEM_10              =>  x"00000e0d00000c0b00000a0900000807000006050000040300000 2bc00000100",
    MEM_11              =>  x"00001e1d00001c1b00001a1900001817000016150000141300001 2110000100f",
    MEM_12              =>  x"00002e2d00002c2b00002a2900002827000026250000242300002 2210000201f",
    MEM_13              =>  x"00003e3d00003c3b00003a3900003837000036350000343300003 2310000302f",
    MEM_14              =>  x"00004e4d00004c4b00004a4900004847000046450000444300004 2410000403f",
    MEM_15              =>  x"00005e5d00005c5b00005a5900005857000056550000545300005 2510000504f",
    MEM_16              =>  x"00006e6d00006c6b00006a6900006867000066650000646300006 2610000605f",
    MEM_17              =>  x"00007e7d00007c7b00007a7900007877000076750000747300007 2710000706f",
    MEM_18              =>  x"00000e0d00000c0b00000a0900000807000006050000040300000 2bc00000100",
    MEM_19              =>  x"00001e1d00001c1b00001a1900001817000016150000141300001 2110000100f",
    MEM_1A              =>  x"00002e2d00002c2b00002a2900002827000026250000242300002 2210000201f",
    MEM_1B              =>  x"00003e3d00003c3b00003a3900003837000036350000343300003 2310000302f",
    MEM_1C              =>  x"00004e4d00004c4b00004a4900004847000046450000444300004 2410000403f",
    MEM_1D              =>  x"00005e5d00005c5b00005a5900005857000056550000545300005 2510000504f",
    MEM_1E              =>  x"00006e6d00006c6b00006a6900006867000066650000646300006 2610000605f",
    MEM_1F              =>  x"00007e7d00007c7b00007a7900007877000076750000747300007 2710000706f",
    MEM_20              =>  x"00000e0d00000c0b00000a0900000807000006050000040300000 2bc00000100",
    MEM_21              =>  x"00001e1d00001c1b00001a1900001817000016150000141300001 2110000100f",
    MEM_22              =>  x"00002e2d00002c2b00002a2900002827000026250000242300002 2210000201f",
    MEM_23              =>  x"00003e3d00003c3b00003a3900003837000036350000343300003 2310000302f",
    MEM_24              =>  x"00004e4d00004c4b00004a4900004847000046450000444300004 2410000403f",
    MEM_25              =>  x"00005e5d00005c5b00005a5900005857000056550000545300005 2510000504f",
    MEM_26              =>  x"00006e6d00006c6b00006a6900006867000066650000646300006 2610000605f",
    MEM_27              =>  x"00007e7d00007c7b00007a7900007877000076750000747300007 2710000706f",
    MEM_28              =>  x"00000e0d00000c0b00000a0900000807000006050000040300000 2bc00000100",
    MEM_29              =>  x"00001e1d00001c1b00001a1900001817000016150000141300001 2110000100f",
    MEM_2A              =>  x"00002e2d00002c2b00002a2900002827000026250000242300002 2210000201f",
    MEM_2B              =>  x"00003e3d00003c3b00003a3900003837000036350000343300003 2310000302f",
    MEM_2C              =>  x"00004e4d00004c4b00004a4900004847000046450000444300004 2410000403f",
    MEM_2D              =>  x"00005e5d00005c5b00005a5900005857000056550000545300005 2510000504f",
    MEM_2E              =>  x"00006e6d00006c6b00006a6900006867000066650000646300006 2610000605f",
    MEM_2F              =>  x"00007e7d00007c7b00007a7900007877000076750000747300007 2710000706f",
    MEM_30              =>  x"00000e0d00000c0b00000a0900000807000006050000040300000 2bc00000100",
    MEM_31              =>  x"00001e1d00001c1b00001a1900001817000016150000141300001 2110000100f",
    MEM_32              =>  x"00002e2d00002c2b00002a2900002827000026250000242300002 2210000201f",
    MEM_33              =>  x"00003e3d00003c3b00003a3900003837000036350000343300003 2310000302f",
    MEM_34              =>  x"00004e4d00004c4b00004a4900004847000046450000444300004 2410000403f",
    MEM_35              =>  x"00005e5d00005c5b00005a5900005857000056550000545300005 2510000504f",
    MEM_36              =>  x"00006e6d00006c6b00006a6900006867000066650000646300006 2610000605f",
    MEM_37              =>  x"00007e7d00007c7b00007a7900007877000076750000747300007 2710000706f",
    MEM_38              =>  x"00000e0d00000c0b00000a0900000807000006050000040300000 2bc00000100",
    MEM_39              =>  x"00001e1d00001c1b00001a1900001817000016150000141300001 2110000100f",
    MEM_3A              =>  x"00002e2d00002c2b00002a2900002827000026250000242300002 2210000201f",
    MEM_3B              =>  x"00003e3d00003c3b00003a3900003837000036350000343300003 2310000302f",
    MEM_3C              =>  x"00004e4d00004c4b00004a4900004847000046450000444300004 2410000403f",
    MEM_3D              =>  x"00005e5d00005c5b00005a5900005857000056550000545300005 2510000504f",
    MEM_3E              =>  x"00006e6d00006c6b00006a6900006867000066650000646300006 2610000605f",
```

```
    MEM_3F              => x"00007e7d00007c7b00007a7900007877000076750000747300007271000706f",
    MEMP_00             => x"0000000000000000000000000000000000000000000000000000000000000010",
    MEMP_01             => x"0000000000000000000000000000000000000000000000000000000000000010",
    MEMP_02             => x"0000000000000000000000000000000000000000000000000000000000000010",
    MEMP_03             => x"0000000000000000000000000000000000000000000000000000000000000010",
    MEMP_04             => x"0000000000000000000000000000000000000000000000000000000000000010",
    MEMP_05             => x"0000000000000000000000000000000000000000000000000000000000000010",
    MEMP_06             => x"0000000000000000000000000000000000000000000000000000000000000010",
    MEMP_07             => x"0000000000000000000000000000000000000000000000000000000000000010"
)
port map
(
    -- User Interface
    TX_DATA(39 downto 16)       =>    tile1_txdata0_float_i,
    TX_DATA(15 downto 0)        =>    tile1_txdata0_i,

    TX_CHARISK(3 downto 2)      =>    tile1_txcharisk0_float_i,
    TX_CHARISK(1 downto 0)      =>    tile1_txcharisk0_i,
    -- System Interface
    USER_CLK                =>    tile1_txusrclk0_i,
    SYSTEM_RESET            =>    tile1_tx_system_reset0_c
);

tile1_frame_gen1 : FRAME_GEN
generic map
(
    WORDS_IN_BRAM           =>    EXAMPLE_WORDS_IN_BRAM,
    MEM_00              => x"00000e0d00000c0b00000a0900000807000006050000040300000 2bc00000100",
    MEM_01              => x"00001e1d00001c1b00001a1900001817000016150000141300001 2110000100f",
    MEM_02              => x"00002e2d00002c2b00002a2900002827000026250000242300002 2210000201f",
    MEM_03              => x"00003e3d00003c3b00003a3900003837000036350000343300003 2310000302f",
    MEM_04              => x"00004e4d00004c4b00004a4900004847000046450000444300004 2410000403f",
    MEM_05              => x"00005e5d00005c5b00005a5900005857000056550000545300005 2510000504f",
    MEM_06              => x"00006e6d00006c6b00006a6900006867000066650000646300006 2610000605f",
    MEM_07              => x"00007e7d00007c7b00007a7900007877000076750000747300007 2710000706f",
    MEM_08              => x"00000e0d00000c0b00000a0900000807000006050000040300000 2bc00000100",
    MEM_09              => x"00001e1d00001c1b00001a1900001817000016150000141300001 2110000100f",
    MEM_0A              => x"00002e2d00002c2b00002a2900002827000026250000242300002 2210000201f",
    MEM_0B              => x"00003e3d00003c3b00003a3900003837000036350000343300003 2310000302f",
    MEM_0C              => x"00004e4d00004c4b00004a4900004847000046450000444300004 2410000403f",
    MEM_0D              => x"00005e5d00005c5b00005a5900005857000056550000545300005 2510000504f",
    MEM_0E              => x"00006e6d00006c6b00006a6900006867000066650000646300006 2610000605f",
    MEM_0F              => x"00007e7d00007c7b00007a7900007877000076750000747300007 2710000706f",
    MEM_10              => x"00000e0d00000c0b00000a0900000807000006050000040300000 2bc00000100",
    MEM_11              => x"00001e1d00001c1b00001a1900001817000016150000141300001 2110000100f",
    MEM_12              => x"00002e2d00002c2b00002a2900002827000026250000242300002 2210000201f",
    MEM_13              => x"00003e3d00003c3b00003a3900003837000036350000343300003 2310000302f",
    MEM_14              => x"00004e4d00004c4b00004a4900004847000046450000444300004 2410000403f",
    MEM_15              => x"00005e5d00005c5b00005a5900005857000056550000545300005 2510000504f",
    MEM_16              => x"00006e6d00006c6b00006a6900006867000066650000646300006 2610000605f",
    MEM_17              => x"00007e7d00007c7b00007a7900007877000076750000747300007 2710000706f",
    MEM_18              => x"00000e0d00000c0b00000a0900000807000006050000040300000 2bc00000100",
    MEM_19              => x"00001e1d00001c1b00001a1900001817000016150000141300001 2110000100f",
    MEM_1A              => x"00002e2d00002c2b00002a2900002827000026250000242300002 2210000201f",
    MEM_1B              => x"00003e3d00003c3b00003a3900003837000036350000343300003 2310000302f",
    MEM_1C              => x"00004e4d00004c4b00004a4900004847000046450000444300004 2410000403f",
    MEM_1D              => x"00005e5d00005c5b00005a5900005857000056550000545300005 2510000504f",
    MEM_1E              => x"00006e6d00006c6b00006a6900006867000066650000646300006 2610000605f",
    MEM_1F              => x"00007e7d00007c7b00007a7900007877000076750000747300007 2710000706f",
    MEM_20              => x"00000e0d00000c0b00000a0900000807000006050000040300000 2bc00000100",
    MEM_21              => x"00001e1d00001c1b00001a1900001817000016150000141300001 2110000100f",
    MEM_22              => x"00002e2d00002c2b00002a2900002827000026250000242300002 2210000201f",
    MEM_23              => x"00003e3d00003c3b00003a3900003837000036350000343300003 2310000302f",
    MEM_24              => x"00004e4d00004c4b00004a4900004847000046450000444300004 2410000403f",
    MEM_25              => x"00005e5d00005c5b00005a5900005857000056550000545300005 2510000504f",
    MEM_26              => x"00006e6d00006c6b00006a6900006867000066650000646300006 2610000605f",
    MEM_27              => x"00007e7d00007c7b00007a7900007877000076750000747300007 2710000706f",
    MEM_28              => x"00000e0d00000c0b00000a0900000807000006050000040300000 2bc00000100",
    MEM_29              => x"00001e1d00001c1b00001a1900001817000016150000141300001 2110000100f",
    MEM_2A              => x"00002e2d00002c2b00002a2900002827000026250000242300002 2210000201f",
```

```
    MEM_2B              =>  x"00003e3d00003c3b00003a3900003837000036350000343300003231000302f",
    MEM_2C              =>  x"00004e4d00004c4b00004a4900004847000046450000444300004241000403f",
    MEM_2D              =>  x"00005e5d00005c5b00005a5900005857000056550000545300005251000504f",
    MEM_2E              =>  x"00006e6d00006c6b00006a6900006867000066650000646300006261000605f",
    MEM_2F              =>  x"00007e7d00007c7b00007a7900007877000076750000747300007271000706f",
    MEM_30              =>  x"00000e0d00000c0b00000a0900000807000006050000040300000 2bc00000100",
    MEM_31              =>  x"00001e1d00001c1b00001a1900001817000016150000141300001211000 0100f",
    MEM_32              =>  x"00002e2d00002c2b00002a2900002827000026250000242300002221000 0201f",
    MEM_33              =>  x"00003e3d00003c3b00003a3900003837000036350000343300003231000 3302f",
    MEM_34              =>  x"00004e4d00004c4b00004a4900004847000046450000444300004241000 4403f",
    MEM_35              =>  x"00005e5d00005c5b00005a5900005857000056550000545300005251000 5504f",
    MEM_36              =>  x"00006e6d00006c6b00006a6900006867000066650000646300006261000 6605f",
    MEM_37              =>  x"00007e7d00007c7b00007a7900007877000076750000747300007271000 7706f",
    MEM_38              =>  x"00000e0d00000c0b00000a0900000807000006050000040300000 2bc00000100",
    MEM_39              =>  x"00001e1d00001c1b00001a1900001817000016150000141300001211000 0100f",
    MEM_3A              =>  x"00002e2d00002c2b00002a2900002827000026250000242300002221000 0201f",
    MEM_3B              =>  x"00003e3d00003c3b00003a3900003837000036350000343300003231000 3302f",
    MEM_3C              =>  x"00004e4d00004c4b00004a4900004847000046450000444300004241000 4403f",
    MEM_3D              =>  x"00005e5d00005c5b00005a5900005857000056550000545300005251000 5504f",
    MEM_3E              =>  x"00006e6d00006c6b00006a6900006867000066650000646300006261000 6605f",
    MEM_3F              =>  x"00007e7d00007c7b00007a7900007877000076750000747300007271000 7706f",
    MEMP_00             =>  x"00000000000000000000000000000000000000000000000000000000000 00010",
    MEMP_01             =>  x"00000000000000000000000000000000000000000000000000000000000 00010",
    MEMP_02             =>  x"00000000000000000000000000000000000000000000000000000000000 00010",
    MEMP_03             =>  x"00000000000000000000000000000000000000000000000000000000000 00010",
    MEMP_04             =>  x"00000000000000000000000000000000000000000000000000000000000 00010",
    MEMP_05             =>  x"00000000000000000000000000000000000000000000000000000000000 00010",
    MEMP_06             =>  x"00000000000000000000000000000000000000000000000000000000000 00010",
    MEMP_07             =>  x"00000000000000000000000000000000000000000000000000000000000 00010"
  )
  port map
  (
    -- User Interface
    TX_DATA(39 downto 16)       =>    tile1_txdata1_float_i,
    TX_DATA(15 downto 0)        =>    tile1_txdata1_i,

    TX_CHARISK(3 downto 2)      =>    tile1_txcharisk1_float_i,
    TX_CHARISK(1 downto 0)      =>    tile1_txcharisk1_i,
    -- System Interface
    USER_CLK                    =>    tile1_txusrclk0_i,
    SYSTEM_RESET                =>    tile1_tx_system_reset1_c
  );
  ---------------------------------- Frame Checkers ------------------------
  -- The example design uses Block RAM based frame checkers to verify incoming
  -- data. By default the frame generators are loaded with a data sequence that
  -- matches the outgoing sequence of the frame generators for the TX ports.

  -- You can modify the expected data sequence by changing the INIT values of the frame
  -- checkers in this file. Pay careful attention to bit order and the spacing
  -- of your control and alignment characters.

  -- When the frame checker receives data, it attempts to synchronise to the
  -- incoming pattern by looking for the first sequence in the pattern. Once it
  -- finds the first sequence, it increments through the sequence, and indicates an
  -- error whenever the next value received does not match the expected value.
  tile0_frame_check0_reset_i          <= reset_on_data_error_i when (EXAMPLE_CONFIG_INDEPENDENT_LANES=0) else
tile0_matchn0_i;
  -- tile0_frame_check0 is always connected to the lane with the start of char
  -- and this lane starts off the data checking on all the other lanes. The INC_IN port is tied off
  tile0_inc_in0_i                     <= '0';

  tile0_frame_check0 : FRAME_CHECK
  generic map
  (
    RX_DATA_WIDTH               =>    16,
    USE_COMMA                   =>    1,
    WORDS_IN_BRAM               =>    EXAMPLE_WORDS_IN_BRAM,
    CONFIG_INDEPENDENT_LANES    =>    1,
    START_OF_PACKET_CHAR        =>    x"bc",
    MEM_00              =>  x"00000e0d00000c0b00000a0900000807000006050000040300000 2bc00000100",
```

```
MEM_01    => x"00001e1d00001c1b00001a1900001817000016150000141300001211000010f",
MEM_02    => x"00002e2d00002c2b00002a2900002827000026250000242300002221000020f",
MEM_03    => x"00003e3d00003c3b00003a3900003837000036350000343300003231000030f",
MEM_04    => x"00004e4d00004c4b00004a4900004847000046450000444300004241000040f",
MEM_05    => x"00005e5d00005c5b00005a5900005857000056550000545300005251000050f",
MEM_06    => x"00006e6d00006c6b00006a6900006867000066650000646300006261000060f",
MEM_07    => x"00007e7d00007c7b00007a7900007877000076750000747300007271000070f",
MEM_08    => x"00000e0d00000c0b00000a0900000807000006050000040300000 2bc00000100",
MEM_09    => x"00001e1d00001c1b00001a1900001817000016150000141300001211000010f",
MEM_0A    => x"00002e2d00002c2b00002a2900002827000026250000242300002221000020f",
MEM_0B    => x"00003e3d00003c3b00003a3900003837000036350000343300003231000030f",
MEM_0C    => x"00004e4d00004c4b00004a4900004847000046450000444300004241000040f",
MEM_0D    => x"00005e5d00005c5b00005a5900005857000056550000545300005251000050f",
MEM_0E    => x"00006e6d00006c6b00006a6900006867000066650000646300006261000060f",
MEM_0F    => x"00007e7d00007c7b00007a7900007877000076750000747300007271000070f",
MEM_10    => x"00000e0d00000c0b00000a0900000807000006050000040300000 2bc00000100",
MEM_11    => x"00001e1d00001c1b00001a1900001817000016150000141300001211000010f",
MEM_12    => x"00002e2d00002c2b00002a2900002827000026250000242300002221000020f",
MEM_13    => x"00003e3d00003c3b00003a3900003837000036350000343300003231000030f",
MEM_14    => x"00004e4d00004c4b00004a4900004847000046450000444300004241000040f",
MEM_15    => x"00005e5d00005c5b00005a5900005857000056550000545300005251000050f",
MEM_16    => x"00006e6d00006c6b00006a6900006867000066650000646300006261000060f",
MEM_17    => x"00007e7d00007c7b00007a7900007877000076750000747300007271000070f",
MEM_18    => x"00000e0d00000c0b00000a0900000807000006050000040300000 2bc00000100",
MEM_19    => x"00001e1d00001c1b00001a1900001817000016150000141300001211000010f",
MEM_1A    => x"00002e2d00002c2b00002a2900002827000026250000242300002221000020f",
MEM_1B    => x"00003e3d00003c3b00003a3900003837000036350000343300003231000030f",
MEM_1C    => x"00004e4d00004c4b00004a4900004847000046450000444300004241000040f",
MEM_1D    => x"00005e5d00005c5b00005a5900005857000056550000545300005251000050f",
MEM_1E    => x"00006e6d00006c6b00006a6900006867000066650000646300006261000060f",
MEM_1F    => x"00007e7d00007c7b00007a7900007877000076750000747300007271000070f",
MEM_20    => x"00000e0d00000c0b00000a0900000807000006050000040300000 2bc00000100",
MEM_21    => x"00001e1d00001c1b00001a1900001817000016150000141300001211000010f",
MEM_22    => x"00002e2d00002c2b00002a2900002827000026250000242300002221000020f",
MEM_23    => x"00003e3d00003c3b00003a3900003837000036350000343300003231000030f",
MEM_24    => x"00004e4d00004c4b00004a4900004847000046450000444300004241000040f",
MEM_25    => x"00005e5d00005c5b00005a5900005857000056550000545300005251000050f",
MEM_26    => x"00006e6d00006c6b00006a6900006867000066650000646300006261000060f",
MEM_27    => x"00007e7d00007c7b00007a7900007877000076750000747300007271000070f",
MEM_28    => x"00000e0d00000c0b00000a0900000807000006050000040300000 2bc00000100",
MEM_29    => x"00001e1d00001c1b00001a1900001817000016150000141300001211000010f",
MEM_2A    => x"00002e2d00002c2b00002a2900002827000026250000242300002221000020f",
MEM_2B    => x"00003e3d00003c3b00003a3900003837000036350000343300003231000030f",
MEM_2C    => x"00004e4d00004c4b00004a4900004847000046450000444300004241000040f",
MEM_2D    => x"00005e5d00005c5b00005a5900005857000056550000545300005251000050f",
MEM_2E    => x"00006e6d00006c6b00006a6900006867000066650000646300006261000060f",
MEM_2F    => x"00007e7d00007c7b00007a7900007877000076750000747300007271000070f",
MEM_30    => x"00000e0d00000c0b00000a0900000807000006050000040300000 2bc00000100",
MEM_31    => x"00001e1d00001c1b00001a1900001817000016150000141300001211000010f",
MEM_32    => x"00002e2d00002c2b00002a2900002827000026250000242300002221000020f",
MEM_33    => x"00003e3d00003c3b00003a3900003837000036350000343300003231000030f",
MEM_34    => x"00004e4d00004c4b00004a4900004847000046450000444300004241000040f",
MEM_35    => x"00005e5d00005c5b00005a5900005857000056550000545300005251000050f",
MEM_36    => x"00006e6d00006c6b00006a6900006867000066650000646300006261000060f",
MEM_37    => x"00007e7d00007c7b00007a7900007877000076750000747300007271000070f",
MEM_38    => x"00000e0d00000c0b00000a0900000807000006050000040300000 2bc00000100",
MEM_39    => x"00001e1d00001c1b00001a1900001817000016150000141300001211000010f",
MEM_3A    => x"00002e2d00002c2b00002a2900002827000026250000242300002221000020f",
MEM_3B    => x"00003e3d00003c3b00003a3900003837000036350000343300003231000030f",
MEM_3C    => x"00004e4d00004c4b00004a4900004847000046450000444300004241000040f",
MEM_3D    => x"00005e5d00005c5b00005a5900005857000056550000545300005251000050f",
MEM_3E    => x"00006e6d00006c6b00006a6900006867000066650000646300006261000060f",
MEM_3F    => x"00007e7d00007c7b00007a7900007877000076750000747300007271000070f",
MEMP_00   => x"0000000000000000000000000000000000000000000000000000000000000010",
MEMP_01   => x"0000000000000000000000000000000000000000000000000000000000000010",
MEMP_02   => x"0000000000000000000000000000000000000000000000000000000000000010",
MEMP_03   => x"0000000000000000000000000000000000000000000000000000000000000010",
MEMP_04   => x"0000000000000000000000000000000000000000000000000000000000000010",
MEMP_05   => x"0000000000000000000000000000000000000000000000000000000000000010",
MEMP_06   => x"0000000000000000000000000000000000000000000000000000000000000010",
```

