RAMP BEE3 Interchip Link

by Tracy Wang

Research Project

Submitted to the Department of Electrical Engineering and Computer Sciences, University of California at Berkeley, in partial satisfaction of the requirements for the degree of **Master of Science, Plan II**.

Approval for the Report and Comprehensive Examination:

Committee:

Professor John Wawrzynek
Research Advisor

(Date)

* * * * * * *

Professor Krste Asanovic
Second Reader

(Date)

5/22/08

Contents

List of Figures

1. Abstract

In this report we investigate high-speed and reliable FPGA to FPGA link implementations on the BEE3 board. Using the Xilinx Virtex-5 FPGAs, we implement a source synchronous link and analyze its performance and reliability at various data rates. To ensure reliability for all data patterns, we create a "forced inversion" module that modulates the data to limit its spectral content. We show that our implementation runs reliably up to a data rate of 966Mbps per wire.

2. Introduction

The Research Accelerator for Multiple Processors project (RAMP) aims to enable research on multi-core systems by creating multi-core simulators built on FPGAs [2]. These systems reside on multiple FPGA chips across multiple boards, so it is crucial that FPGAs can communicate effectively and reliability with each other through an interchip module. Hence, the goal of this project is to investigate high speed and reliable interchip implementations on the Berkeley Emulation Engine 3 board.

The Berkeley Emulation Engine (BEE) is a hardware platform built from FPGAs. There have been several revisions of the BEE boards; the BEE2 has been instrumental in several RAMP projects including the RAMP Blue [7]; the BEE3 is the newest board.

We began this project before the first BEE3 board became available, so we started by building a testing infrastructure for the existing interchip link on the BEE2. When the BEE3 board finally arrived, we shifted focus to creating a high speed source synchronous interchip module for the BEE3. We will first give a brief background on source synchronous links, then summarize our work for the BEE2 interchip, and finally segue into topics concerning the BEE3 interchip interface module.

2.1 Source Synchronous Links

In digital communication, when a transmitter chip sends data to a receiver chip, the receiver must sample the data with some clock source. One option is generate a clock on the board and distributed it to both chips; we call this board synchronous. Board synchronous implementation requires slower operating speed so that the clocks can be in phase with each other [3].

Source synchronous links can run at a higher operating speed by ensuring that the receiver is using the same clock as the transmitter when sampling incoming data and eliminating unnecessary routing delays [4]. The transmitter sends both the clock and data and the receiver samples the incoming data relative to the incoming clock. Since the clock and data are sent on the same bus, they are subjected to the same skewing conditions. Figure 1 shows the difference between board synchronous and source synchronous implementation.

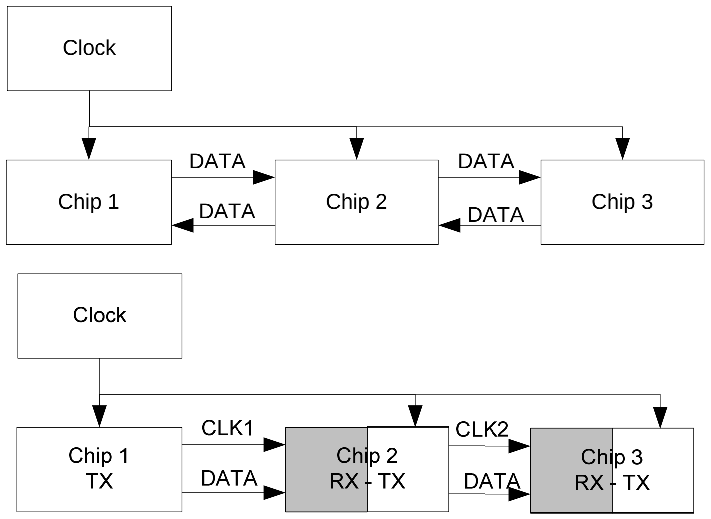


Figure 1: Top shows the board synchronous communication; bottom shows the source synchronous implementation.

Even though source synchronous links can run faster, there are still a few disadvantages compared to the alternatives. First, the implementation is more complex and resource intensive, requiring additional logic for link calibration and sampling adjustments. Second, the receiver must run on two clock domains: one driven by the source clock, and another by the receiver global clock, shown in Figure 1 by the shaded region on Device 2. This means that additional logic is required to propagate the data across the clock domain, increasing the complexity as well as room for error.

Although there are different implementations concerning the clock source, there are a few other concerns with source synchronous digital communication. The spectral content of the data affects the error rate during transmission [6]. Data patterns with an uneven frequency response stress the link's frequency response and signal integrity degrades as a result. Pulse shaping is a common technique used in digital communication to control the spectral content of the data through filters. This problem must be addressed to enable high speed links.