```
    MEMP_07               => x"0000000000000000000000000000000000000000000000000000000000000010"
  )
  port map
  (
    -- MGT Interface
    RX_DATA                   =>    tile0_rxdata0_i,
    RX_ENMCOMMA_ALIGN         =>    tile0_rxenmcommaalign0_i,
    RX_ENPCOMMA_ALIGN         =>    tile0_rxenpcommaalign0_i,
    RX_ENCHAN_SYNC            =>    open,
    RX_CHANBOND_SEQ           =>    tied_to_ground_i,
    -- Control Interface
    INC_IN                    =>    tile0_inc_in0_i,
    INC_OUT                   =>    tile0_inc_out0_i,
    PATTERN_MATCH_N           =>    tile0_matchn0_i,
    RESET_ON_ERROR            =>    tile0_frame_check0_reset_i,
    -- System Interface
    USER_CLK                  =>    tile0_txusrclk0_i,
    SYSTEM_RESET              =>    tile0_rx_system_reset0_c,
    ERROR_COUNT               =>    tile0_error_count0_i
  );

  tile0_frame_check1_reset_i          <= reset_on_data_error_i when (EXAMPLE_CONFIG_INDEPENDENT_LANES=0) else
tile0_matchn1_i;

  -- tile0_frame_check0 is always connected to the lane with the start of char
  -- and this lane starts off the data checking on all the other lanes. The INC_IN port is tied off
  tile0_inc_in1_i                   <= '0';

  tile0_frame_check1 : FRAME_CHECK
  generic map
  (
    RX_DATA_WIDTH             =>    16,
    USE_COMMA                 =>    1,
    WORDS_IN_BRAM             =>    EXAMPLE_WORDS_IN_BRAM,
    CONFIG_INDEPENDENT_LANES      =>    1,
    START_OF_PACKET_CHAR          =>    x"bc",
    MEM_00            => x"00000e0d00000c0b00000a0900000807000006050000040300000 2bc00000100",
    MEM_01            => x"00001e1d00001c1b00001a1900001817000016150000141300001 2110000100f",
    MEM_02            => x"00002e2d00002c2b00002a2900002827000026250000242300002 2210000201f",
    MEM_03            => x"00003e3d00003c3b00003a3900003837000036350000343300003 2310000302f",
    MEM_04            => x"00004e4d00004c4b00004a4900004847000046450000444300004 2410000403f",
    MEM_05            => x"00005e5d00005c5b00005a5900005857000056550000545300005 2510000504f",
    MEM_06            => x"00006e6d00006c6b00006a6900006867000066650000646300006 2610000605f",
    MEM_07            => x"00007e7d00007c7b00007a7900007877000076750000747300007 2710000706f",
    MEM_08            => x"00000e0d00000c0b00000a0900000807000006050000040300000 2bc00000100",
    MEM_09            => x"00001e1d00001c1b00001a1900001817000016150000141300001 2110000100f",
    MEM_0A            => x"00002e2d00002c2b00002a2900002827000026250000242300002 2210000201f",
    MEM_0B            => x"00003e3d00003c3b00003a3900003837000036350000343300003 2310000302f",
    MEM_0C            => x"00004e4d00004c4b00004a4900004847000046450000444300004 2410000403f",
    MEM_0D            => x"00005e5d00005c5b00005a5900005857000056550000545300005 2510000504f",
    MEM_0E            => x"00006e6d00006c6b00006a6900006867000066650000646300006 2610000605f",
    MEM_0F            => x"00007e7d00007c7b00007a7900007877000076750000747300007 2710000706f",
    MEM_10            => x"00000e0d00000c0b00000a0900000807000006050000040300000 2bc00000100",
    MEM_11            => x"00001e1d00001c1b00001a1900001817000016150000141300001 2110000100f",
    MEM_12            => x"00002e2d00002c2b00002a2900002827000026250000242300002 2210000201f",
    MEM_13            => x"00003e3d00003c3b00003a3900003837000036350000343300003 2310000302f",
    MEM_14            => x"00004e4d00004c4b00004a4900004847000046450000444300004 2410000403f",
    MEM_15            => x"00005e5d00005c5b00005a5900005857000056550000545300005 2510000504f",
    MEM_16            => x"00006e6d00006c6b00006a6900006867000066650000646300006 2610000605f",
    MEM_17            => x"00007e7d00007c7b00007a7900007877000076750000747300007 2710000706f",
    MEM_18            => x"00000e0d00000c0b00000a0900000807000006050000040300000 2bc00000100",
    MEM_19            => x"00001e1d00001c1b00001a1900001817000016150000141300001 2110000100f",
    MEM_1A            => x"00002e2d00002c2b00002a2900002827000026250000242300002 2210000201f",
    MEM_1B            => x"00003e3d00003c3b00003a3900003837000036350000343300003 2310000302f",
    MEM_1C            => x"00004e4d00004c4b00004a4900004847000046450000444300004 2410000403f",
    MEM_1D            => x"00005e5d00005c5b00005a5900005857000056550000545300005 2510000504f",
    MEM_1E            => x"00006e6d00006c6b00006a6900006867000066650000646300006 2610000605f",
    MEM_1F            => x"00007e7d00007c7b00007a7900007877000076750000747300007 2710000706f",
    MEM_20            => x"00000e0d00000c0b00000a0900000807000006050000040300000 2bc00000100",
    MEM_21            => x"00001e1d00001c1b00001a1900001817000016150000141300001 2110000100f",
```

```
    MEM_22              => x"00002e2d00002c2b00002a2900002827000026250000242300002221000020 1f",
    MEM_23              => x"00003e3d00003c3b00003a3900003837000036350000343300003231000030 2f",
    MEM_24              => x"00004e4d00004c4b00004a4900004847000046450000444300004241000040 3f",
    MEM_25              => x"00005e5d00005c5b00005a5900005857000056550000545300005251000050 4f",
    MEM_26              => x"00006e6d00006c6b00006a6900006867000066650000646300006261000060 5f",
    MEM_27              => x"00007e7d00007c7b00007a7900007877000076750000747300007271000070 6f",
    MEM_28              => x"00000e0d00000c0b00000a0900000807000006050000040300000 2bc00000100",
    MEM_29              => x"00001e1d00001c1b00001a1900001817000016150000141300001211000010 0f",
    MEM_2A              => x"00002e2d00002c2b00002a2900002827000026250000242300002221000020 1f",
    MEM_2B              => x"00003e3d00003c3b00003a3900003837000036350000343300003231000030 2f",
    MEM_2C              => x"00004e4d00004c4b00004a4900004847000046450000444300004241000040 3f",
    MEM_2D              => x"00005e5d00005c5b00005a5900005857000056550000545300005251000050 4f",
    MEM_2E              => x"00006e6d00006c6b00006a6900006867000066650000646300006261000060 5f",
    MEM_2F              => x"00007e7d00007c7b00007a7900007877000076750000747300007271000070 6f",
    MEM_30              => x"00000e0d00000c0b00000a0900000807000006050000040300000 2bc00000100",
    MEM_31              => x"00001e1d00001c1b00001a1900001817000016150000141300001211000010 0f",
    MEM_32              => x"00002e2d00002c2b00002a2900002827000026250000242300002221000020 1f",
    MEM_33              => x"00003e3d00003c3b00003a3900003837000036350000343300003231000030 2f",
    MEM_34              => x"00004e4d00004c4b00004a4900004847000046450000444300004241000040 3f",
    MEM_35              => x"00005e5d00005c5b00005a5900005857000056550000545300005251000050 4f",
    MEM_36              => x"00006e6d00006c6b00006a6900006867000066650000646300006261000060 5f",
    MEM_37              => x"00007e7d00007c7b00007a7900007877000076750000747300007271000070 6f",
    MEM_38              => x"00000e0d00000c0b00000a0900000807000006050000040300000 2bc00000100",
    MEM_39              => x"00001e1d00001c1b00001a1900001817000016150000141300001211000010 0f",
    MEM_3A              => x"00002e2d00002c2b00002a2900002827000026250000242300002221000020 1f",
    MEM_3B              => x"00003e3d00003c3b00003a3900003837000036350000343300003231000030 2f",
    MEM_3C              => x"00004e4d00004c4b00004a4900004847000046450000444300004241000040 3f",
    MEM_3D              => x"00005e5d00005c5b00005a5900005857000056550000545300005251000050 4f",
    MEM_3E              => x"00006e6d00006c6b00006a6900006867000066650000646300006261000060 5f",
    MEM_3F              => x"00007e7d00007c7b00007a7900007877000076750000747300007271000070 6f",
    MEMP_00             => x"00000000000000000000000000000000000000000000000000000000000000 10",
    MEMP_01             => x"00000000000000000000000000000000000000000000000000000000000000 10",
    MEMP_02             => x"00000000000000000000000000000000000000000000000000000000000000 10",
    MEMP_03             => x"00000000000000000000000000000000000000000000000000000000000000 10",
    MEMP_04             => x"00000000000000000000000000000000000000000000000000000000000000 10",
    MEMP_05             => x"00000000000000000000000000000000000000000000000000000000000000 10",
    MEMP_06             => x"00000000000000000000000000000000000000000000000000000000000000 10",
    MEMP_07             => x"00000000000000000000000000000000000000000000000000000000000000 10"
  )
  port map
  (
    -- MGT Interface
    RX_DATA                     =>    tile0_rxdata1_i,
    RX_ENMCOMMA_ALIGN           =>    tile0_rxenmcommaalign1_i,
    RX_ENPCOMMA_ALIGN           =>    tile0_rxenpcommaalign1_i,
    RX_ENCHAN_SYNC              =>    open,
    RX_CHANBOND_SEQ             =>    tied_to_ground_i,
    -- Control Interface
    INC_IN                      =>    tile0_inc_in1_i,
    INC_OUT                     =>    tile0_inc_out1_i,
    PATTERN_MATCH_N             =>    tile0_matchn1_i,
    RESET_ON_ERROR              =>    tile0_frame_check1_reset_i,
    -- System Interface
    USER_CLK                    =>    tile0_txusrclk0_i,
    SYSTEM_RESET                =>    tile0_rx_system_reset1_c,
    ERROR_COUNT                 =>    tile0_error_count1_i
  );

  tile1_frame_check0_reset_i          <= reset_on_data_error_i when (EXAMPLE_CONFIG_INDEPENDENT_LANES=0) else
tile1_matchn0_i;

  -- in the "independent lanes" configuration, each of the lanes looks for the unique start char and
  -- in this case, the INC_IN port is tied off.
  -- Else, the data checking is triggered by the "master" lane
  tile1_inc_in0_i                   <= tile0_inc_out0_i when (EXAMPLE_CONFIG_INDEPENDENT_LANES=0) else '0';

  tile1_frame_check0 : FRAME_CHECK
  generic map
  (
    RX_DATA_WIDTH               =>    16,
```

```
USE_COMMA                 =>  1,
WORDS_IN_BRAM             =>  EXAMPLE_WORDS_IN_BRAM,
CONFIG_INDEPENDENT_LANES  =>  EXAMPLE_CONFIG_INDEPENDENT_LANES,
START_OF_PACKET_CHAR      =>  x"bc",
MEM_00          => x"00000e0d00000c0b00000a0900000807000006050000040300000 2bc00000100",
MEM_01          => x"00001e1d00001c1b00001a1900001817000016150000141300001 2110000100f",
MEM_02          => x"00002e2d00002c2b00002a2900002827000026250000242300002 2210000201f",
MEM_03          => x"00003e3d00003c3b00003a3900003837000036350000343300003 2310000302f",
MEM_04          => x"00004e4d00004c4b00004a4900004847000046450000444300004 2410000403f",
MEM_05          => x"00005e5d00005c5b00005a5900005857000056550000545300005 2510000504f",
MEM_06          => x"00006e6d00006c6b00006a6900006867000066650000646300006 2610000605f",
MEM_07          => x"00007e7d00007c7b00007a7900007877000076750000747300007 2710000706f",
MEM_08          => x"00000e0d00000c0b00000a0900000807000006050000040300000 2bc00000100",
MEM_09          => x"00001e1d00001c1b00001a1900001817000016150000141300001 2110000100f",
MEM_0A          => x"00002e2d00002c2b00002a2900002827000026250000242300002 2210000201f",
MEM_0B          => x"00003e3d00003c3b00003a3900003837000036350000343300003 2310000302f",
MEM_0C          => x"00004e4d00004c4b00004a4900004847000046450000444300004 2410000403f",
MEM_0D          => x"00005e5d00005c5b00005a5900005857000056550000545300005 2510000504f",
MEM_0E          => x"00006e6d00006c6b00006a6900006867000066650000646300006 2610000605f",
MEM_0F          => x"00007e7d00007c7b00007a7900007877000076750000747300007 2710000706f",
MEM_10          => x"00000e0d00000c0b00000a0900000807000006050000040300000 2bc00000100",
MEM_11          => x"00001e1d00001c1b00001a1900001817000016150000141300001 2110000100f",
MEM_12          => x"00002e2d00002c2b00002a2900002827000026250000242300002 2210000201f",
MEM_13          => x"00003e3d00003c3b00003a3900003837000036350000343300003 2310000302f",
MEM_14          => x"00004e4d00004c4b00004a4900004847000046450000444300004 2410000403f",
MEM_15          => x"00005e5d00005c5b00005a5900005857000056550000545300005 2510000504f",
MEM_16          => x"00006e6d00006c6b00006a6900006867000066650000646300006 2610000605f",
MEM_17          => x"00007e7d00007c7b00007a7900007877000076750000747300007 2710000706f",
MEM_18          => x"00000e0d00000c0b00000a0900000807000006050000040300000 2bc00000100",
MEM_19          => x"00001e1d00001c1b00001a1900001817000016150000141300001 2110000100f",
MEM_1A          => x"00002e2d00002c2b00002a2900002827000026250000242300002 2210000201f",
MEM_1B          => x"00003e3d00003c3b00003a3900003837000036350000343300003 2310000302f",
MEM_1C          => x"00004e4d00004c4b00004a4900004847000046450000444300004 2410000403f",
MEM_1D          => x"00005e5d00005c5b00005a5900005857000056550000545300005 2510000504f",
MEM_1E          => x"00006e6d00006c6b00006a6900006867000066650000646300006 2610000605f",
MEM_1F          => x"00007e7d00007c7b00007a7900007877000076750000747300007 2710000706f",
MEM_20          => x"00000e0d00000c0b00000a0900000807000006050000040300000 2bc00000100",
MEM_21          => x"00001e1d00001c1b00001a1900001817000016150000141300001 2110000100f",
MEM_22          => x"00002e2d00002c2b00002a2900002827000026250000242300002 2210000201f",
MEM_23          => x"00003e3d00003c3b00003a3900003837000036350000343300003 2310000302f",
MEM_24          => x"00004e4d00004c4b00004a4900004847000046450000444300004 2410000403f",
MEM_25          => x"00005e5d00005c5b00005a5900005857000056550000545300005 2510000504f",
MEM_26          => x"00006e6d00006c6b00006a6900006867000066650000646300006 2610000605f",
MEM_27          => x"00007e7d00007c7b00007a7900007877000076750000747300007 2710000706f",
MEM_28          => x"00000e0d00000c0b00000a0900000807000006050000040300000 2bc00000100",
MEM_29          => x"00001e1d00001c1b00001a1900001817000016150000141300001 2110000100f",
MEM_2A          => x"00002e2d00002c2b00002a2900002827000026250000242300002 2210000201f",
MEM_2B          => x"00003e3d00003c3b00003a3900003837000036350000343300003 2310000302f",
MEM_2C          => x"00004e4d00004c4b00004a4900004847000046450000444300004 2410000403f",
MEM_2D          => x"00005e5d00005c5b00005a5900005857000056550000545300005 2510000504f",
MEM_2E          => x"00006e6d00006c6b00006a6900006867000066650000646300006 2610000605f",
MEM_2F          => x"00007e7d00007c7b00007a7900007877000076750000747300007 2710000706f",
MEM_30          => x"00000e0d00000c0b00000a0900000807000006050000040300000 2bc00000100",
MEM_31          => x"00001e1d00001c1b00001a1900001817000016150000141300001 2110000100f",
MEM_32          => x"00002e2d00002c2b00002a2900002827000026250000242300002 2210000201f",
MEM_33          => x"00003e3d00003c3b00003a3900003837000036350000343300003 2310000302f",
MEM_34          => x"00004e4d00004c4b00004a4900004847000046450000444300004 2410000403f",
MEM_35          => x"00005e5d00005c5b00005a5900005857000056550000545300005 2510000504f",
MEM_36          => x"00006e6d00006c6b00006a6900006867000066650000646300006 2610000605f",
MEM_37          => x"00007e7d00007c7b00007a7900007877000076750000747300007 2710000706f",
MEM_38          => x"00000e0d00000c0b00000a0900000807000006050000040300000 2bc00000100",
MEM_39          => x"00001e1d00001c1b00001a1900001817000016150000141300001 2110000100f",
MEM_3A          => x"00002e2d00002c2b00002a2900002827000026250000242300002 2210000201f",
MEM_3B          => x"00003e3d00003c3b00003a3900003837000036350000343300003 2310000302f",
MEM_3C          => x"00004e4d00004c4b00004a4900004847000046450000444300004 2410000403f",
MEM_3D          => x"00005e5d00005c5b00005a5900005857000056550000545300005 2510000504f",
MEM_3E          => x"00006e6d00006c6b00006a6900006867000066650000646300006 2610000605f",
MEM_3F          => x"00007e7d00007c7b00007a7900007877000076750000747300007 2710000706f",
MEMP_00           => x"0000000000000000000000000000000000000000000000000000000000000010",
MEMP_01           => x"0000000000000000000000000000000000000000000000000000000000000010",
```

```
    MEMP_02             => x"0000000000000000000000000000000000000000000000000000000000000010",
    MEMP_03             => x"0000000000000000000000000000000000000000000000000000000000000010",
    MEMP_04             => x"0000000000000000000000000000000000000000000000000000000000000010",
    MEMP_05             => x"0000000000000000000000000000000000000000000000000000000000000010",
    MEMP_06             => x"0000000000000000000000000000000000000000000000000000000000000010",
    MEMP_07             => x"0000000000000000000000000000000000000000000000000000000000000010"
  )
  port map
  (
    -- MGT Interface
    RX_DATA                 =>  tile1_rxdata0_i,
    RX_ENMCOMMA_ALIGN       =>  tile1_rxenmcommaalign0_i,
    RX_ENPCOMMA_ALIGN       =>  tile1_rxenpcommaalign0_i,
    RX_ENCHAN_SYNC          =>  open,
    RX_CHANBOND_SEQ         =>  tied_to_ground_i,
    -- Control Interface
    INC_IN                  =>  tile1_inc_in0_i,
    INC_OUT                 =>  tile1_inc_out0_i,
    PATTERN_MATCH_N         =>  tile1_matchn0_i,
    RESET_ON_ERROR          =>  tile1_frame_check0_reset_i,
    -- System Interface
    USER_CLK                =>  tile1_txusrclk0_i,
    SYSTEM_RESET            =>  tile1_rx_system_reset0_c,
    ERROR_COUNT             =>  tile1_error_count0_i
  );

  tile1_frame_check1_reset_i        <= reset_on_data_error_i when (EXAMPLE_CONFIG_INDEPENDENT_LANES=0) else
tile1_matchn1_i;

  -- in the "independent lanes" configuration, each of the lanes looks for the unique start char and
  -- in this case, the INC_IN port is tied off.
  -- Else, the data checking is triggered by the "master" lane
  tile1_inc_in1_i             <= tile0_inc_out1_i when (EXAMPLE_CONFIG_INDEPENDENT_LANES=0) else '0';

  tile1_frame_check1 : FRAME_CHECK
  generic map
  (
    RX_DATA_WIDTH           =>  16,
    USE_COMMA               =>  1,
    WORDS_IN_BRAM           =>  EXAMPLE_WORDS_IN_BRAM,
    CONFIG_INDEPENDENT_LANES    =>  EXAMPLE_CONFIG_INDEPENDENT_LANES,
    START_OF_PACKET_CHAR        =>  x"bc",
    MEM_00          => x"00000e0d00000c0b00000a0900000807000006050000040300000 2bc00000100",
    MEM_01          => x"00001e1d00001c1b00001a1900001817000016150000141300001 2110000100f",
    MEM_02          => x"00002e2d00002c2b00002a2900002827000026250000242300002 2210000201f",
    MEM_03          => x"00003e3d00003c3b00003a3900003837000036350000343300003 2310000302f",
    MEM_04          => x"00004e4d00004c4b00004a4900004847000046450000444300004 2410000403f",
    MEM_05          => x"00005e5d00005c5b00005a5900005857000056550000545300005 2510000504f",
    MEM_06          => x"00006e6d00006c6b00006a6900006867000066650000646300006 2610000605f",
    MEM_07          => x"00007e7d00007c7b00007a7900007877000076750000747300007 2710000706f",
    MEM_08          => x"00000e0d00000c0b00000a0900000807000006050000040300000 2bc00000100",
    MEM_09          => x"00001e1d00001c1b00001a1900001817000016150000141300001 2110000100f",
    MEM_0A          => x"00002e2d00002c2b00002a2900002827000026250000242300002 2210000201f",
    MEM_0B          => x"00003e3d00003c3b00003a3900003837000036350000343300003 2310000302f",
    MEM_0C          => x"00004e4d00004c4b00004a4900004847000046450000444300004 2410000403f",
    MEM_0D          => x"00005e5d00005c5b00005a5900005857000056550000545300005 2510000504f",
    MEM_0E          => x"00006e6d00006c6b00006a6900006867000066650000646300006 2610000605f",
    MEM_0F          => x"00007e7d00007c7b00007a7900007877000076750000747300007 2710000706f",
    MEM_10          => x"00000e0d00000c0b00000a0900000807000006050000040300000 2bc00000100",
    MEM_11          => x"00001e1d00001c1b00001a1900001817000016150000141300001 2110000100f",
    MEM_12          => x"00002e2d00002c2b00002a2900002827000026250000242300002 2210000201f",
    MEM_13          => x"00003e3d00003c3b00003a3900003837000036350000343300003 2310000302f",
    MEM_14          => x"00004e4d00004c4b00004a4900004847000046450000444300004 2410000403f",
    MEM_15          => x"00005e5d00005c5b00005a5900005857000056550000545300005 2510000504f",
    MEM_16          => x"00006e6d00006c6b00006a6900006867000066650000646300006 2610000605f",
    MEM_17          => x"00007e7d00007c7b00007a7900007877000076750000747300007 2710000706f",
    MEM_18          => x"00000e0d00000c0b00000a0900000807000006050000040300000 2bc00000100",
    MEM_19          => x"00001e1d00001c1b00001a1900001817000016150000141300001 2110000100f",
    MEM_1A          => x"00002e2d00002c2b00002a2900002827000026250000242300002 2210000201f",
    MEM_1B          => x"00003e3d00003c3b00003a3900003837000036350000343300003 2310000302f",
```

```
      MEM_1C               => x"00004e4d00004c4b00004a490000484700004645000044430000424 10000403f",
      MEM_1D               => x"00005e5d00005c5b00005a590000585700005655000054530000525 10000504f",
      MEM_1E               => x"00006e6d00006c6b00006a690000686700006665000064630000626 10000605f",
      MEM_1F               => x"00007e7d00007c7b00007a790000787700007675000074730000727 10000706f",
      MEM_20               => x"00000e0d00000c0b00000a090000080700000605000004030000002bc00000100",
      MEM_21               => x"00001e1d00001c1b00001a190000181700001615000014130000121 10000100f",
      MEM_22               => x"00002e2d00002c2b00002a290000282700002625000024230000222 10000201f",
      MEM_23               => x"00003e3d00003c3b00003a390000383700003635000034330000323 10000302f",
      MEM_24               => x"00004e4d00004c4b00004a490000484700004645000044430000424 10000403f",
      MEM_25               => x"00005e5d00005c5b00005a590000585700005655000054530000525 10000504f",
      MEM_26               => x"00006e6d00006c6b00006a690000686700006665000064630000626 10000605f",
      MEM_27               => x"00007e7d00007c7b00007a790000787700007675000074730000727 10000706f",
      MEM_28               => x"00000e0d00000c0b00000a090000080700000605000004030000002bc00000100",
      MEM_29               => x"00001e1d00001c1b00001a190000181700001615000014130000121 10000100f",
      MEM_2A               => x"00002e2d00002c2b00002a290000282700002625000024230000222 10000201f",
      MEM_2B               => x"00003e3d00003c3b00003a390000383700003635000034330000323 10000302f",
      MEM_2C               => x"00004e4d00004c4b00004a490000484700004645000044430000424 10000403f",
      MEM_2D               => x"00005e5d00005c5b00005a590000585700005655000054530000525 10000504f",
      MEM_2E               => x"00006e6d00006c6b00006a690000686700006665000064630000626 10000605f",
      MEM_2F               => x"00007e7d00007c7b00007a790000787700007675000074730000727 10000706f",
      MEM_30               => x"00000e0d00000c0b00000a090000080700000605000004030000002bc00000100",
      MEM_31               => x"00001e1d00001c1b00001a190000181700001615000014130000121 10000100f",
      MEM_32               => x"00002e2d00002c2b00002a290000282700002625000024230000222 10000201f",
      MEM_33               => x"00003e3d00003c3b00003a390000383700003635000034330000323 10000302f",
      MEM_34               => x"00004e4d00004c4b00004a490000484700004645000044430000424 10000403f",
      MEM_35               => x"00005e5d00005c5b00005a590000585700005655000054530000525 10000504f",
      MEM_36               => x"00006e6d00006c6b00006a690000686700006665000064630000626 10000605f",
      MEM_37               => x"00007e7d00007c7b00007a790000787700007675000074730000727 10000706f",
      MEM_38               => x"00000e0d00000c0b00000a090000080700000605000004030000002bc00000100",
      MEM_39               => x"00001e1d00001c1b00001a190000181700001615000014130000121 10000100f",
      MEM_3A               => x"00002e2d00002c2b00002a290000282700002625000024230000222 10000201f",
      MEM_3B               => x"00003e3d00003c3b00003a390000383700003635000034330000323 10000302f",
      MEM_3C               => x"00004e4d00004c4b00004a490000484700004645000044430000424 10000403f",
      MEM_3D               => x"00005e5d00005c5b00005a590000585700005655000054530000525 10000504f",
      MEM_3E               => x"00006e6d00006c6b00006a690000686700006665000064630000626 10000605f",
      MEM_3F               => x"00007e7d00007c7b00007a790000787700007675000074730000727 10000706f",
      MEMP_00              => x"0000000000000000000000000000000000000000000000000000000000000010",
      MEMP_01              => x"0000000000000000000000000000000000000000000000000000000000000010",
      MEMP_02              => x"0000000000000000000000000000000000000000000000000000000000000010",
      MEMP_03              => x"0000000000000000000000000000000000000000000000000000000000000010",
      MEMP_04              => x"0000000000000000000000000000000000000000000000000000000000000010",
      MEMP_05              => x"0000000000000000000000000000000000000000000000000000000000000010",
      MEMP_06              => x"0000000000000000000000000000000000000000000000000000000000000010",
      MEMP_07              => x"0000000000000000000000000000000000000000000000000000000000000010"
   )
   port map
   (
      -- MGT Interface
      RX_DATA                 =>    tile1_rxdata1_i,
      RX_ENMCOMMA_ALIGN       =>    tile1_rxenmcommaalign1_i,
      RX_ENPCOMMA_ALIGN       =>    tile1_rxenpcommaalign1_i,
      RX_ENCHAN_SYNC          =>    open,
      RX_CHANBOND_SEQ         =>    tied_to_ground_i,
      -- Control Interface
      INC_IN                  =>    tile1_inc_in1_i,
      INC_OUT                 =>    tile1_inc_out1_i,
      PATTERN_MATCH_N         =>    tile1_matchn1_i,
      RESET_ON_ERROR          =>    tile1_frame_check1_reset_i,
      -- System Interface
      USER_CLK                =>    tile1_txusrclk0_i,
      SYSTEM_RESET            =>    tile1_rx_system_reset1_c,
      ERROR_COUNT             =>    tile1_error_count1_i
   );
no_chipscope : if EXAMPLE_USE_CHIPSCOPE = 0 generate

   -- If Chipscope is not being used, drive GTX reset signal
   -- from the top level ports
   tile0_gtxreset_i           <= GTXRESET_IN;
   tile1_gtxreset_i           <= GTXRESET_IN;
```

```
    -- assign resets for frame_gen modules
    tile0_tx_system_reset0_c            <= not tile0_tx_resetdone0_r2;
    tile0_tx_system_reset1_c            <= not tile0_tx_resetdone1_r2;
    tile1_tx_system_reset0_c            <= not tile1_tx_resetdone0_r2;
    tile1_tx_system_reset1_c            <= not tile1_tx_resetdone1_r2;
    -- assign resets for frame_check modules
    tile0_rx_system_reset0_c            <= not tile0_rx_resetdone0_r2;
    tile0_rx_system_reset1_c            <= not tile0_rx_resetdone1_r2;
    tile1_rx_system_reset0_c            <= not tile1_rx_resetdone0_r2;
    tile1_rx_system_reset1_c            <= not tile1_rx_resetdone1_r2;
    tile0_loopback0_i                 <= tied_to_ground_vec_i(2 downto 0);
    tile0_txdiffctrl0_i               <= tied_to_ground_vec_i(2 downto 0);
    tile0_txpreemphasis0_i              <= tied_to_ground_vec_i(2 downto 0);
    tile0_txenprbstst0_i              <= tied_to_vcc_vec_i(1 downto 0);
    tile0_rxeqmix0_i                  <= tied_to_ground_vec_i(1 downto 0);
    tile0_loopback1_i                 <= tied_to_ground_vec_i(2 downto 0);
    tile0_txdiffctrl1_i               <= tied_to_ground_vec_i(2 downto 0);
    tile0_txpreemphasis1_i              <= tied_to_ground_vec_i(2 downto 0);
    tile0_txenprbstst1_i              <= tied_to_vcc_vec_i(1 downto 0);
    tile0_rxeqmix1_i                  <= tied_to_ground_vec_i(1 downto 0);
    tile1_loopback0_i                 <= tied_to_ground_vec_i(2 downto 0);
    tile1_txdiffctrl0_i               <= tied_to_ground_vec_i(2 downto 0);
    tile1_txpreemphasis0_i              <= tied_to_ground_vec_i(2 downto 0);
    tile1_txenprbstst0_i              <= tied_to_vcc_vec_i(1 downto 0);
    tile1_rxeqmix0_i                  <= tied_to_ground_vec_i(1 downto 0);
    tile1_loopback1_i                 <= tied_to_ground_vec_i(2 downto 0);
    tile1_txdiffctrl1_i               <= tied_to_ground_vec_i(2 downto 0);
    tile1_txpreemphasis1_i              <= tied_to_ground_vec_i(2 downto 0);
    tile1_txenprbstst1_i              <= tied_to_vcc_vec_i(1 downto 0);
    tile1_rxeqmix1_i                  <= tied_to_ground_vec_i(1 downto 0);
end generate no_chipscope;

end RTL;
```

APPENDIX B

Communication Firmware and Software

In this appendix details of the firmware and software used to communicate with the test module are presented. As discussed in Chapter 6, low-level communication takes place between the microcontroller and the PC USB drivers. The microcontroller must be programmed to accept communications from the USB driver and to also handshake with the drivers. These functions are programmed in assembly language and downloaded to the microcontroller. The code consists of six files mainly based on source code provided in [123]. The main file is named "hs.a51", which lists all the other files to use. The second file is "Declare.a51" which declares all the variables used in the project. The third file is "EZint.a51" which interprets interrupts from the USB drivers. The fourth file is "EZMain.a51". This is the main body of the project. This code controls how data is interpreted from the USB and written to the FPGA and vice versa. The fifth file is "Decode.a51", which handles how to decode USB packets. The final file is "DTables.a51", which declares descriptors required in the project. All these files are compiled together using an available compiler for microcontrollers and embedded processors called Keil µVision. The compiler produces one compiled hexadecimal file, which is downloaded to the microcontroller.

High level communication to the test module is done using a software interface, also discussed in Chapter 6. This software interface is programmed in C#. The software interface for the test module is based off much of the work done by Carl Grey at the High

Speed Digital Design Lab at Georgia Tech. The original interface is shown in Figure B.1, and its code provided in the file titled DLC_interface.cs. This interface essentially communicated over the USB to the microcontroller after it was loaded with firmware. This interface was modified to develop the test module interface shown in Figure B.2. The code for this interface is included as file titled test_module.cs. This file uses many of the functions provided by DLC_interface.cs, and thus both files are compiled as one project.

Hs.a51

```
NAME    XilinxInterface
; Version 0.9
;
; Based on ButtonsAndLight example from USB-By-Example

;               g) EP0Size made an equate to ease coding of other components
EP0Size   EQU     64              ; For EZ-USB

;
$INCLUDE(Declare.A51)
$INCLUDE(EZInt.A51)
$INCLUDE(EZMain.A51)
$INCLUDE(Decode.A51)
$INCLUDE(DTables.A51)

END
```

Declare.a51

```
; This module declares the variables and constants used in the examples
; It is common to all of the examples
;
; Declare Special Function Registers used
TimerControl        DATA    088H
TimerMode           DATA    089H
Timer0High          DATA    08CH
EI                  DATA    0A8H
EIE                 DATA    0E8H    ; EZ-USB specific
EXIF                DATA    091H    ; EZ-USB specific
EICON               DATA    0D8H    ; EZ-USB specific
PageReg             DATA    092H    ; EZ-USB specific, used with MOVX @Ri
DPS                 DATA    086H    ; EZ-USB specific, used with dual data pointers
;
; "External" memory locations used, EZ-USB specific
; Note that most of these variables are in Page 7FH
SETUPDAT            EQU     07FE8H
SUDPTR              EQU     07FD4H
EP0Control          EQU     07FB4H
EP0InBuffer         EQU     07F00H
EP0OutBuffer        EQU     07EC0H          ; Not in Page 7FH
EP1InBuffer         EQU     07E80H          ; Not in Page 7FH
IN0ByteCount        EQU     07FB5H
Out0ByteCount       EQU     07FC5H
IN1ByteCount        EQU     07FB7H
IN07IEN             EQU     07FACH
IN07IRQ             EQU     07FA9H
OUT07IEN            EQU     07FADH
OUT07IRQ            EQU     07FAAH
USBIEN              EQU     07FAEH
USBIRQ              EQU     07FABH
USBControl          EQU     07FD6H
I2CData             EQU     07FA6H
I2CControl          EQU     07FA5H
PortA_Config        EQU     07F93H
PortB_Config        EQU     07F94H
PortC_Config        EQU     07F95H
PortA_OUT           EQU     07F96H
PortB_OUT           EQU     07F97H
PortC_OUT           EQU     07F98H
PortA_PINS          EQU     07F99H
PortB_PINS          EQU     07F9AH
PortC_PINS          EQU     07F9BH
PortA_OE EQU        07F9CH
PortB_OE EQU        07F9DH
PortC_OE EQU        07F9EH
;
; Byte Variables

                    DSEG    AT 20H
FLAGS:              DS      1       ; This register is bit-addressable
; Bit Variables
Configured          EQU     FLAGS.0 ; Is this device configured
STALL               EQU     FLAGS.1 ; Need to STALL endpoint 0
SendData  EQU       FLAGS.2 ; Need to send data to PC Host
IsDescriptor        EQU     FLAGS.3 ; Enable a shortcut reply
SetAddress          EQU     FLAGS.4 ; Set the SIE address
;
MonitorSpace:       DS      1FH     ; Used by Dscope
;Expired_Time:      DS      1       ; A downcounter for timed Reports
ReplyCount:         DS      1       ; Byte count for following buffer
ReplyBuffer:        DS      2       ; Buffer for immediate reply
CurrentConfiguration:
                    DS      1       ; Some examples support > 1 configurations
SaveDPH: DS         1       ; Needed to save Descriptor Pointer ..
SaveDPL: DS         1       ; .. for descriptors > EP0Size
SaveLength:         DS      1       ; Number of bytes still to send
```

```
SetupData:                  ; Buffer in direct access memory
RequestType:        DS  1
Request:    DS      1
wValueLow:          DS  1
wValueHigh:         DS  1
wIndexLow:          DS  1
wIndexHigh:         DS  1
wLengthLow:         DS  1
wLengthHigh:        DS  1
;
;Old_Buttons:       DS  1       ; Used by BAL: stores current button position
;LEDstrobe:         DS  1       ; Used by BAL: strobe one LED on at a time
;LEDvalue:          DS  1       ; Used by BAL: stores current LED value
Msec_Counter:       DS  1       ; Used by BAL: counts up to 4 msec
INAddressA:         DS  1       ; Incoming Address from USB
INAddressB:         DS  1       ; Incoming Address from USB
OUTAddressA:        DS  1       ; Outgoing Address to USB
OUTAddressB:        DS  1       ; Outgoing Address to USB
INData:             DS  1       ; Incoming Data from USB
OUTData: DS         1       ; Outgoing Data to USb
INControl: DS       1       ; Control Byte containing Read/Write Info
ValidCount:         DS  1       ; Keeps count of valid outputs
;
```

EZInt.a51

```
; This module contains all the EZUSB-specific hardware code
; This module also contains all of the interrupt vector declarations and
; the first level interrupt servicing (register save, call subroutine,
; clear interrupt source, restore registers, return)
; Suspend and Resume are handled totally in this module
;
; A Reset sends us to Program space location 0
            CSEG AT 0                   ; Code space
            USING 0                     ; Reset forces Register Bank 0
            LJMP    Reset
;
; The interrupt vector table is also located here
; EZ-USB has two levels of USB interrupts:
; 1-the main level is described in this table (at ORG 43H)
; 2-there are 21 sources of USB interrupts and these are described in USB_ISR
; This means that two levels of acknowledgement and clearing will be required
;           LJMP    INT0_ISR ; Features not used are commented out
;           ORG     0BH
;           LJMP    Timer0_ISR
;           ORG     13H
;           LJMP    INT1_ISR
;           ORG     1BH
;           LJMP    Timer1_ISR
;           ORG     23H
;           LJMP    UART0_ISR
;           ORG     2BH
;           LJMP    Timer2_ISR
;           ORG     33H
;           LJMP    WakeUp_ISR
;           ORG     3BH
;           LJMP    UART1_ISR
            ORG     43H
            LJMP    USB_ISR             ; Auto Vector will replace byte 45H
;           ORG     4BH
;           LJMP    I2C_ISR
;           ORG     53H
;           LJMP    INT4_ISR
;           ORG     5BH
;           LJMP    INT5_ISR
;           ORG     63H
;           LJMP    INT6_ISR

            ORG     1200H               ; Load above monSIO0.hex
USB_ISR:LJMP    SUDAV_ISR
            DB      0                   ; Pad entries to 4 bytes
            LJMP    SOF_ISR
            DB      0
            LJMP    SUTOK_ISR
            DB      0
            LJMP    Suspend_ISR
            DB      0
            LJMP    USBReset_ISR
            DB      0
            LJMP    Reserved
            DB      0
            LJMP    EP0In_ISR
;           DB      0           ; Comment out features not used
;           LJMP    EP0Out_ISR
;           DB      0
;           LJMP    EP1In_ISR
;           DB      0
;           LJMP    EP1Out_ISR
;           DB      0
;           LJMP    EP2In_ISR
;           DB      0
;           LJMP    EP2Out_ISR
;           DB      0
```

```
;               LJMP    EP3In_ISR
;               DB      0
;               LJMP    EP3Out_ISR
;               DB      0
;               LJMP    EP4In_ISR
;               DB      0
;               LJMP    EP4Out_ISR
;               DB      0
;               LJMP    EP5In_ISR
;               DB      0
;               LJMP    EP5Out_ISR
;               DB      0
;               LJMP    EP6In_ISR
;               DB      0
;               LJMP    EP6Out_ISR
;               DB      0
;               LJMP    EP7In_ISR
;               DB      0
;               LJMP    EP7Out_ISR
; End of Interrupt Vector tables

; When a feature is used insert the required interrupt processing here
; The example use only used Endpoints 0 and 1 and also SOF for timing
Reserved:
INT0_ISR:
Timer0_ISR:
INT1_ISR:
Timer1_ISR:
UART0_ISR:
Timer2_ISR:
UART1_ISR:
I2C_ISR:
INT4_ISR:
INT5_ISR:
INT6_ISR:
SUTOK_ISR:
EP0Out_ISR:
EP1In_ISR:
EP1Out_ISR:
EP2In_ISR:
EP2Out_ISR:
EP3In_ISR:
EP3Out_ISR:
EP4In_ISR:
EP4Out_ISR:
EP5In_ISR:
EP5Out_ISR:
EP6In_ISR:
EP6Out_ISR:
EP7In_ISR :
EP7Out_ISR:
Not_Used:                       ; Should not get any of these
            RETI

ClearINT2:                              ; Tell the hardware that we're done
            MOV     A, EXIF
            CLR     ACC.4               ; Clear the Interrupt 2 bit
            MOV     EXIF, A
            RET

USBReset_ISR:                           ; Bus has been Reset, move to DEFAULT state
            CLR     Configured
      CALL          ClearINT2
                                        ; No need to clear source of interrupt
            RETI

Suspend_ISR:                            ; SIE detected an Idle bus
            MOV     A, PCON
            ORL     A, #1
            MOV     PCON, A             ; Go to sleep!
```

```
            NOP
            NOP                         ; Wake up here due to a USBResume
            NOP
            CALL    ClearINT2
            RETI

WakeUp_ISR:                             ; Not using external WAKEUP in these examples
                                        ; So this must be due to a USBResume
            CLR     EICON.4             ; Clear the wakeup interrupt source
            RETI

EP0In_ISR:                              ; A prepared packet has been read by PC host
            MOV     A, SaveLength       ; Do I have any more data to send?
            JZ      NoMoreToSend
            MOV     DPH, SaveDPH        ; Retreive descriptor pointer
            MOV     DPL, SaveDPL
            CALL    SendNextPieceOfDescriptor
NoMoreToSend:
            CALL    ClearINT2
            MOV     A, #00000001b
            MOV     DPTR, #IN07IRQ
            MOVX    @DPTR, A            ; Clear source of interrupt
            RETI

SOF_ISR:                                ; A Start-Of-Frame packet has been received
; This routine services the real time interrupt
; It is also responsible for the "real world" buttons and lights
;
ServiceTimerRoutine:
; LED routine moved to exmain.a51
            MOV     A, ValidCount
            CALL    CreateInputReport

Done:       CALL    ClearINT2
                        ; Clear the source of the interrupt
            MOV     A, #00000010b
ExitISR:MOV         DPTR, #USBIRQ
            MOVX    @DPTR, A
            RETI

SUDAV_ISR:                              ; A Setup packet has been received
            MOV     SaveLength, #0      ; Clear any pending transactions (if any)
            MOV     DPTR, #SETUPDAT     ; Copy packet to direct access memory
            MOV     R0, #SetupData
            MOV     R7, #8
CopySD:     MOVX    A, @DPTR
            MOV     @R0, A
            INC     DPTR
            INC     R0
            DJNZ    R7, CopySD
            CALL    ServiceSetupPacket  ; Handle the decode of the Setup packet
; if SetAddress { Update SIE address }  // NOP on EZ-USB
; if STALL { Stall the endpoint }
; if SendData {
;           if IsDescriptor { send DPTR->descriptor, A = length }
;           else { send ReplyBuffer }
;           }
            JB      STALL, SendSTALL
            JNB     SendData, HandShake
            JB      IsDescriptor, LoadEP0
                ; Send data in ReplyBuffer
            MOV     DPTR, #EP0InBuffer+1
            MOV     R0, #ReplyBuffer+1
            MOV     R7, #2              ; Copy the two byte buffer
CopyRB:     MOV     A, @R0
            MOVX    @DPTR, A
            DEC     DPL
            DEC     R0
            DJNZ    R7, CopyRB
            MOV     A, @R0              ; Get BufferCount
```

```
SendEP0InBuffer:
        MOV     DPTR, #In0ByteCount
StartXfer:
        MOVX    @DPTR, A                ; This write initiates the transfer
HandShake:                              ; Handshake with host
        MOV     R7, #00000010b          ; Set HSNAK to tell the SIE that we're done
SetEP0Control:
        MOV     DPTR, #EP0Control
        MOVX    A, @DPTR
        ORL     A, R7
        MOVX    @DPTR, A                ; We're done
        CALL    ClearINT2
        MOV     A, #00000001b           ; Clear the source of the interrupt
        JMP     ExitISR
SendSTALL:                              ; Invalid Request was received
        MOV     R7, #00000011b          ; Set EP0STALL and HSNAK
        JMP     SetEP0Control
LoadEP0:                                ; Send the data pointed to by DPTR
        MOV     R7, A                   ; Save LENGTH
; Need to return the smaller of "Requested Length" and "Actual Length"
; If "Requested Length" > 255 then use "Actual Length"
; There are no descriptors > 255 in this example
        MOV     A, wLengthHigh
        JNZ     UseActual
        CLR     C
        SUBB    A, wLengthLow
        MOV     A, wLengthLow           ; This does not affect Carry
        JNC     UsewLengthLow
UseActual:
        MOV     A, R7
UsewLengthLow:
SendNextPieceOfDescriptor:              ; DPTR -> Descriptor to be sent
        MOV     R7, A                   ; Save LENGTH again
        MOV     SaveLength, #0          ; Default case, overwrite if necessary
; Do I have more than a single packet to send?
        CLR     C
        SUBB    A, #EP0Size
        JC      SendPacket
; Need to send multiple packets.
; Calculate and save address of next packet, send next packet now
        MOV     SaveLength, A           ; Send these next time
        MOV     R7, #EP0Size
        PUSH    DPH                     ; Save current pointer
        PUSH    DPL
        MOV     A, R7                   ; Retreive length
        CALL    BumpDPTR
        MOV     SaveDPH, DPH
        MOV     SaveDPL, DPL
        POP     DPL
        POP     DPH
SendPacket:
        MOV     A, R7                   ; Retreive length
        MOV     R6, A                   ; Save length in R6 for move
        MOV     R0, #LOW(EP0InBuffer)   ; PageReg = 7FH = HIGH(EP0InBuffer)
CopySTD:MOVX    A, @DPTR
        MOVX    @R0, A
        INC     DPTR
        INC     R0
        DJNZ    R6, CopySTD
        MOV     A, R7                   ; Retrieve LENGTH
        JMP     SendEP0InBuffer

GetOutputReport:                        ; Wait for this, it's next on USB
        MOV     DPTR, #Out0ByteCount    ; Enable EP0OutBuffer to receive data
        MOVX    @DPTR, A                ; Any value will do
    MOV DPTR, #EP0Control   ; Wait for valid data in EP0OutBuffer
Wait40: MOVX    A, @DPTR
        ANL     A, #00001000b           ; Check OUTBSY
        JNZ     Wait40
        RET
```

EZMain.a51

```
; This module initializes the microcontroller then executes MAIN forever
; It is hardware dependant

Reset:
        MOV     SP, #0DFH               ; Initialize the Stack
        MOV     PageReg, #7FH           ; Allows MOVX Ri to access EZ-USB memory

        MOV     R0, #Low(USBControl)    ; Simulate a disconnect
        MOVX    A, @R0
        ANL     A, #11110011b           ; Clear DISCON, DISCOE
        MOVX    @R0, A
        CALL    Wait100msec             ; Give the host time to react
        MOVX    A, @R0                  ; Reconnect with this new identity
        ORL     A, #00000110b   ; Set DISCOE to enable pullup resistor
        MOVX    @R0, A                  ; Set RENUM so that 8051 handles USB requests
        CLR     A
        MOV     FLAGS, A        ; Start in Default state
InitVariables:
    MOV INControl, A
        MOV     INAddressA, A
        MOV     INAddressB, A
        MOV     OUTAddressA, A
        MOV     OUTAddressB, A
        MOV     INData, A
        MOV     OUTData, A
        MOV     ValidCount, A
Initialize4msecCounter:
        MOV     Msec_counter, A
InitializeIOSystem:             ; A=output, B=output C=input
        MOV     R0, #LOW(PortA_Config)  ; PageReg = 7F = HIGH(PortA_Config)
        CLR     A
        MOVX    @R0, A                  ; No alternate functions on PortA
        INC     R0
        MOVX    @R0, A                  ; No alternate functions on PortB
        INC     R0
        MOVX    @R0, A                  ; No alternate functions on PortC

        MOV     R1, #LOW(PortA_OE)
        CPL     A                       ; = 0FFH
        MOVX    @R1, A                  ; Enable PortA for Output
        INC     R1                      ; Point to PortB_OE
        MOVX    @R1, A                  ; Enable PortB for Output
        INC     R1                      ; Point to PortC_OE
        CLR     A
        MOVX    @R1, A                  ; Enable Port C for Input

InitializeInterruptSystem:      ; First initialize the USB level
        MOV     A, #00000001b
        MOV     R0, #LOW(IN07IEN)
        MOVX    @R0, A                  ; Enable interrupts from EP0IN only
        INC     R0
        CLR     A
        MOVX    @R0, A                  ; Disable interrupts from OUT Endpoints 0-7
        INC     R0
        MOV     A, #00000011b
        MOVX    @R0, A                  ; Enable (Resume, Suspend,) SOF and SUDAV INTs
        INC     R0
        MOV     A, #00000001b
        MOVX    @R0, A                  ; Enable Auto Vectoring for USB interrupts
                ; Now enable the main level
        MOV     EIE, #00000001b         ; Enable INT2 = USB Interrupt (only)
    MOV EI, #10010000b          ; Enable interrupt subsystem (and Ser0 for dScope)

; Initialization Complete.
;
MAIN:
        NOP                             ; Not much of a main loop for this example
```

```
        JMP     MAIN                    ; All actions are initiated by interrupts
; We are a slave, we wait to be told what to do

Wait100msec:
        MOV     R7, #100
Wait1msec:                              ; A delay loop
        MOV     DPS, #0                 ; Select primary DPTR
        MOV     DPTR, #-1200
More:   INC     DPTR                    ; 3 cycles
        MOV     A, DPL          ; + 2
        ORL     A, DPH          ; + 2
        JNZ     More            ; + 3 = 10 cycles x 1200 = 1msec
        DJNZ    R7, Wait1msec
        RET

ProcessOutputReport:                    ; A Report has just been received
; The report is four bytes long (Control, AddressA, AddressB, Data)
        MOV     DPTR, #EP0OutBuffer     ; Point to the Report
        MOVX    A, @DPTR                ; Get the Address Byte
        MOV     INControl,A             ; Move it into memory
        INC     DPTR
        MOVX    A,@DPTR                 ; Get the Data Byte
        MOV     INAddressA, A           ; Get the Address

        INC     DPTR
        MOVX    A,@DPTR                 ; Get the Data Byte
        MOV     INAddressB, A           ; Get the Data

        INC     DPTR
        MOVX    A,@DPTR                 ; Get the Data Byte
        MOV     INData, A       ; Get the Address

        MOV     A, INControl
        JZ      ReadfromXilinx

WritetoXilinx:
; Write Address first on PortB, pulsing Write Bit
; Write Data next on PortB, pulsing Write Bit
        MOV     A, INAddressA
        MOV     DPTR, #PortB_Out
        MOVX    @DPTR, A                ; Send the Address to Xilinx
    MOV     A, #16
    MOV     DPTR, #PortA_Out
    MOVX    @DPTR, A            ; Trigger set PortA4  (Set Write Addr Low Byte)

    CLR     A
    MOV     DPTR, #PortA_Out
    MOVX    @DPTR, A            ; Clear control bits

        MOV     A, INAddressB
        MOV     DPTR, #PortB_Out
        MOVX    @DPTR, A                ; Send the Address to Xilinx
    MOV     A, #48
    MOV     DPTR, #PortA_Out
    MOVX    @DPTR, A            ; Trigger set both bits (Set Write Addr High Byte)

    CLR     A
    MOV     DPTR, #PortA_Out
    MOVX    @DPTR, A            ; Clear control bits

        MOV     A, INData
        MOV     DPTR, #PortB_Out
        MOVX    @DPTR, A                ; Send the Data to Xilinx
    MOV     A, #32
    MOV     DPTR, #PortA_Out
    MOVX    @DPTR, A            ; Trigger set PortA5  (Set Write Data & Clear Write Addr)
    CLR     A
    MOVX    @DPTR, A            ; Trigger clear PortA5  (Clear Write Data)
        RET
```

```
ReadfromXilinx:
; Write Address first on PortB, pulsing Read Bit
; Read Data next on PortC
            MOV     A, INAddressA
            MOV     OUTAddressA, A
            MOV     DPTR, #PortB_Out
            MOVX    @DPTR, A                    ; Send the Address to Xilinx
      MOV    A, #16
      MOV    DPTR, #PortA_Out
      MOVX   @DPTR, A                  ; Trigger set PortA5  (Set Load Addr)

      CLR    A
      MOVX   @DPTR, A                  ; Trigger clear PortA5  (Clear Load Addr)

            MOV     A, INAddressB
            MOV     OUTAddressB, A
            MOV     DPTR, #PortB_Out
            MOVX    @DPTR, A                    ; Send the Address to Xilinx
      MOV    A, #48
      MOV    DPTR, #PortA_Out
      MOVX   @DPTR, A                  ; Trigger set PortA5  (Set Load Addr)

      CLR    A
      MOVX   @DPTR, A                  ; Trigger clear PortA5  (Clear Load Addr)

            MOV     DPTR, #PortC_Pins
            MOVX    A, @DPTR
            MOV     OUTData, A                  ; Read Data from Xilinx
            MOV     A, ValidCount
            INC     A
            MOV     ValidCount, A

CreateInputReport:                  ; Called when data is requested by Host
; The report is 4 bytes: Valid Byte, Address Low, Address High, Data
; Value in A is Valid Byte (leftover from above)
            MOV     DPTR, #EP1InBuffer ; Point to the buffer
            MOVX    @DPTR, A                    ; Ready Valid Byte
            INC     DPTR                        ; increment the buffer

            MOV     A, OUTAddressA
            MOVX    @DPTR, A                    ; Ready Address
            INC     DPTR                        ; increment the buffer

            MOV     A, OUTAddressB
            MOVX    @DPTR, A                    ; Ready Data
            INC     DPTR                        ;increment the buffer

            MOV     A, OUTData
            MOVX    @DPTR, A                    ; Ready Data
            INC     DPTR                        ;increment the buffer

            MOV     DPTR, #IN1ByteCount
            MOV     A, #4                       ; 4 total bytes now
            MOVX    @DPTR, A                    ; Endpoint 1 now 'armed', next IN will get data
            RET
```

Decode.a51

```
; This module is common to all of the examples.
; It decodes the USB Setup Packets and generates appropriate responses.
; Interpretation of Reports is handled by MAIN
;
            CSEG
ServiceSetupPacket:
            MOV     A, RequestType
            MOV     C, ACC.7            ; Bit 7 = 1 means IO device needs to send data to PC Host
            MOV     SendData, C
            ANL     A, #01011100b               ; IF RequestType[6.4.3.2] = 1 THEN goto BadRequest
            JNZ     BadRequest
            MOV     A, RequestType              ; IF RequestType[1&0] = 1 THEN goto BadRequest
            MOV     C, ACC.0
            ANL     C, ACC.1
            JC      BadRequest
            JNB     ACC.5, NotB5                ; IF RequestType[5] = 1 THEN RequestType[1,0] = [1,1]
            MOV     A, #00000011b
NotB5:      ANL     A, #00000011b               ; Set CommandIndex[5,4] = RequestType[1,0]
            SWAP    A
            MOV     R7, A                       ; Save HI nibble of CommandIndex
                                                ; Set CommandIndex[3,0] = Request[3,0]
            MOV     A, Request
            ANL     A, #11110000b               ; Check if Request > 15
            JNZ     BadRequest
            MOV     A, Request
            ANL     A, #00001111b               ; Only 13 are defined today, handle in table
            ORL     A, R7
;           CALL    CorrectSubroutine   ; goto CommandTable(CommandIndex)
CorrectSubroutine:                      ; Jump to the subroutine that DPTR is pointing to
            MOV     ReplyCount, #1              ; Set up a default reply
            MOV     ReplyBuffer, #0
            MOV     ReplyBuffer+1, #0
            CLR     SetAddress                  ; Clear all flags
            CLR     STALL
            CLR     IsDescriptor
            MOV     DPTR, #CommandTable
            CALL    BumpDPTR                    ; Point to entry
            MOVX    A, @DPTR                    ; Get the offset
            MOV     DPTR, #Subroutines
            JMP     @A+DPTR                             ; Go to the correct Subroutine

BadRequest:                                 ; Decoded a Bad Request, STALL the Endpoint
            SETB    STALL
            RET
                                            ; Support routines
NextDPTR:                                   ; Returns (DPTR + byte DPTR is pointing to)
            MOVX    A, @DPTR
BumpDPTR:                                   ; Returns (DPTR + ACC)
            ADD     A, DPL
            MOV     DPL, A
            JNC     Skip
            INC     DPH                     ; Need 16 bit arithmetic here
Skip:       RET

; Since the table only contains byte offsets, it is important that all these routines are
; within one page (100H) of Subroutines
; V3.0 - CommandTable moved outside of this one page limited space
CommandTable:
; First 16 commands are for the Device
            DB LOW(Device_Get_Status - Subroutines)
            DB LOW(Device_Clear_Feature - Subroutines)
            DB LOW(Invalid - Subroutines)
            DB LOW(Device_Set_Feature - Subroutines)
            DB LOW(Invalid - Subroutines)
            DB LOW(Set_Address - Subroutines)
            DB LOW(Get_Descriptor - Subroutines)
            DB LOW(Set_Descriptor - Subroutines)
```

```
                DB LOW(Get_Configuration - Subroutines)
                DB LOW(Set_Configuration - Subroutines)
                DB LOW(Invalid - Subroutines)
                DB LOW(Invalid - Subroutines)
                DB LOW(Invalid - Subroutines)
                DB LOW(Invalid - Subroutines)
                DB LOW(Invalid - Subroutines)
                DB LOW(Invalid - Subroutines)
; Next 16 commands are for the Interface
                DB LOW(Interface_Get_Status - Subroutines)
                DB LOW(Interface_Clear_Feature - Subroutines)
                DB LOW(Invalid - Subroutines)
                DB LOW(Interface_Set_Feature - Subroutines)
                DB LOW(Invalid - Subroutines)
                DB LOW(Invalid - Subroutines)
                DB LOW(Get_Class_Descriptor - Subroutines)
                DB LOW(Set_Class_Descriptor - Subroutines)
                DB LOW(Invalid - Subroutines)
                DB LOW(Invalid - Subroutines)
                DB LOW(Get_Interface - Subroutines)
                DB LOW(Set_Interface - Subroutines)
                DB LOW(Invalid - Subroutines)
                DB LOW(Invalid - Subroutines)
                DB LOW(Invalid - Subroutines)
                DB LOW(Invalid - Subroutines)
; Next 16 commands are for the Endpoint
                DB LOW(Endpoint_Get_Status - Subroutines)
                DB LOW(Endpoint_Clear_Feature - Subroutines)
                DB LOW(Invalid - Subroutines)
                DB LOW(Endpoint_Set_Feature - Subroutines)
                DB LOW(Invalid - Subroutines)
                DB LOW(Invalid - Subroutines)
                DB LOW(Invalid - Subroutines)
                DB LOW(Invalid - Subroutines)
                DB LOW(Invalid - Subroutines)
                DB LOW(Invalid - Subroutines)
                DB LOW(Invalid - Subroutines)
                DB LOW(Invalid - Subroutines)
                DB LOW(Endpoint_Sync_Frame - Subroutines)
                DB LOW(Invalid - Subroutines)
                DB LOW(Invalid - Subroutines)
                DB LOW(Invalid - Subroutines)
; Next 16 commands are Class Requests
                DB LOW(Invalid - Subroutines)
                DB LOW(Get_Report - Subroutines)
                DB LOW(Get_Idle - Subroutines)
                DB LOW(Get_Protocol - Subroutines)
                DB LOW(Invalid - Subroutines)
                DB LOW(Invalid - Subroutines)
                DB LOW(Invalid - Subroutines)
                DB LOW(Invalid - Subroutines)
                DB LOW(Invalid - Subroutines)
                DB LOW(Set_Report - Subroutines)
                DB LOW(Set_Idle - Subroutines)
                DB LOW(Set_Protocol - Subroutines)
                DB LOW(Invalid - Subroutines)
                DB LOW(Invalid - Subroutines)
                DB LOW(Invalid - Subroutines)
                DB LOW(Invalid - Subroutines)

Subroutines:
;
; Many requests are INVALID for this example
Get_Protocol:                   ; We are not a Boot device
Set_Protocol:                   ; We are not a Boot device
Set_Descriptor:                 ; Our Descriptors are static
Set_Class_Descriptor:   ; Our Descriptors are static
Set_Interface:                  ; We only have one Interface
Get_Interface:                  ; We do not have an Alternate setting
Set_Idle:               ; V3.0 Optional command, not supported
```

```
Get_Idle:                       ; V3.0 Optional command, not supported
Device_Set_Feature:             ; We have no features that can be set or cleared
Interface_Set_Feature:                  ; We have no features that can be set or cleared
Endpoint_Set_Feature:                   ; We have no features that can be set or cleared
Endpoint_Clear_Feature:         ; V3.0 We have no features that can be set or cleared
Device_Clear_Feature:                   ; We have no features that can be set or cleared
Interface_Clear_Feature:        ; We have no features that can be set or cleared
Endpoint_Sync_Frame:                    ; We are not an Isonchronous device

Invalid:                        ; Invalid Request made, STALL the Endpoint
        SETB    STALL
Reply:  RET

Set_Address:                            ; Set the address that the SIE will respond to
        SETB    SetAddress
        RET

Set_Report:                                     ; Host wants to sent us a Report.
; The ONLY case in this example where host sends data to us
        JNB     Configured, Invalid     ; Need to be Configured to do this command
        CALL    GetOutputReport                 ; Handled in EZUSB.A51
        JMP     ProcessOutputReport     ; RETurn via this subroutine
Get_Report:                                     ; Host wants a Report
        JNB     Configured, Invalid     ; Need to be Configured to do this command
        MOV     ReplyBuffer, #42H       ; Reply with a recognizable (arbitary) value
        RET
Get_Configuration:                      ; Respond with CurrentConfiguration
        MOV     ReplyBuffer, CurrentConfiguration
        RET
Device_Get_Status:                      ; Only two bits of Device Status are defined
        MOV     ReplyBuffer, #1                 ; Bit 1=Remote Wakeup(=0), Bit 0=Self Powered(=1)
        RET
Interface_Get_Status:                   ; Interface Status is currently defined as 0
Endpoint_Get_Status:
    MOV ReplyCount, #2                  ; Need a two byte 0 response
        RET
Set_Configuration:                      ; Valid values are 0 and 1
        MOV     A, wValueLow
        JZ      Deconfigured
        DEC     A
        JNZ     Invalid
        SETB    Configured
        MOV     CurrentConfiguration, #1
        RET
Deconfigured:
        CLR     Configured
        MOV     CurrentConfiguration, A
        RET
Get_Descriptor:                                 ; Host wants to know who/what we are
        SETB    IsDescriptor
        MOV     A, wValueHigh
        DEC     A                               ; Valid Values are 1, 2 and 3
        MOV     DPTR, #DeviceDescriptor
        JZ      ReturnLength
        DEC     A
        MOV     DPTR, #ConfigurationDescriptor
        JNZ     TryString
        MOV     A, #ConfigLength
        RET
TryString:
        DEC     A
        JNZ     Invalid
; Request is for a String Descriptor
        MOV     DPTR, #String0                  ; Point to String 0
        MOV     A, wValueLow                    ; Get String Index
NextString:
        JZ      ReturnLength
        MOV     R7, A                           ; Save String Index
        CALL    NextDPTR
        MOVX    A, @DPTR                        ; Get the String Length (= 0 means we're at Backstop)
```

```
            JZ          Invalid                     ; Asked for a string I don't have
            MOV         A, R7
            DEC         A
            JMP         NextString          ; Check if we are there yet
Get_Class_Descriptor:                               ; Valid values are 21H, 22H, 23H for Class Request
            SETB        IsDescriptor
            MOV         A, wValueHigh
            CLR         C
            SUBB        A, #21H
            MOV         DPTR, #HIDDescriptor
            JZ          ReturnLength
            DEC         A
            MOV         DPTR, #ReportDescriptor
            JZ          ReturnRDlength
;           DEC         A                           ; This example does not use Physical Descriptors
;           JZ          Send_Physical_Descriptor
            JMP         Invalid
;
ReturnLength:
            MOVX        A, @DPTR                    ; Get Descriptor Length (first byte)
            RET
ReturnRDlength:                                     ; Report Descriptor is different format
            MOV         A, #ReportLength
            RET
; Error check: this MUST be on within a page of Subroutines
WithinSamePage EQU $ - Subroutines
;
```

DTables.a51

```
; This module declares the descriptors
;
; This example has one Device Descriptor with:
;               One Configuration - single IN port and single OUT port
;               One Interface - there is only one method of accessing the ports
;               One HID Descriptor - to make PC host software simpler
;               One Endpoint Descriptor - for HID Input Reports
;               One Report Descriptor - one byte IN and one byte OUT reports
;               Multiple Sting Descriptors - to aid the user
;
                CSEG
DeviceDescriptor:
                DB      18, 1                   ; Length, Type
                DB      10H, 1                  ; USB Rev 1.1 (=0110H, low=10H, High=01H)
                DB      0, 0, 0                 ; Class, Subclass and Protocol
                DB      EP0Size
                DB      42H, 42H, 1, 42H, 0, 1; Vendor ID, Product ID and Version
                DB      1, 2, 0                 ; Manufacturer, Product & Serial# Names
                DB      1                       ; #Configs
ConfigurationDescriptor:
                DB      9, 2                    ; Length, Type
                DB      LOW(ConfigLength), HIGH(ConfigLength)
                DB      1, 1, 0                 ; #Interfaces, Configuration#, Config. Name
                DB      10000000b               ; Attributes = Bus Powered
                DB      250                     ; Max. Power is 250x2 = 500mA
InterfaceDescriptor:
                DB      9, 4                    ; Length, Type
                DB      0, 0, 1                 ; No alternate setting, HID uses EP1
                DB      3                       ; Class = Human Interface Device
                DB      0, 0                    ; Subclass and Protocol
                DB      0                       ; Interface Name
HIDDescriptor:
                DB      9, 21H                  ; Length, Type
                DB      0, 1                    ; HID Class Specification compliance
                DB      0                       ; Country localization (=none)
                DB      1                       ; Number of descriptors to follow
                DB      22H                     ; And it's a Report descriptor
                DB      LOW(ReportLength), HIGH(ReportLength)
EndpointDescriptor:
                DB      7, 5                    ; Length, Type
                DB      10000001b               ; Address = IN 1
                DB      00000011b               ; Interrupt
                DB      EP0Size, 0              ; Maximum packet size (this example only uses 1)
                DB      100                     ; Poll every 0.1 seconds
ConfigLength    EQU $ - ConfigurationDescriptor

ReportDescriptor:               ; Generated with HID Tool, copied to here
                DB      6, 0, 0FFH ; Usage_Page (Vendor Defined)
                DB      9, 1                    ; Usage (I/O Device)
                DB      0A1H, 1                 ; Collection (Application)
                DB      19H, 1                  ;   Usage_Minimum (Button 1)
                DB      29H, 8                  ;   Usage_Maximum (Button 8)
                DB      15H, 0                  ;   Logical_Minimum (0)
                DB      25H, 1                  ;   Logical_Maximum (1)
                DB      75H, 1                  ;   Report_Size (1)
                DB      95H, 32                 ;   Report_Count (8)
                DB      81H, 2                  ;   Input (Data,Var,Abs)
                DB      19H, 1                  ;   Usage_Minimum (Led 1)
                DB      29H, 24                 ;   Usage_Maximum (Led 8)
                DB      91H, 2                  ;   Output (Data,Var,Abs)
                DB      0C0H                    ; End_Collection
ReportLength    EQU $-ReportDescriptor

String0:                        ; Declare the UNICODE strings
                DB      4, 3, 9, 4   ; Only English language strings supported
String1:                        ; Manufacturer
                DB      (String2-String1),3 ; Length, Type
```

```
        DB      "U",0,"S",0,"B",0," ",0,"D",0,"e",0,"s",0,"i",0,"g",0,"n",0," ", 0
        DB      "B",0,"y",0," ",0,"E",0,"x",0,"a",0,"m",0,"p",0,"l",0,"e",0
String2:                        ; Product Name
        DB      (EndOfDescriptors-String2),3
        DB      "G",0,"e",0,"o",0,"r",0,"g",0,"i",0,"a",0," ",0,"T",0,"e",0,"c",0,"h",0," ",0
        DB      "T",0,"e",0,"s",0,"t",0," ",0,"C",0,"o",0,"r",0,"e",0," ",0,"V",0,"e",0,"r",0,"2",0
EndOfDescriptors:
        DB      0                       ; Backstop for String Descriptors
```

DLC_Interface.cs

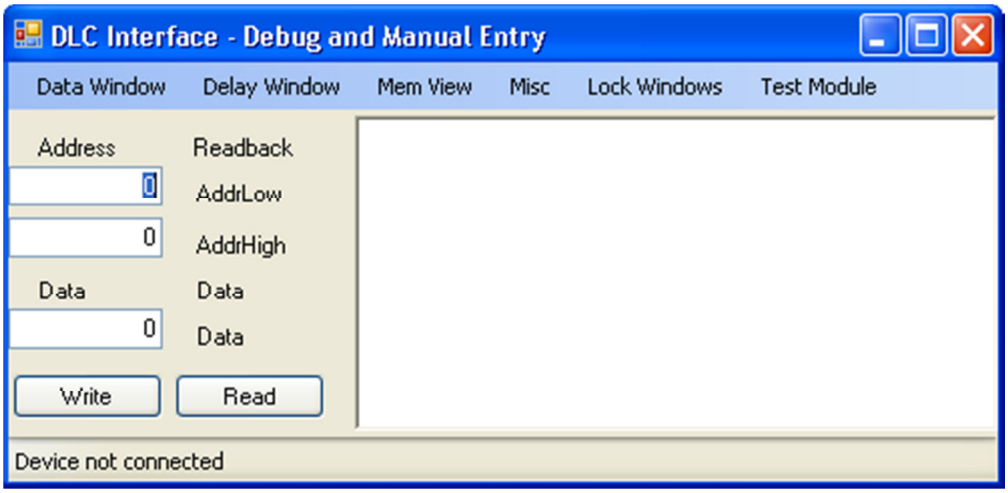


Figure B.1 DLC software interface

```
using System;
using System.Collections;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Windows.Forms;
using Microsoft.Win32.SafeHandles;
using System.Runtime.InteropServices;
using System.Diagnostics;
using System.Threading;
using Microsoft.VisualBasic;

namespace BitParallel_HID_Interface
{
    internal enum reportType { Read, Write };

    public partial class BitParallel : Form
    {
        private Byte[] dummyReport = new Byte[5];
        private Byte validByte = 0;

        private IntPtr deviceNotificationHandle;
        private SafeFileHandle hidHandle;
        private String hidUsage;
        private Boolean myDeviceDetected;
        private String myDevicePathName;
        private SafeFileHandle readHandle;
        private SafeFileHandle writeHandle;

        private Boolean exclusiveAccess;

        private DelayWindow frameDelay;
        private DataWindow frameData;
        private MemoryPanel frameMemory;
        private Core_Module coreWindow;

        private Debugging MyDebugging = new Debugging(); //  For viewing results of API
calls via Debug.Write.
        private DeviceManagement MyDeviceManagement = new DeviceManagement();
        internal Hid MyHid = new Hid();

        public Boolean emulate = false;
```

```
        public BitParallel()
        {
            InitializeComponent();
        }

        private void BitParallel_Load(object sender, EventArgs e)
        {
            FindTheHid();
            frameDelay = new DelayWindow(this);
            frameData = new DataWindow(this);
            frameMemory = new MemoryPanel(this);
            coreWindow = new Core_Module(this);
            frameDelay.Visible = false;
            frameData.Visible = false;
            coreWindow.Visible = false;
            frameDelay.Owner = this;
            frameData.Owner = this;
            coreWindow.Owner = this;
            if (myDeviceDetected == true)
            {
                Boolean successRead = false;
                successRead = ReadInputReport(ref dummyReport);
                if (successRead == true)
                {
                    validByte = dummyReport[1];
                    labelValid.Text = validByte.ToString();
                }
            }

            timer1.Enabled = false;
        }

        private void buttonWrite_Click(object sender, EventArgs e)
        {
            String byteValue = null;
            Int32 count = 0;
            Boolean success = false;
            Byte[] outputReportBuffer = null;

            if (MyHid.Capabilities.OutputReportByteLength > 3)
            {
                outputReportBuffer = new Byte[MyHid.Capabilities.OutputReportByteLength];

                //  Store the report ID in the first byte of the buffer:
                outputReportBuffer[0] = 0; // report ID

                outputReportBuffer[1] = Convert.ToByte(reportType.Write);

                outputReportBuffer[2] = Convert.ToByte(textBoxAddrLow.Text);

                if (MyHid.Capabilities.OutputReportByteLength == 5)
                {

                    outputReportBuffer[3] = Convert.ToByte(textBoxAddrHigh.Text);
                    outputReportBuffer[4] = Convert.ToByte(textBoxData.Text);  // double
byte addressing detected, include high address and then data
                }
                else
                {
                    outputReportBuffer[3] = Convert.ToByte(textBoxData.Text);  // single
byte addressing detected, ignore high address
                }
            }
            success = SendOutputReport(ref outputReportBuffer);

            if (success)
            {
                DebugLine("An Output report has been written.");

                //  Display the report data in the form's list box.
```

```
                DebugLine(" Output Report ID: " + String.Format("{0:X2} ",
outputReportBuffer[0]));
                DebugLine(" Output Report Data:");

                for (count = 0; count <= outputReportBuffer.Length - 1; count++)
                {
                    //  Display bytes as 2-character hex strings.
                    byteValue = String.Format("{0:X2} ", outputReportBuffer[count]);
                    DebugLine(byteValue);
                }
            }
            else
            {
                DebugLine("The attempt to write an Output report has failed.");
            }
        }

        private void buttonRead_Click(object sender, EventArgs e)
        {
            Boolean successWrite = false;
            Byte[] outputReportBuffer = new
Byte[MyHid.Capabilities.OutputReportByteLength];

            if (MyHid.Capabilities.OutputReportByteLength > 3)
            {
                outputReportBuffer[0] = 0; // reportID
                outputReportBuffer[1] = Convert.ToByte(reportType.Read);
                outputReportBuffer[2] = Convert.ToByte(textBoxAddrLow.Text);

                if (MyHid.Capabilities.OutputReportByteLength == 5)
                {

                    outputReportBuffer[3] = Convert.ToByte(textBoxAddrHigh.Text);
                    outputReportBuffer[4] = Convert.ToByte(textBoxData.Text);  // double
byte addressing detected, include high address and then data
                }
                else
                {
                    outputReportBuffer[3] = Convert.ToByte(textBoxData.Text);  // single
byte addressing detected, ignore high address
                }
            }
            successWrite = SendOutputReport(ref outputReportBuffer);
            if (successWrite == true)
            {
                DebugLine("Read address written.  Waiting for result");
                validByte++;
                int retry = 0;
                while (retry < 128)
                {
                    ReadInputReport(ref dummyReport);
                    if (dummyReport[1] == validByte)
                    {
                        DebugLine("Read success: " + dummyReport[4]);
                        retry = 128;
                    }
                    else
                    {
                        if(menuItemSilenceOverride.Checked)
                            DebugLine("Saw unexpected byte on try " + retry + ": " +
dummyReport[1] + ".  Expected: " + validByte);
                        Thread.Sleep(10);
                        retry++;
                    }
                }

            }

        }

        internal Boolean SendOutputReport(ref Byte[] outputReportBuffer)
```

```
        {
            Boolean success = false;

            if (myDeviceDetected == false)
                FindTheHid();

            if (myDeviceDetected == false)
                return false;

            //debugText.Focus();

            try
            {
                //  Don't attempt to exchange reports if valid handles aren't available
                //  (as for a mouse or keyboard under Windows 2000/XP.)
                if (!readHandle.IsInvalid && !writeHandle.IsInvalid)
                {
                    //  Don't attempt to send an Output report if the HID Output report
is too small.
                    if (MyHid.Capabilities.OutputReportByteLength > 3)
                    {
                        //  Write a report.
                        Hid.OutputReportViaInterruptTransfer myOutputReport = new
Hid.OutputReportViaInterruptTransfer();
                        success = myOutputReport.Write(outputReportBuffer, writeHandle);
                    }
                    else
                    {
                        DebugLine("The HID doesn't have an Output report or it's too
small (" + MyHid.Capabilities.OutputReportByteLength + ").");
                    }
                }
                else
                {
                    DebugLine("Invalid handle. The device is probably a system mouse or
keyboard.");
                    DebugLine("No attempt to write an Output report or read an Input
report was made.");
                }
            }
            catch (Exception ex)
            {
                throw;
            }

            return success;
        }

        internal Boolean ReadInputReport(ref Byte[] inputReportBuffer)
        {
            Boolean success = false;

            if (myDeviceDetected == false)
                FindTheHid();

            if (myDeviceDetected == false)
                return false;

            try
            {
                Hid.InputReportViaInterruptTransfer myInputReport = new
Hid.InputReportViaInterruptTransfer();
                myInputReport.Read(hidHandle, readHandle, writeHandle, ref
myDeviceDetected, ref inputReportBuffer, ref success);
            }
            catch (Exception ex)
            {
                throw;
            }

            return success;
```

```
        }

        internal Boolean SimpleSend(byte lowerAddr, byte upperAddr, byte data, Boolean
silent)
        {
            Boolean success = false;
            int dataLoc = 3;

            silent = silent & !menuItemSilenceOverride.Checked;
            //if (silent == false)
                //debugText.Focus();  // only give the window focus if we need to output
something

            Byte[] outputReportBuffer = new
Byte[MyHid.Capabilities.OutputReportByteLength];

            if (emulate)
                outputReportBuffer = new Byte[4];

            if (outputReportBuffer.Length >= 4)
            {
                //  Store the report ID in the first byte of the buffer:
                outputReportBuffer[0] = 0; // report ID
                outputReportBuffer[1] = Convert.ToByte(reportType.Write);
                outputReportBuffer[2] = lowerAddr;
                if (MyHid.Capabilities.OutputReportByteLength == 5)
                {

                    outputReportBuffer[3] = upperAddr;
                    outputReportBuffer[4] = data;
                    dataLoc = 4;
                }
                else
                {
                    outputReportBuffer[3] = data;
                }
                if(!silent)
                    DebugAdd("Sending " + outputReportBuffer[dataLoc].ToString() + " to
address " + outputReportBuffer[2]);
                success = SendOutputReport(ref outputReportBuffer);
                if(!silent)
                    DebugLine(" succeeded: " + success);
            }

            return success;
        }

        internal Boolean SimpleRead(byte lowerAddr, byte upperAddr, ref byte data,
Boolean silent)
        {
            Boolean successWrite = false;
            Boolean successRead = false;
            Byte[] outputReportBuffer = new
Byte[MyHid.Capabilities.OutputReportByteLength];
            int dataLoc = 4;

            silent = silent | menuItemSilenceOverride.Checked;
            //if (silent == false)
                //debugText.Focus();  // only give the window focus if we need to output
something

            if (MyHid.Capabilities.OutputReportByteLength > 3)
            {
                outputReportBuffer[0] = 0; // reportID
                outputReportBuffer[1] = Convert.ToByte(reportType.Read);
                outputReportBuffer[2] = lowerAddr;

                if (MyHid.Capabilities.OutputReportByteLength == 5)
                {

                    outputReportBuffer[3] = upperAddr;
```

```
                    outputReportBuffer[4] = 0;  // double byte addressing detected,
include high address and then bogus data
                }
                else
                {
                    outputReportBuffer[3] = 0;  // single byte addressing detected,
ignore high address
                    dataLoc = 3;
                }
            }

            /* So why did we just send a report when we're really reading?  By declaring
reportType.Read we're instructing the device
             * to store and enable the address which will then be used to populate the
proper data onto the cypress input pins.
             * This data will eventually be present in the output report along with a
ValidByte incrementally higher than the last time
             * we issued a read request.  The result may not be immediately available, so
poll the device a few times.  Run this in a
             * seperate process or ensure a timeout to prevent deadlock
             */

            successWrite = SendOutputReport(ref outputReportBuffer);
            if (successWrite == true)
            {
                if(!silent)
                    DebugLine("Read address written.  Waiting for result");
                validByte++;
                int retry = 0;
                while (retry < 128)
                {
                    ReadInputReport(ref dummyReport);
                    if (dummyReport[1] == validByte)
                    {
                        if(!silent)
                            DebugLine("Read success: " + dummyReport[dataLoc]);
                        data = dummyReport[dataLoc];
                        successRead = true;
                        retry = 128;  // kludge to break early
                    }
                    else
                    {
                        //DebugLine("Saw unexpected byte: " + dummyReport[1] + ".
Expected: " + validByte);
                        Thread.Sleep(10);
                        retry++;
                    }
                }

            }

            return successRead;
        }

        private Boolean FindTheHid()
        {
            Boolean deviceFound = false;
            String[] devicePathName = new String[128];
            String functionName = "";
            Guid hidGuid = Guid.Empty;
            Int32 memberIndex = 0;
            Int16 myVendorID = Convert.ToInt16("4242", 16);
            Int16 myProductID = Convert.ToInt16("4201", 16);
            Boolean success = false;

            //debugText.Focus();

            try
            {
                Debug.WriteLine("Attempting to open HID Devices");
                myDeviceDetected = false;
```

```
                //  ***
                //  API function: 'HidD_GetHidGuid

                //  Purpose: Retrieves the interface class GUID for the HID class.

                //  Accepts: 'A System.Guid object for storing the GUID.
                //  ***

                Hid.HidD_GetHidGuid(ref hidGuid);

                functionName = "GetHidGuid";
                Debug.WriteLine(MyDebugging.ResultOfAPICall(functionName));
                Debug.WriteLine("  GUID for system HIDs: " + hidGuid.ToString());

                //  Fill an array with the device path names of all attached HIDs.

                deviceFound = MyDeviceManagement.FindDeviceFromGuid(hidGuid, ref
devicePathName);

                //  If there is at least one HID, attempt to read the Vendor ID and
Product ID
                //  of each device until there is a match or all devices have been
examined.

                if (deviceFound)
                {
                    memberIndex = 0;

                    do
                    {
                        //  ***
                        //  API function:
                        //  CreateFile

                        //  Purpose:
                        //  Retrieves a handle to a device.

                        //  Accepts:
                        //  A device path name returned by
SetupDiGetDeviceInterfaceDetail
                        //  The type of access requested (read/write).
                        //  FILE_SHARE attributes to allow other processes to access the
device while this handle is open.
                        //  A Security structure or IntPtr.Zero.
                        //  A creation disposition value. Use OPEN_EXISTING for devices.
                        //  Flags and attributes for files. Not used for devices.
                        //  Handle to a template file. Not used.

                        //  Returns: a handle without read or write access.
                        //  This enables obtaining information about all HIDs, even
system
                        //  keyboards and mice.
                        //  Separate handles are used for reading and writing.
                        //  ***

                        hidHandle = FileIO.CreateFile(devicePathName[memberIndex], 0,
FileIO.FILE_SHARE_READ | FileIO.FILE_SHARE_WRITE, IntPtr.Zero, FileIO.OPEN_EXISTING, 0,
0);

                        functionName = "CreateFile";
                        Debug.WriteLine(MyDebugging.ResultOfAPICall(functionName));
                        Debug.WriteLine("  Returned handle: " + hidHandle.ToString());

                        if (!hidHandle.IsInvalid)
                        {
                            //  The returned handle is valid,
                            //  so find out if this is the device we're looking for.

                            //  Set the Size property of DeviceAttributes to the number
of bytes in the structure.
```

```
                            MyHid.DeviceAttributes.Size =
Marshal.SizeOf(MyHid.DeviceAttributes);

                            //  ***
                            //  API function:
                            //  HidD_GetAttributes

                            //  Purpose:
                            //  Retrieves a HIDD_ATTRIBUTES structure containing the
Vendor ID,
                            //  Product ID, and Product Version Number for a device.

                            //  Accepts:
                            //  A handle returned by CreateFile.
                            //  A pointer to receive a HIDD_ATTRIBUTES structure.

                            //  Returns:
                            //  True on success, False on failure.
                            //  ***

                            success = Hid.HidD_GetAttributes(hidHandle, ref
MyHid.DeviceAttributes);

                            if (success)
                            {
                                Debug.WriteLine("  HIDD_ATTRIBUTES structure filled
without error.");
                                Debug.WriteLine("  Structure size: " +
MyHid.DeviceAttributes.Size);
                                Debug.WriteLine("  Vendor ID: " +
Convert.ToString(MyHid.DeviceAttributes.VendorID, 16));
                                Debug.WriteLine("  Product ID: " +
Convert.ToString(MyHid.DeviceAttributes.ProductID, 16));
                                Debug.WriteLine("  Version Number: " +
Convert.ToString(MyHid.DeviceAttributes.VersionNumber, 16));

                                //  Find out if the device matches the one we're looking
for.

                                Debug.WriteLine("Looking for: " + myVendorID + " and " +
myProductID);
                                Debug.WriteLine("Looking at " +
MyHid.DeviceAttributes.VendorID + " and " + MyHid.DeviceAttributes.ProductID);

                                if ((MyHid.DeviceAttributes.VendorID == myVendorID) &&
(MyHid.DeviceAttributes.ProductID == myProductID))
                                {

                                    Debug.WriteLine("  My device detected");

                                    //  Display the information in form's list box.

                                    myDeviceDetected = true;

                                    //  Save the DevicePathName for OnDeviceChange().

                                    myDevicePathName = devicePathName[memberIndex];
                                }
                                else
                                {
                                    //  It's not a match, so close the handle.

                                    myDeviceDetected = false;
                                    hidHandle.Close();
                                }
                            }
                            else
                            {
                                //  There was a problem in retrieving the information.
```

```
                                Debug.WriteLine("  Error in filling HIDD_ATTRIBUTES
structure.");
                                myDeviceDetected = false;
                                hidHandle.Close();
                            }
                        }

                        //  Keep looking until we find the device or there are no devices
left to examine.

                        memberIndex = memberIndex + 1;
                    }
                    while (!((myDeviceDetected || (memberIndex ==
devicePathName.Length))));
                }

                if (myDeviceDetected)
                {
                    //  The device was detected.
                    //  Register to receive notifications if the device is removed or
attached.

                    success =
MyDeviceManagement.RegisterForDeviceNotifications(myDevicePathName, this.Handle, hidGuid,
ref deviceNotificationHandle);

                    Debug.WriteLine("RegisterForDeviceNotifications = " + success);

                    //timer1.Enabled = true;

                    //  Learn the capabilities of the device.

                    MyHid.Capabilities = MyHid.GetDeviceCapabilities(hidHandle);

                    if (success)
                    {
                        //  Find out if the device is a system mouse or keyboard.

                        hidUsage = MyHid.GetHidUsage(MyHid.Capabilities);

                        //  Get handles to use in requesting Input and Output reports.

                        readHandle = FileIO.CreateFile(myDevicePathName,
FileIO.GENERIC_READ, FileIO.FILE_SHARE_READ | FileIO.FILE_SHARE_WRITE, IntPtr.Zero,
FileIO.OPEN_EXISTING, FileIO.FILE_FLAG_OVERLAPPED, 0);

                        functionName = "CreateFile, ReadHandle";
                        Debug.WriteLine(MyDebugging.ResultOfAPICall(functionName));
                        Debug.WriteLine("  Returned handle: " + readHandle.ToString());

                        if (readHandle.IsInvalid)
                        {
                            exclusiveAccess = true;
                        }
                        else
                        {
                            writeHandle = FileIO.CreateFile(myDevicePathName,
FileIO.GENERIC_WRITE, FileIO.FILE_SHARE_READ | FileIO.FILE_SHARE_WRITE, IntPtr.Zero,
FileIO.OPEN_EXISTING, 0, 0);

                            functionName = "CreateFile, WriteHandle";
                            Debug.WriteLine(MyDebugging.ResultOfAPICall(functionName));
                            Debug.WriteLine("  Returned handle: " +
writeHandle.ToString());

                            //  Flush any waiting reports in the input buffer. (optional)

                            MyHid.FlushQueue(readHandle);
                        }
                    }
                }
```

```
                else
                {
                    //  The device wasn't detected.

                    Debug.WriteLine(" Device not found.");
                    timer1.Enabled = false;
                }

                if (myDeviceDetected == true)
                    toolStripStatusConnected.Text = "Device connected";
                else
                    toolStripStatusConnected.Text = "Device not connected";
                return myDeviceDetected;
            }
            catch (Exception ex)
            {
                throw;
            }
        }

        internal void OnDeviceChange(Message m)
        {
            Debug.WriteLine("WM_DEVICECHANGE");

            try
            {
                if ((m.WParam.ToInt32() == DeviceManagement.DBT_DEVICEARRIVAL))
                {

                    //  If WParam contains DBT_DEVICEARRIVAL, a device has been attached.

                    DebugLine("A device has been attached.");

                    //  Find out if it's the device we're communicating with.

                    if (MyDeviceManagement.DeviceNameMatch(m, myDevicePathName))
                    {
                        DebugLine("My device attached.");
                        toolStripStatusConnected.Text = "Device reattached";  // we see
it, but the handle may have changed
                        //rerun findTheHID on next transaction to reenable the connection
                        timer1.Enabled = true;
                    }

                }
                else if ((m.WParam.ToInt32() ==
DeviceManagement.DBT_DEVICEREMOVECOMPLETE))
                {

                    //  If WParam contains DBT_DEVICEREMOVAL, a device has been removed.

                    DebugLine("A device has been removed.");

                    //  Find out if it's the device we're communicating with.

                    if (MyDeviceManagement.DeviceNameMatch(m, myDevicePathName))
                    {

                        DebugLine("My device removed.");

                        //  Set MyDeviceDetected False so on the next data-transfer
attempt,
                        //  FindTheHid() will be called to look for the device
                        //  and get a new handle.

                        myDeviceDetected = false;
                        //timer1.Enabled = false;
                        toolStripStatusConnected.Text = "Device not connected";
                    }
                }
            }
```

```
    catch (Exception ex)
    {
        throw;
    }
}

private void dataWindowToolStripMenuItem_Click(object sender, EventArgs e)
{
    if (frameData.Visible == true)
        frameData.Hide();
    else
    {
        Point loc = this.DesktopLocation;
        Size frameSize = this.Size;
        int x = loc.X + frameSize.Width;
        int y = loc.Y;
        frameData.SetDesktopLocation(x, y);
        frameData.Show();
    }
}

private void delayWindowToolStripMenuItem_Click(object sender, EventArgs e)
{
    if (frameDelay.Visible == true)
        frameDelay.Hide();
    else
    {
        Point loc = this.DesktopLocation;
        Size frameSize = this.Size;
        int x = loc.X;
        int y = loc.Y + frameSize.Height;
        frameDelay.SetDesktopLocation(x, y);
        frameDelay.Show();
    }
}

private void memViewToolStripMenuItem_Click(object sender, EventArgs e)
{
    if (frameMemory.Visible == true)
        frameMemory.Hide();
    else
    {
        frameMemory.Show();
    }
}

protected override void WndProc(ref Message m)
{
    try
    {
        //  The OnDeviceChange routine processes WM_DEVICECHANGE messages.

        if (m.Msg == DeviceManagement.WM_DEVICECHANGE)
        {
            OnDeviceChange(m);
        }

        //  Let the base form process the message.

        base.WndProc(ref m);
    }
    catch (Exception ex)
    {
        DisplayException(this.Name, ex);
        throw;
    }
}

internal static void DisplayException(String moduleName, Exception e)
{
    String message = null;
```

```
            String caption = null;

            //  Create an error message.

            message = "Exception: " + e.Message + "\r\n" + "Module: " + moduleName +
"\r\n" + "Method: " + e.TargetSite.Name;

            caption = "Unexpected Exception";

            MessageBox.Show(message, caption, MessageBoxButtons.OK);
            Debug.Write(message);
        }

        internal void DebugLine(String newText)
        {
            debugText.AppendText(newText + "\r\n");
        }

        internal void DebugAdd(String newText)
        {
            debugText.AppendText(newText);
        }

        private void timer1_Tick(object sender, EventArgs e)
        {
            Boolean successRead = false;

            Byte[] inputReportBuffer = new
Byte[MyHid.Capabilities.OutputReportByteLength];
            successRead = ReadInputReport(ref inputReportBuffer);

            if (myDeviceDetected == false)
                return;

            if (successRead == true)
            {
                for (int i = 0; i < inputReportBuffer.Length; i++)
                {
                    switch (i)
                    {
                        case (0): // fallthrough, don't care about the report ID
                        case (1): labelValid.Text = inputReportBuffer[i].ToString();
                            break;
                        case (2): labelAddrLow.Text = inputReportBuffer[i].ToString();
                            break;
                        case (3): if (inputReportBuffer.Length > 4)
                            {  // if the report length is 5, we're double byte addressing
                                labelAddrHigh.Text = inputReportBuffer[i].ToString();
                                labelData.Text = inputReportBuffer[i + 1].ToString();
                            }
                            else
                            {  // otherwise it's just single byte addressing and the high
address is irrelevant
                                labelAddrHigh.Text = "N/A";
                                labelData.Text = inputReportBuffer[i].ToString();
                            }
                            break;
                    }
                }
            }
            else
            {
                DebugLine("Read operation appears to have failed.");
            }
        }

        private void synchronizeToolStripMenuItem_Click(object sender, EventArgs e)
        {
            syncValidByte();
        }
```

```
        private void syncValidByte()
        {
            if (myDeviceDetected == true)
            {
                Boolean successRead = false;
                successRead = ReadInputReport(ref dummyReport);
                if (successRead == true)
                {
                    validByte = dummyReport[1];
                    DebugLine("Valid Byte reset to: " + validByte);
                    labelValid.Text = validByte.ToString();
                }
            }
        }

        private void BitParallel_Move(object sender, EventArgs e)
        {
            Point loc = this.DesktopLocation;
            Size frameSize = this.Size;

            if ((frameDelay.Visible == true) && (delayWindowLock.Checked == true))
            {
                int x = loc.X;
                int y = loc.Y + frameSize.Height;;
                frameDelay.SetDesktopLocation(x, y);
                frameDelay.Show();
            }

            if ((frameData.Visible == true) && (dataWindowLock.Checked == true))
            {
                int x = loc.X + frameSize.Width;
                int y = loc.Y;
                frameData.SetDesktopLocation(x, y);
                frameData.Show();
            }


        }

        private void mainWindowTickTimerToolStripMenuItem_CheckStateChanged(object
sender, EventArgs e)
        {
            if (mainWindowTickTimerToolStripMenuItem.Checked == true)
                timer1.Enabled = true;
            else
                timer1.Enabled = false;
        }

        private void debugText_TextChanged(object sender, EventArgs e)
        {
            SuspendLayout();
            Point pt = debugText.GetPositionFromCharIndex(debugText.SelectionStart);
            if (pt.Y > debugText.Height)
            {
                debugText.ScrollToCaret();
            }
            ResumeLayout(true);
        }

        private void ashTestToolStripMenuItem_Click(object sender, EventArgs e)
        {
            if (coreWindow.Visible == true)
                coreWindow.Hide();
            else
            {
                Point loc = this.DesktopLocation;
                Size frameSize = this.Size;
                int x = loc.X;
                int y = loc.Y + frameSize.Height;
                coreWindow.SetDesktopLocation(x, y);
                coreWindow.Show();
```

```
                }
            }

        }
    }
```

Test_module.cs

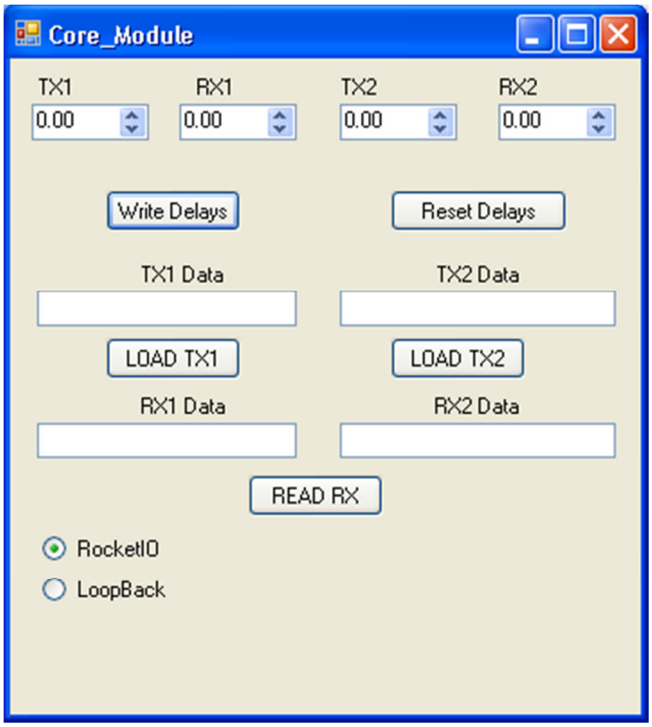


Figure B.2 Test module software interface

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Windows.Forms;

namespace BitParallel_HID_Interface
{
    public partial class Core_Module : Form
    {

        private BitParallel bp;

        public Core_Module(BitParallel source)
        {
            InitializeComponent();
            bp = source;
        }

        private void label1_Click(object sender, EventArgs e)
        {

        }

        private void button1_Click(object sender, EventArgs e)
        {
            NumericUpDown[] delayValArray = new
NumericUpDown[]{numericUpDown1,numericUpDown2,numericUpDown3,numericUpDown4};
            byte writeAddr = 6;
```

```
            foreach (NumericUpDown delayUpDown in delayValArray){

                int delayVal = (int) (delayUpDown.Value * 100);
                byte lowByte = (byte) (delayVal % 256);
                byte highByte =(byte) (delayVal / 256);

                bp.SimpleSend(writeAddr, 0, lowByte, true);
                writeAddr++;
                bp.SimpleSend(writeAddr, 0, highByte, true);
                writeAddr++;
            }

            bp.SimpleSend(0, 0, 2, true);

        }

        private void button2_Click(object sender, EventArgs e)
        {
            numericUpDown1.Value = 0;
            numericUpDown2.Value = 0;
            numericUpDown3.Value = 0;
            numericUpDown4.Value = 0;

            button1_Click(sender, e);

            //byte test = 0;
            //bp.SimpleRead(6, 0, ref test, true);

        }

        private void radioButton1_CheckedChanged(object sender, EventArgs e)
        {

        }
    }
}
```

APPENDIX C

Physical Board Design and Layout

In this appendix, details of the test module's physical design and layout are presented. As discussed in Chapter 7, the test module PCB is designed using Mentor Graphics PADS software suite. Specifically, two applications of the suite are used, namely it's logic editor and its layout editor. The test module components are first entered into the logic editor and connected to each other according to design. Once this step is complete, the design is exported to the layout editor, where the physical placement and routing of the components is done.

The logic design of the test module was divided up into three parts. The first part consisted of the core logic block. This contained the FPGA, flash memory, microcontroller, and the programmable delay chips. The schematic for this section is shown in Figure C.1. The second part contained the application specific logic, namely buffers, fan-out buffers, multiplexors and XOR gate. The schematic for this section is shown in Figure C.2. The final part of logic entry contained all the connectors used on the test module, i.e. the 40-pin connectors, SMP connectors, power posts, etc. The schematic is shown in Figure C.3.

Once the logic design of the test module it completed, it is exported to the layout editor. As discussed in Chapter 7, the test module is designed on a 10-layer PCB board (shown in Figure 7.2). The layouts for each layer are included in Figure C.4-Figure C.13.

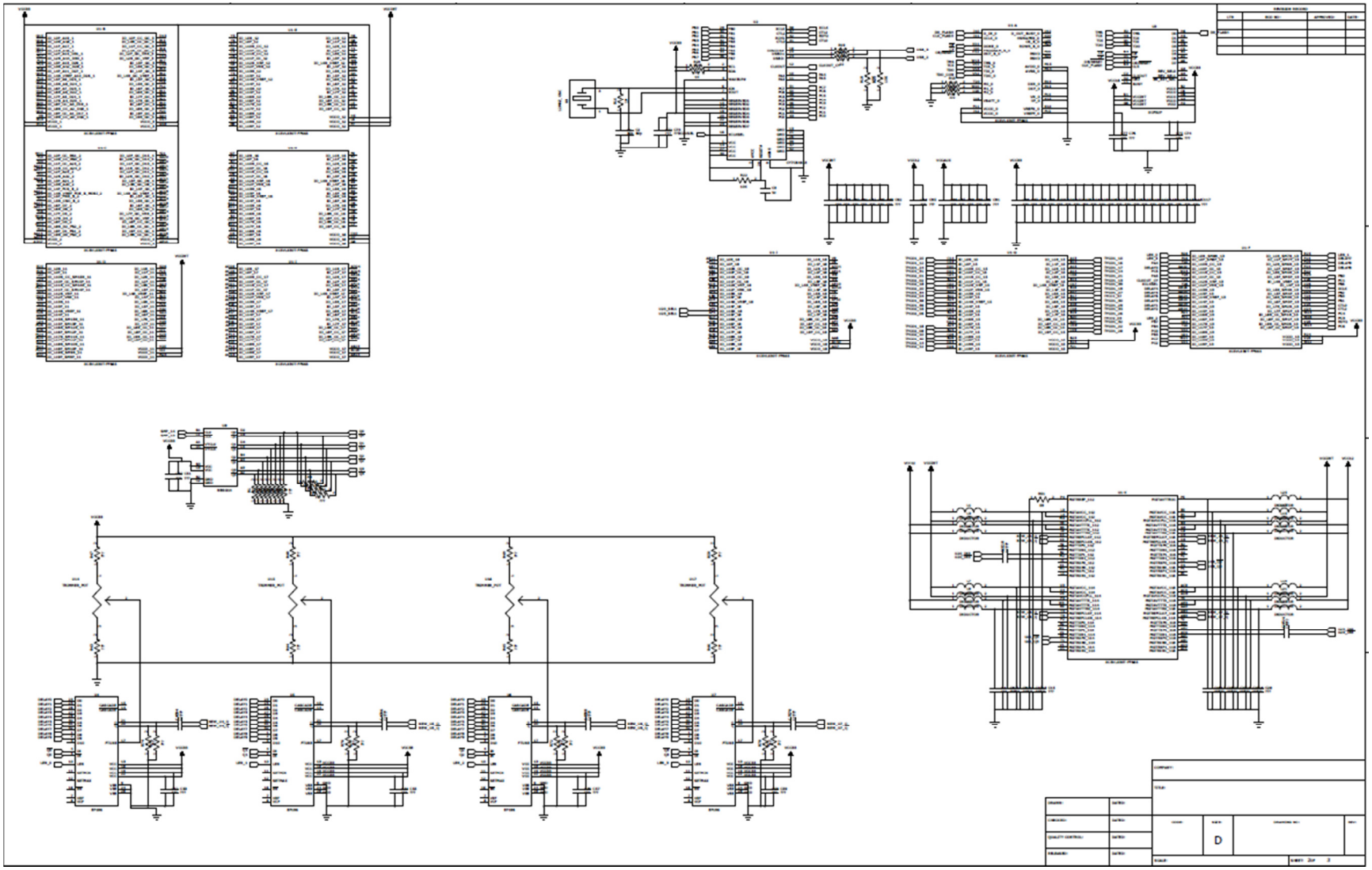


Figure C.1 Core logic block schematic

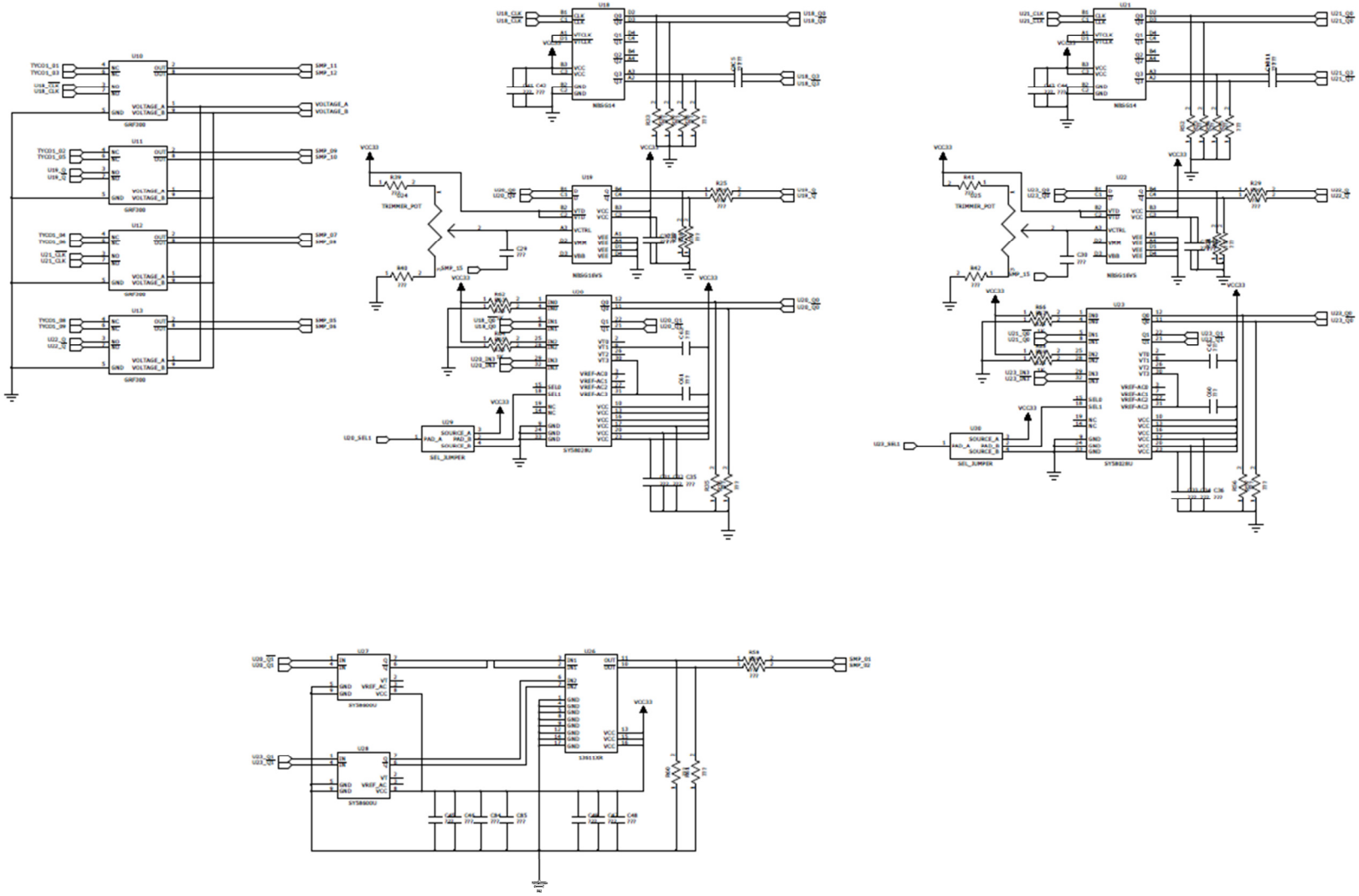


Figure C.2 Application specific logic schematic

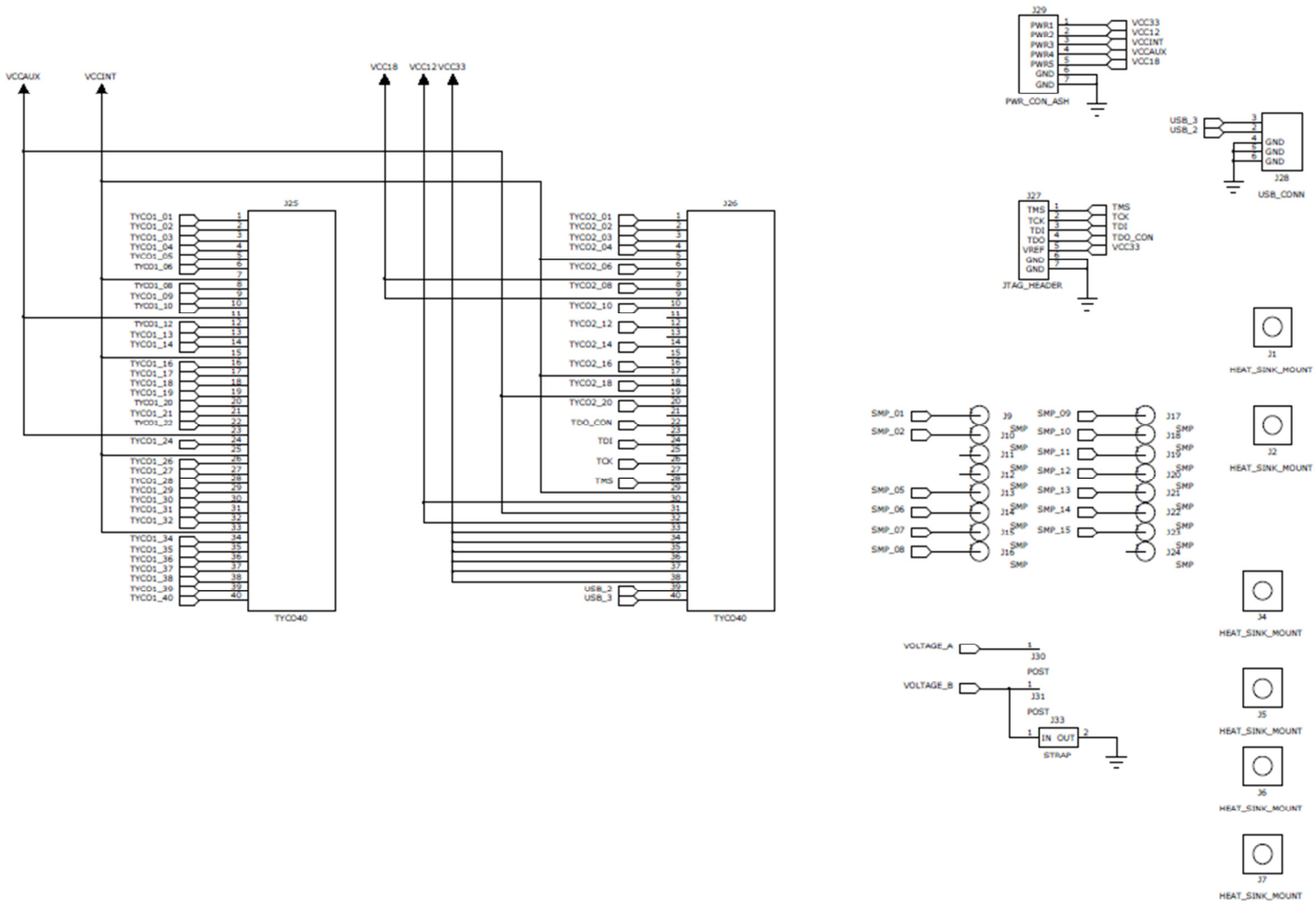


Figure C.3 Test Module connectors schematic

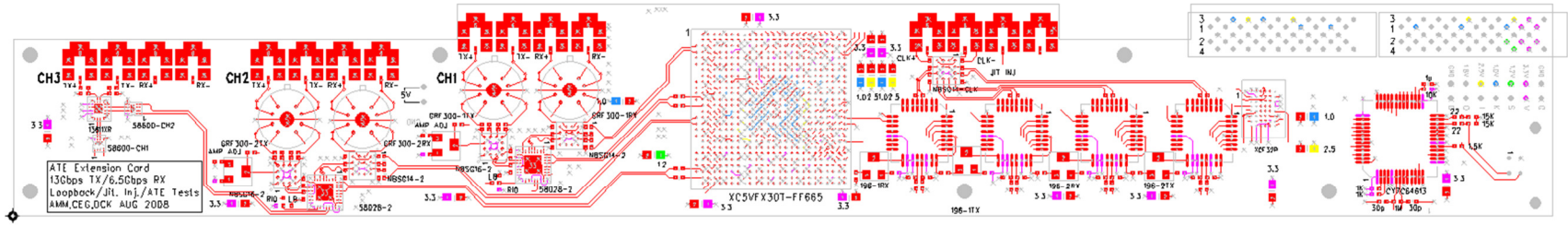


Figure C.4 Test Module Layer 1: Signal – Top

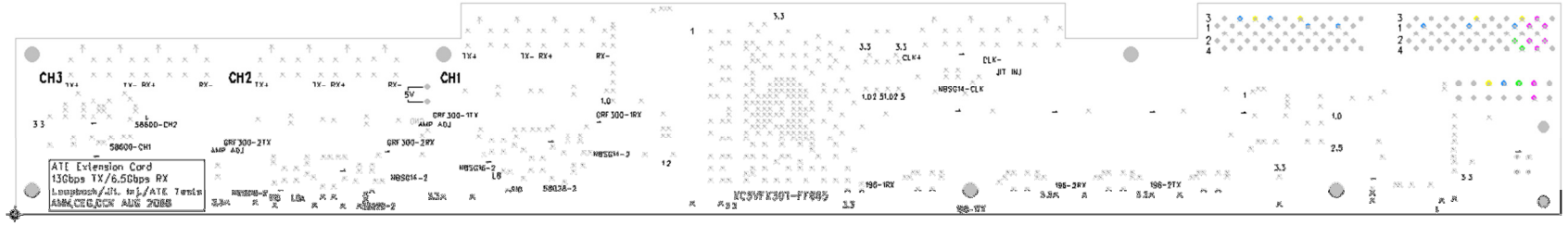


Figure C.5 Test Module Layer 2: Ground Plane - 1

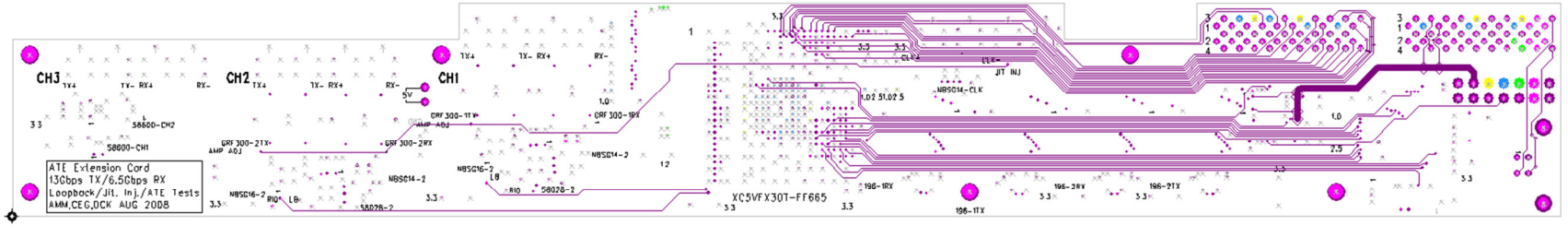


Figure C.6 Test Module Layer 3: Signal – Inner 1

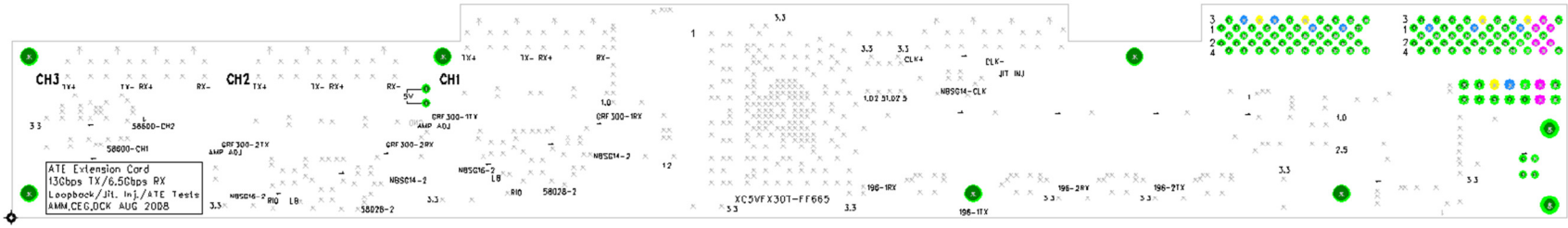


Figure C.7 Test Module Layer 4: Ground Plane - 2

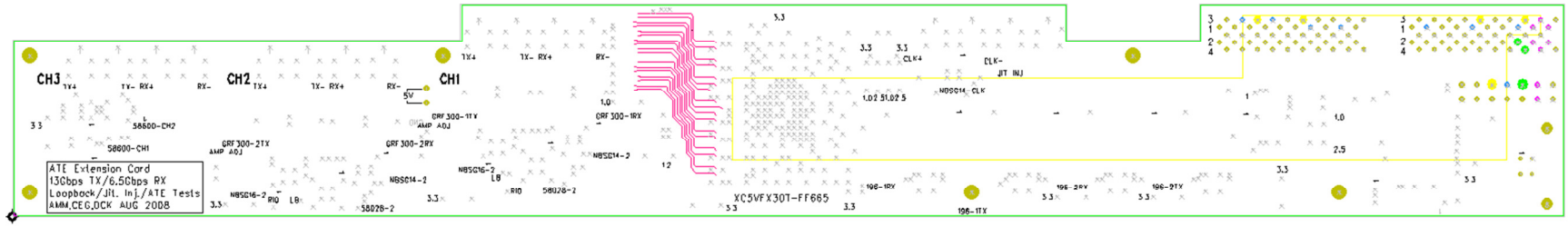


Figure C.8 Test Module Layer 5: Power Plane – 1

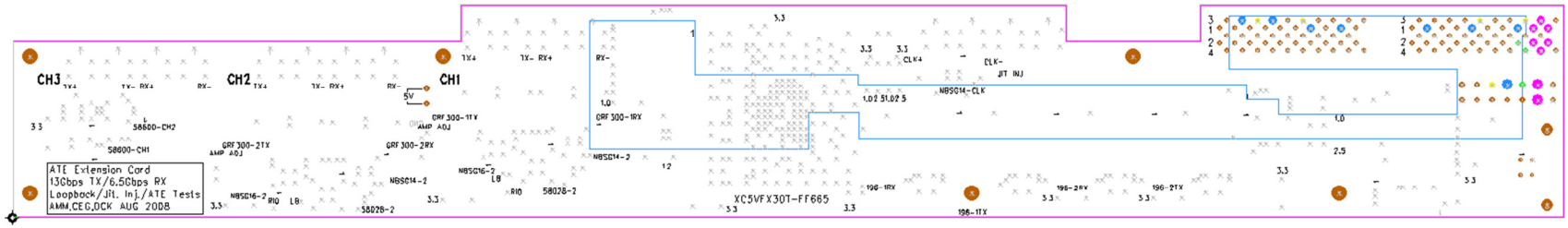


Figure C.9 Test Module Layer 6: Power Plane - 2

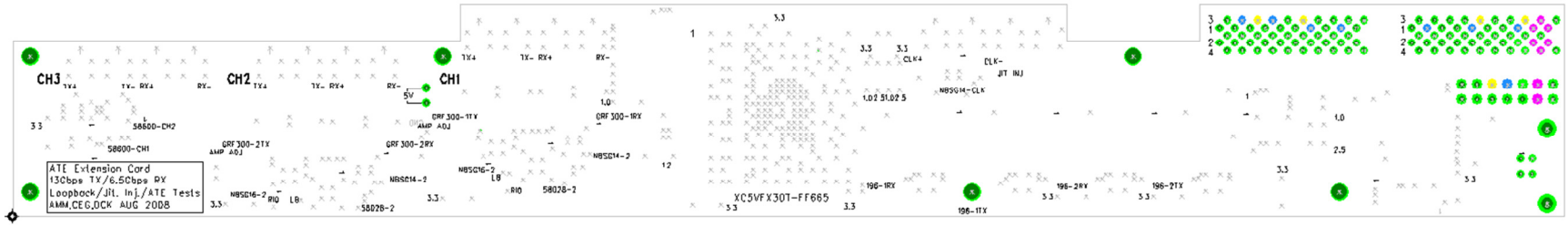


Figure C.10 Test Module Layer 7: Ground Plane - 3

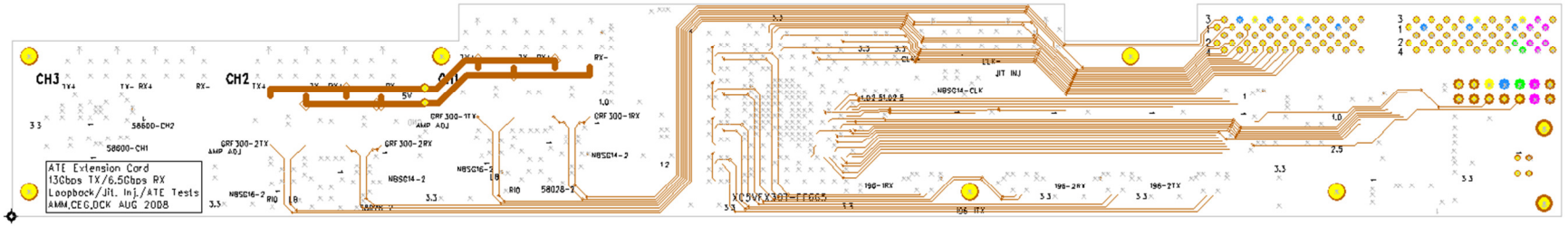


Figure C.11 Test Module Layer 8: Signal – Inner 2

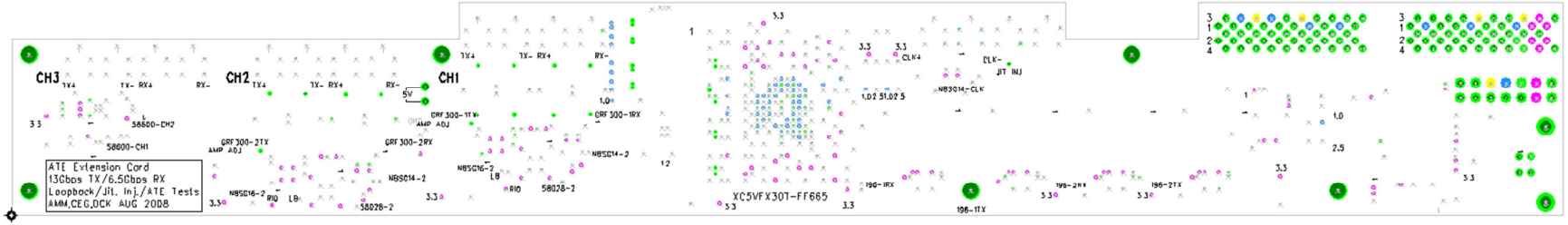


Figure C.12 Test Module Layer 9: Ground Plane - 4

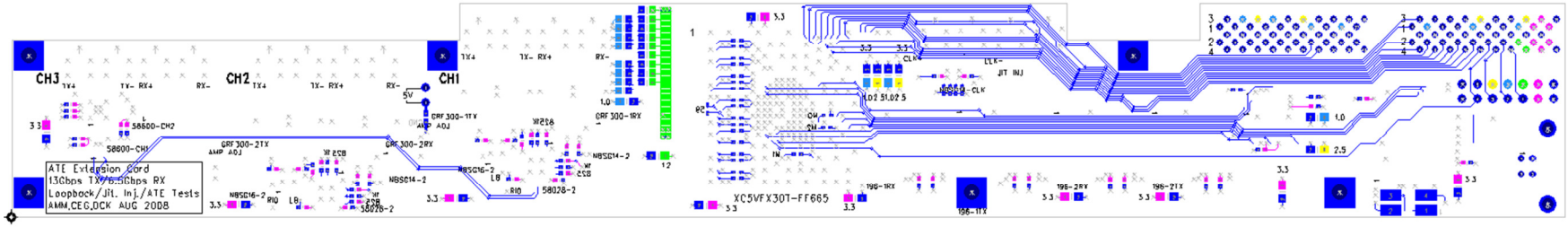


Figure C.13 Test Module Layer 10: Signal - Bottom

REFERENCES

[1]. International Technology Roadmap for Semiconductors 2005 Edition - *Test and Test Equipment*, The International Technology Roadmap for Semiconductors, 2005.

[2]. International Technology Roadmap for Semiconductors 2009 Edition - Test and Test Equipment, The International Technology Roadmap for Semiconductors, 2009.

[3]. Moore, G. E.; , "Cramming more components onto integrated circuits," *Electronics*, Volume 38, Number 8, April 19, 1965.

[4]. Kanellos, M.; , "New Life for Moore's Law," *News.com Special Report*, April 19, 2005, http://news.cnet.com/New-life-for-Moores-Law/2009-1006_3-5672485.html.

[5]. Verigy V93000 Series High-Speed I/O Test Solution – Solution Overview, Verigy Ltd., CA, 2006.

[6]. Credence Sapphire D-6408 Datasheet, Credence Systems Corporation, Milpitas CA, 2007.

[7]. Landis, D.L.; , "A Test Methodology for Wafer Scale Systems" *IEEE Trans. Computer-Aided Design*, Vol. 11, No. 1, pp.76-82, January 1992.

[8]. Butler, K.M.; , "Estimating the economic benefits of DFT," *Design & Test of Computers, IEEE* , vol.16, no.1, pp.71-79, Jan-Mar 1999.

[9]. Davidson, S.; , "Justifying DFT with a Hierarchical Top-Down Cost-Benefit Model," *Test Conference, 2008. ITC 2008. IEEE International* , vol., no., pp.1-10, 28-30 Oct. 2008.

[10]. Wei, S.; Nag, P.K.; Blanton, R.D.; Gattiker, A.; Maly, W.; , "To DFT or not to DFT?," *Test Conference, 1997. Proceedings., International* , vol., no., pp.557-566, 1-6 Nov 1997.

[11]. Jagiela, M.; , "The Dynamic World of Semiconductors and Implications for Test," *Korea Test Conference* , June 2007.

[12]. Lara, G.; , "Industry's Highest Bandwidth FPGA Enables World's First Single-FPGA Solution for 400G Communications Line Cards," WP385 v1.1, Xilinx, Inc., San Jose, CA, November 2010.

[13]. Furukawa, Y.; Rajsuman, R.; , "New trends drive ATE open architecture," *Semiconductor International*, Issue 8, July 2005

[14]. Grobart, S.; , “In Smartphone Era, Point-and-Shoots Stay Home,” *The New York Times*, December 3, 2010.

[15]. MacDonald, C.; , “Embedded Test to the Rescue: What can chip manufacturers do about soaring test cost and their impact on ATE?”, *Semiconductor Magazine*, Vol. 1, No. 11, November 2007.

[16]. Fisher, P.; Nesbitt, R.; , “Clock Cycle Estimate and Test Challenges for Future Microprocessor,” Sematech Technology Transfer #98033484A-TR.

[17]. Tummala, R. R.; Rymaszewski, E.J.; Klopfenstein, A.G.; , “Microelectronics Packaging Handbook,” Second Edition, Part II, Chapter 13, pp. 814-867, Klumber Academic Publisher, 1996.

[18]. Anon., “Nicholas DeWolf: The Father of ATE(Automatic Test Equipment),” https://www.chiphistory.org/legends/nick_dewolf/nick_dewolf.htm (accessed January 13, 2011).

[19]. Anon., “About Teradyne – A Brief History,” http://www.teradyne.com/corp/history.html (accessed January 13, 2011).

[20]. Bierman., H. ; “VLSI test gear keeps pace with chip advances,” *Electronics*, April 19, 1984.

[21]. Shahriari, N.; , "Mission possible? Open architecture ATE," *Test Conference, 2002. Proceedings. International*, vol., no., pp. 1206, 2002.

[22]. Evans, A.C.; , "The new ATE: Protocol aware," *Test Conference, 2007. ITC 2007. IEEE International*, vol., no., pp.1-10, 21-26 Oct. 2007.

[23]. Rivoir, J.; , "Protocol-Aware ATE enables cooperative test between DUT and ATE for improved TTM and test quality," *Test Conference, 2007. ITC 2007. IEEE International*, vol., no., pp.1-2, 21-26 Oct. 2007.

[24]. Sunter, S.; , "Protocol-aware ATE: Complement or competitor for structural testing?," *Test Conference, 2007. ITC 2007. IEEE International*, vol., no., pp.1, 21-26 Oct. 2007.

[25]. Williams, T.W.; Parker, K.P.; , "Design for Testability – A Survey," *The Proceedings of the IEEE*, vol. 71, no. 1, pp.98-112, Jan. 1983.

[26]. Agrawal, V. D.; Seth, S. C.; , “ Test Generation for VLSI Chips,” Computer Society Press, c1988.

[27]. Vermeulen, B.; Hora, C.; Kruseman, B.; Marinissen, E.J.; van Rijsinge, R.; , "Trends in testing integrated circuits," *Test Conference, 2004. Proceedings. ITC 2004. International,* vol., no., pp. 688- 697, 26-28 Oct. 2004.

[28]. Abramovici, M.; Breuer, M.A.; Driedman, A.A.; , "Digital Systems Testing and Testable Design," Computer Science Press, 1990.

[29]. IEEE Standard Test Access Port and Boundary-Scan Architecture-Description, IEEE Std 1149.1-2001, 2001.

[30]. Zorian, Y.; Marinissen, E.J.; Dey, S.; , "Testing embedded-core based system chips," *Test Conference, 1998. Proceedings., International* , vol., no., pp.130-143, 18-23 Oct 1998.

[31]. DaSilva, F.; Zorian, Y.; Whetsel, L.; Arabi, K.; Kapur, R.; , "Overview of the ieee P1500 standard," *Test Conference, 2003. Proceedings. ITC 2003. International* , vol.1, no., pp. 988- 997, Sept. 30-Oct. 2, 2003.

[32]. Dekker, R.; Beenker, F.; Thijssen, L.; , "Realistic built-in self-test for static RAMs," *Design & Test of Computers, IEEE* , vol.6, no.1, pp.26-34, Feb 1989.

[33]. Dorsch, J.; , "The Softer Side of Test: Software products to star at International Test Conference," *Electronic News*, Sep. 1999.

[34]. Bedsole, J.; Raina, R.; Crouch, A.; Abadir, M.S.; , "Very low cost testers: Opportunities and challenges," *Design & Test of Computers, IEEE* , vol.18, no.5, pp.60-69, Sep-Oct 2001.

[35]. Keezer, D.C.; , "Multiplexing test system channels for data rates above 1 Gb/s," *Test Conference, 1990. Proceedings., International* , vol., no., pp.362-368, 10-14 Sep 1990.

[36]. Milde, G.R.R.; , "Design and Electrical Characterization of Test Fixtures for High-speed Digital IC's", *International Test Conference 1987 Proceedings*, pp 363-369.

[37]. Henley, F.J.; Choi, H.J.; , "Test Head Design Using Electro-Optic Receivers and GaAs Pin Electronics for a Gigahertz Production Test System", *International Test Conference 1988 Proceedings*, pp 700-709.

[38]. Barton, S.; , "Characterization of High-speed (Above 500 MHZ) Devices Using Advanced ATE - Techniques,Results, and Device Problems", *International Test Conference 1989 Proceedings*, pp 860-868.

[39]. Henley, F.J.; Choi, H.J.; , "Achieving ATE Accuracy at Gigahertz Test Rates: Comparison of Electronic and Electro- Optic Sampling Technologies", *International Test Conference 1989 Proceedings*, p 953.

[40]. Bryson, S.W.; , "Custom Pin Electronics for VLSI Automatic Test Equipment", *International Test Conference 1989 Proceedings*, pp 854-859.

[41]. Tsai, S.J.; Hechtman, C.D.; "GaAs Driver and Sensor for a High Speed Test System", *International Test Conference 1988 Proceedings*, pp 13-22.

[42]. Kratzer, D.J.; Barton, S.; Henley, F.J.; Plomgrem, D.A.; , "High-speed Pattern Generator and GaAs Pin Electronics for a Gigahertz Production Test System", *International Test Conference 1988 Proceedings*, pp 710-718.

[43]. Hsue, C.W.; ,"Clock Signal Distribution Network for High Speed Testers", *International Test Conference 1989 Proceedings*, pp 199-207.

[44]. Keezer, D.C.; Minier, D.; Paradis, M.; Binette, L.; , "Modular extension of ATE to 5 Gbps," *Test Conference, 2004. Proceedings. ITC 2004. International* , vol., no., pp. 748- 757, 26-28 Oct. 2004.

[45]. Keezer, D.C.; Minier, D.; Caron, M.-C.; , "Multiplexing ATE channels for production testing at 2.5 Gbps," *Design & Test of Computers, IEEE* , vol.21, no.4, pp. 288- 301, July-Aug. 2004.

[46]. Meixner, A.; Kakizawa, A.; Provost, B.; Bedwani, S.; , "External Loopback Testing Experiences with High Speed Serial Interfaces," *Test Conference, 2008. ITC 2008. IEEE International* , vol., no., pp.1-10, 28-30 Oct. 2008.

[47]. Robertson, I.; Hetherington, G.; Leslie, T.; Parulkar, I.; Lesnikoski, R.; , "Testing high-speed, large scale implementation of SerDes I/Os on chips used in throughput computing systems," *Test Conference, 2005. Proceedings. ITC 2005. IEEE International* , vol., no., pp.8 pp.-999, 8-8 Nov. 2005.

[48]. Yamaguchi, T.J.; , "Multi-GHz interface devices should be tested using external test resources," *Test Conference, 2002. Proceedings. International* , vol., no., pp. 1229, 2002.

[49]. Fritzsche, W.A.; Haque, A.E.; , "Low cost testing of multi-GBit device pins with ATE assisted loopback instrument," *Test Conference, 2008. ITC 2008. IEEE International* , vol., no., pp.1-8, 28-30 Oct. 2008.

[50]. Sunter, S. *et al*; , “A digital BIST/BOST for parametric testing of 05~10 Gbps serial interface,” *Semicon Japan*, Dec. 2006 .

[51]. Sunter, S.; Roy, A.; , "A selt-testing BOST for high-frequency PLLs, DLLs, and SerDes," *Test Conference, 2007. ITC 2007. IEEE International* , vol., no., pp.1-8, 21-26 Oct. 2007.

[52]. Keezer, D.C.; Minier, D.; Ducharme, P.; Viens, D.; Flynn, G.; McKillop, J.S.; , "Multi-GHz loopback testing using MEMS switches and SiGe logic," *Test Conference, 2007. ITC 2007. IEEE International* , vol., no., pp.1-10, 21-26 Oct. 2007.

[53]. Keezer, D.C.; Minier, D.; Ducharme, P.; Majid, A.; , "An Electronic Module for 12.8 Gbps Multiplexing and Loopback Test," *Test Conference, 2008. ITC 2008. IEEE International* , vol., no., pp.1-9, 28-30 Oct. 2008.

[54]. Keezer, D.C.; Minier, D.; Ducharme, P.; , "Source-synchronous testing of multilane PCI Express and HyperTransport buses," *Design & Test of Computers, IEEE* , vol.23, no.1, pp. 46- 57, Jan.-Feb. 2006.

[55]. Borgioli, A.; Yu Liu; Nagra, A.S.; York, R.A.; , "Low-loss distributed MEMS phase shifter," *Microwave and Guided Wave Letters, IEEE* , vol.10, no.1, pp.7-9, Jan 2000.

[56]. Nagra, A.S.; York, R.A.; , "Distributed analog phase shifters with low insertion loss," *Microwave Theory and Techniques, IEEE Transactions on* , vol.47, no.9, pp.1705-1711, Sep 1999.

[57]. Barker, S.; Rebeiz, G.M.; , "Distributed MEMS true-time delay phase shifters and wide-band switches," *Microwave Theory and Techniques, IEEE Transactions on* , vol.46, no.11, pp.1881-1890, Nov 1998.

[58]. Zhang, H.; Laws, H.; Gupta, K.C.; Lee, Y.C.; Bright, V.M.; , "MEMS variable-Capacitor Phase shifters Part I: loaded-line phase shifter," Wiley Periodicals, Inc. Int RF and Microwave CAE 12:321-337, 2003.

[59]. Keezer, D.C.; Minier, D.; Ducharme, P.; , "Variable Delay of Multi-Gigahertz Digital Signals for Deskew and Jitter-Injection Test Applications," *Design, Automation and Test in Europe, 2008. DATE '08* , vol., no., pp.1486-1491, 10-14 March 2008.

[60]. Miller, C.M.; McQuate, D.J.; , "Jitter Analysis of High-Speed Digital Systems," *Hewlett-Packard Journal*, Vol. 46, No. 1, pp. 49-56, February 1995.

[61]. Keezer, D.C.; Minier, D.; Ducharme, P.; , "Method for Reducing Jitter in Multi-Gigahertz ATE," *Design, Automation & Test in Europe Conference & Exhibition, 2007. DATE '07* , vol., no., pp.1-6, 16-20 April 2007.

[62]. Cai, Y; Laquai, B.; Luehman, K.; , "Jitter testing for gigabit serial communication transceivers," *Design & Test of Computers, IEEE* , vol.19, no.1, pp.66-74, Jan/Feb 2002.

[63]. Laquai, B.; Yi Cai; , "Testing gigabit multilane SerDes interfaces with passive jitter injection filters," *Test Conference, 2001. Proceedings. International* , vol., no., pp.297-304, 2001.

[64]. Shitnanouchi, M.; , "Periodic jitter injection with direct time synthesis by SPP_{tm} ATE for serdes jitter tolerance test in production," *Test Conference, 2003. Proceedings. ITC 2003. International* , vol.1, no., pp. 48- 57, Sept. 30-Oct. 2, 2003.

[65]. Harsh, K.F.; Zhang, W.; Bright, V.M.; Lee, Y.C.; , "Flip-chip assembly for Si-based RF MEMS," *Micro Electro Mechanical Systems, 1999. MEMS '99. Twelfth IEEE International Conference on* , vol., no., pp.273-278, 17-21 Jan 1999.

[66]. Kamgaing, T.; Ichikawa, K.; Zeng, X. Y.; Hwang, K. P.; Min, Y.; Kubota, J.; , "Future Package Technologies for Wireless Communication Systems," Intel Technology Journal, Vol. 9, Issue 04, 2005.

[67]. Sudo, T.; Yoshii, A.; Tamama, T.; Narumi, N.; Sakagawa, Y.; "'ULTIMATE': A 500-MHZ VLSI Test System with High Timing Accuracy", *International Test Conference 1987, Proceedings*, pp 206-213.

[68]. Tummala, R.; , "Fundamentals of Microsystems Packaging," NewYork, McGraw-Hill, 2001.

[69]. Bakir, M.S.; Reed, H.A.; Thacker, H.D.; Patel, C.S.; Kohl, P.A.; Martin, K.P.; Meindl, J.D.; , "Sea of Leads (SoL) ultrahigh density wafer-level chip input/output interconnections for gigascale integration (GSI)," *Electron Devices, IEEE Transactions on* , vol.50, no.10, pp. 2039- 2048, Oct. 2003.

[70]. Keezer, D.C.; Patel, C.S.; Bakir, M.S.; Qing Zhou; Meindl, J.D.; , "Electrical test strategies for a wafer-level packaging technology," *Electronics Packaging Manufacturing, IEEE Transactions on* , vol.26, no.4, pp. 267- 272, Oct. 2003.

[71]. Majid, A. M.; Keezer, D.C.; Jayabalan, J.; Rotaru, M. R.; , "Multi-Gigahertz Testing of Wafer-Level Packaged Devices," *Test Conference, 2006. ITC '06. IEEE International* , vol., no., pp.1-10, Oct. 2006.

[72]. Rao, V.S.; Tay, A.A.O.; Kripesh, V.; Lim, C.T.; Seung Wook Yoon; , "Bed of nails - 100 microns pitch wafer level interconnections process," *Electronics Packaging Technology Conference, 2004. EPTC 2004. Proceedings of 6th* , vol., no., pp. 444- 449, 8-10 Dec. 2004.

[73]. Liao, E.B.; Ang, S.S.; Tay, A.A.O.; Feng, H.H.; Nagarajan, R.; Kripesh, V.; , "Fabrication and parametric study of wafer-level multiple-copper-column interconnect," *Electronic Components and Technology Conference, 2004. Proceedings. 54th* , vol.2, no., pp. 1251- 1255 Vol.2, 1-4 June 2004.

[74]. Chng, A.C.; Tay, A.A.O.; Lim, K.M.; Wong, E.H.; , "Fatigue life estimation of a stretched-solder-column ultra-fine-pitch wafer level package using the macro-micro modelling approach," *Electronic Components and Technology Conference, 2004. Proceedings. 54th* , vol.2, no., pp. 1586- 1591 Vol.2, 1-4 June 2004.

[75]. Patel, C.; Power, C.; Realff, M.; Kohl, P.; Martin, K.; Meindl, J.; “Low cost high density compliant wafer level package” *IEEE Int. Conf. on High-Density Interconnect and Systems Packaging*, pp. 335-339, 2000.

[76]. Patel, C.S.; Realff, M.; Merriweather, S.; Power, C.; Martin, K.; Meindl, J.D.; , "Cost analysis of compliant wafer level package," *Electronic Components and Technology Conference, 2000. 2000 Proceedings. 50th* , vol., no., pp.1634-1639, 2000.

[77]. Jayabalan, J.; Rotaru, M.D.; Rao, V.S.; Kripesh, V.; Iyer, M.K.; Tay, A.A.O.; Ban-Leong Ooi; Mook-Seng Leong; , "A Novel Test Strategy for Fine Pitch Wafer-Level Packaged Devices," *Advanced Packaging, IEEE Transactions on* , vol.30, no.3, pp.439-447, Aug. 2007.

[78]. Chun, D.; Ang, S.S.; Feng Hanhua; Tay, A.A.O.; Rotaru, M.D.; Keezer, D.; Tan, J.P.H.; , "A MEMS based interposer for nano-wafer level packaging test," *Electronics Packaging Technology, 2003 5th Conference (EPTC 2003)* , vol., no., pp. 405- 409, 10-12 Dec. 2003.

[79]. Lee, K.K.; Kim, B.C.; , "MEMS spring probe for next generation wafer level testing," *MEMS, NANO and Smart Systems, 2003. Proceedings. International Conference on* , vol., no., pp. 214- 217, 20-23 July 2003.

[80]. Sporck, N.; , "A new probe card technology using compliant Microsprings^{TM} ," *Test Conference, 1997. Proceedings., International* , vol., no., pp.527-532, 1-6 Nov 1997.

[81]. Novitsky, J.; Pedersen, D.; , "FormFactor introduces an integrated process for wafer-level packaging, burn-in test, and module level assembly," *Advanced Packaging Materials: Processes, Properties and Interfaces, 1999. Proceedings. International Symposium on*, vol., no., pp.226-231, 14-17 Mar 1999.

[82]. Keezer, D.C.; Davis, J.S.; Ang, S.; Rotaru, M.; , "A test strategy for nanoscale wafer level packaged circuits," *Electronics Packaging Technology Conference, 2002. 4th* , vol., no., pp. 175- 179, 10-12 Dec. 2002.

[83]. Chandra, S.J.; Patel, J.H.; , "Test generation in a parallel processing environment," *Computer Design: VLSI in Computers and Processors, 1988. ICCD '88., Proceedings of the 1988 IEEE International Conference on* , vol., no., pp.11-14, 3-5 Oct 1988.

[84]. Lautner, M.; Bieser, G.; , "Probing ASIC Devices" *Solid State Technology,* pp. 111-114, Aug. 1986.

[85]. Mielke, J.A.; Pope, K.A.; , "High-speed fixture interconnects for mixed-signal IC testing," *Test Conference, 1990. Proceedings., International* , vol., no., pp.891-895, 10-14 Sep 1990.

[86]. Majid, A.M.; Keezer, D.C.; Taher, N.; Gray, C.; Ahmad, J.; , "Performance characteristics of a 5 Gbps functional test module," *Electronics Packaging Technology Conference, 2004. EPTC 2004. Proceedings of 6th*, vol., no., pp. 244- 248, 8-10 Dec. 2004.

[87]. Keezer, D.C.; Gray, C.; Majid, A.; Taher, N.; , "Low-cost multi-gigahertz test systems using CMOS FPGAs and PECL," *Design, Automation and Test in Europe, 2005. Proceedings*, vol., no., pp. 152- 157 Vol. 1, 7-11 March 2005.

[88]. Majid, A.M.; Keezer, D.C.; , "An improved low-cost 6.4 Gbps wafer-level tester," *Electronic Packaging Technology Conference, 2005. EPTC 2005. Proceedings of 7th* , vol.2, no., pp.6 pp., 7-9 Dec. 2005.

[89]. Verigy V93000 Pin Scale HX high-speed extension card – Product Overview (5989-5119EN). Verigy Ltd., Cupertino, CA, 2006.

[90]. Credence Saphire D-6408 – Datasheet (PD12207). Credence Systems Corporation, Milpitas CA, 2007.

[91]. Virtex-5 FPGA RocketIO GTX Transceiver User Guide, UG198 v2.1, Xilinx, Inc., San Jose, CA, 2007.

[92]. Moreira, J.; Barnes, H.; Kaga, H.; Comai, M.; Roth, B.; Culver, M.; , "Beyond 10 Gbps? Challenges of Characterizing Future I/O Interfaces with Automated Test Equipment," *Test Conference, 2008. ITC 2008. IEEE International*, vol., no., pp.1-10, 28-30 Oct. 2008.

[93]. Fritzsche, W.A.; Haque, A.E.; , "Low cost testing of multi-GBit device pins with ATE assisted loopback instrument," *Test Conference, 2008. ITC 2008. IEEE International*, vol., no., pp.1-8, 28-30 Oct. 2008.

[94]. Anon., “Advanced Digital with Silicon Germanium Technology,” http://www.teradyne.com/tiger/digital.html (accessed January 21, 2011).

[95]. 7 Series FPGAs Overview *Advance Product Specification*, DS180 v1.3, Xilinx, Inc., San Jose, CA, October 2010.

[96]. Keezer, D.; Gray, C.; Majid, A.; Taher, N.; , "Implementing multi-gigahertz test systems using CMOS FPGAs and PECL components," *Solid-State Circuits Conference, 2005. ESSCIRC 2005. Proceedings of the 31st European*, vol., no., pp. 291- 294, 12-16 Sept. 2005.

[97]. Majid, A.M.; Keezer, D.C.; Karia, J.V.; , "A 5 Gbps Wafer-Level Tester," *Test Symposium, 2005. Proceedings. 14th Asian*, vol., no., pp. 58- 63, 18-21 Dec. 2005.

[98]. Keezer, D.C.; Gray, C.; Majid, A.; Minier, D.; Ducharme, P.; , "A development platform and electronic modules for automated test up to 20 Gbps," *Test Conference, 2009. ITC 2009. International*, vol., no., pp.1-11, 1-6 Nov. 2009.

[99]. Keezer, D.C.; Minier, D.; Caron, M.C.; , "A production-oriented multiplexing system for testing above 2.5 gbps," *Test Conference, 2003. Proceedings. ITC 2003. International* , vol.1, no., pp. 191- 200, Sept. 30-Oct. 2, 2003.

[100]. Virtex-5 Family Overview *Advance Product Specification*, DS100 v4.3, Xilinx, Inc., San Jose, CA, June 2008.

[101]. Virtex-5 FPGA User Guide, UG190 v4.2, Xilinx, Inc., San Jose, CA, May 2008.

[102]. Virtex-5 FPGA RocketIO GTP Transceiver *User Guide*, UG196 v1.6, Xilinx, Inc., San Jose, CA, February 2008.

[103]. Embedded Processor Block in Virtex-5 FPGAs *Reference Guide*, UG200 v1.8, Xilinx, Inc., San Jose, CA, February 2010.

[104]. IEEE Standard Test Access Port and Boundary-Scan Architecture-Description, IEEE Std 1149.1-2001, 2001.

[105]. Platform Flash PROM User Guide, UG161 v1.6, Xilinx, Inc., San Jose, CA, October 2009.

[106]. Universal Serial Bus Specification, Compaq, Hewlett-Packard, Intel, Lucent, Microsoft, NEC, Philips, Revision 2.0, April 2000.

[107]. Universal Serial Bus 3.0 Specification, Hewlett-Packard, Intel, Microsoft, NEC, ST-NXP Wireless, Texas Instruments, Revision 1.0, November 2008.

[108]. “Universal Serial Bus.” *Wikipedia: The Free Encyclopedia*. Wikimedia Foundation, Inc. 11 November 2010, 9 November 2010.

[109]. “USB Class Codes.” *USB Implementers Forum, Inc.*. 11 November 2010, 17 November 2009.

[110]. EZ-USB FX^{TM} USB Microcontroller, Document #:38-08005, Cypress Semiconductor Corporation, San Jose, CA, April 2003.

[111]. Davis, J. S.; , “An FPGA-Based Digital Logic Core for ATE Support and Embedded Test Applications,” Doctoral dissertation, ECE, Georgia Institute of Technology, Atlanta, GA, July, 2003.

[112]. RAMB18 *Primitive: 18K-bit Configurable Synchronous True Dual Port Block RAM*, Xilinx, Inc., San Jose, CA, 2008.

[113]. Johnson, H.; Graham, M.; , "*High-Speed Digital Design: A Handbook of Black Magic,*" Prentice Hall PTR, New Jersey, 1993.

[114]. Caldwell, B.; Getty, D.; , "Coping with SCSI at Gigahertz Speeds," EDN, July 6, 2000.

[115]. Demarest, K.; , "Engineering Electromagnetics," Prentice Hall, New Jersey, 1997.

[116]. Rogers Corporation, "Introducing R04003C and RO435B Laminates with TICER TCR Thin Film Resistor Foils - Preliminary Data Sheet," Publication #92-134, July 2008.

[117]. On Semiconductor, 2.5V/3.3V SiGe Differential 1:4 Clock/Data Driver with RSECL Outputs, NBSG14/D, Rev. 15, August 2009.

[118]. Inphi Corporation, 13611XR 13 Gbps XOR/XNR Data Sheet, Ver. 2.2, June, 2007.

[119]. On Semiconductor, 2.5V/3.3V SiGe Differential Receiver/Driver Variable Output Swing, NBSG16VS/D, Rev. 12, September, 2008.

[120]. On Semiconductor, 3.3V ECL Programmable Delay Chip, MC10EP195/D, Rev. 18, April, 2009.

[121]. White paper; , "*Fulfilling Needs for 40G-100G Network-Centric Operations and Warfare,*" WP-01138-1.1, Altera Corporation, San Jose, CA, September 2010.

[122]. Inphi Corporation, 25711XR 25 Gbps XOR/XNOR Data Sheet, Ver. 3.1, May, 2007.

[123]. Hyde, J.; , "*USB Design-by-example: a practical guide to building IO devices,*" John Wiley & Sons, New York, 1999.

VITA

Ashraf Muhammad Majid was born on July 26th 1979, in Dhaka Bangladesh. He received the B.S. degree (with high honors) in computer engineering from the Georgia Institute of Technology (Georgia Tech) in 2003. He continued on to receive the M.S. degree in electrical and computer engineering, and the M.S. degree in quantitative and computational finance in 2005 and 2007, respectively.

In the Spring of 2004, Dr. Majid joined the High-Speed Digital Test Lab at Georgia Tech under the guidance of Dr. David Keezer in order to pursue the Ph.D. degree in electrical and computer engineer. His primary areas of expertise are high-speed digital test, high-performance electronic systems design, and wafer-level probing. He has over seven years of experience in designing high-speed digital test systems. He has published numerous refereed and invited papers and received the 2006 Top-Ten International Test Conference Paper award. In the Spring of 2011, Dr. Majid received the Ph.D. degree in electrical and computer engineering from Georgia Tech.

Dr. Majid is currently the Vice President of Market Risk Management at SunTrust Banks, Inc. in Atlanta. His current role involves the modeling and risk management of financial derivatives. Prior to joining SunTrust, Dr. Majid worked as a test engineer at Nokia Corp. in Atlanta.