2.2 BEE2

BEE stands for Berkeley Emulation Engine. It is the hardware platform built from FPGAs. There have been several revisions of the BEE boards; the BEE2 has been instrumental in several RAMP projects including the RAMP Blue [7]; the BEE3 is the newest board with various upgrades from the BEE2.

As the hardware is upgraded on the BEE3, so have the expectations of the modules running on top of the hardware. The BEE2 interchip had been developed and used when we started the project. As the BEE3 was still in production, we used the BEE2 as a starting point for the project. We will first describe the BEE2 board then talk about its interchip and the testing framework.

The BEE2 board has five Virtex2Pro FPGAs: one control FPGA and four user FPGAs. As shown in Figure 2, the user FPGAs are connected in a circular fashion, while the control FPGA is connected to each user FPGA in a star shape. A simple kernel runs on top of the control FPGA to interact with the user FPGAs.

We started the project with a RAMP Description Language file. The RAMP Description Language (RDL) is developed to describe hardware based simulators [4]. RDLC is the java-based compiler which takes an RDL file as an input and can be configured to output the Verilog project and bit files. The focus of RDL is standardization and parameterization of complex distributed simulators. In our case, we used an RDL file that specifies a counter unit and modified this file to easily tweak our test setup.

The BEE2 interchip is a board synchronous link that uses a few control signals to communicate. Figure 3 shows this implementation: 32 pins are used to send 32 bits of data, and one pin sends control signals back from the receiver to let the transmitter know that it is ready to receive again. The rounded boxes show that the transmitter and receiver reside on two chips on the same board. Our tests run the interchip at 100MHz.

Figure 4 shows the setup of the testing application on the BEE2. The Selectmap interfaces are used to communicate with the transmitter and receiver chips. The user can send bits into the transmitter Selectmap to start or stop the tests, and print the output from the receiver Selectmap. The Selectmap interface uses a generated asynchronous FIFO connected to the Selectmap pins on the chip. Using the write and read enable control signals, the testing application can specify when to read and write from this asynchronous FIFO. The figure also shows the RDL generated portions of the test.

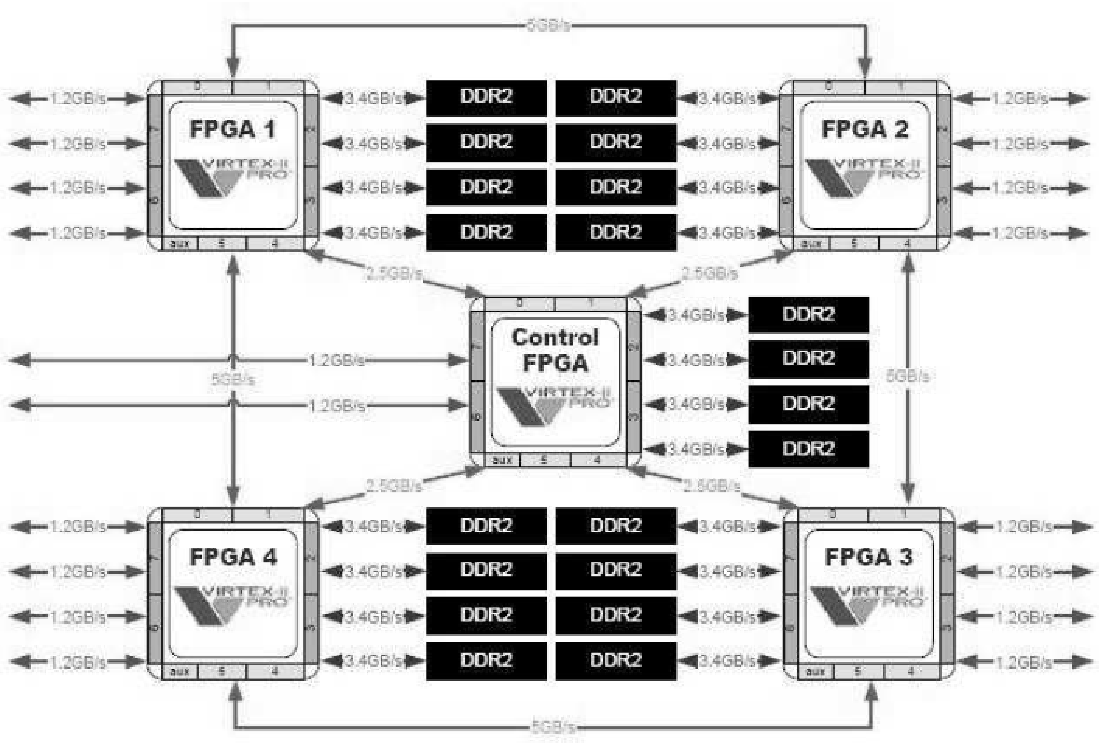


Figure 2: BEE2 Architecture [1].

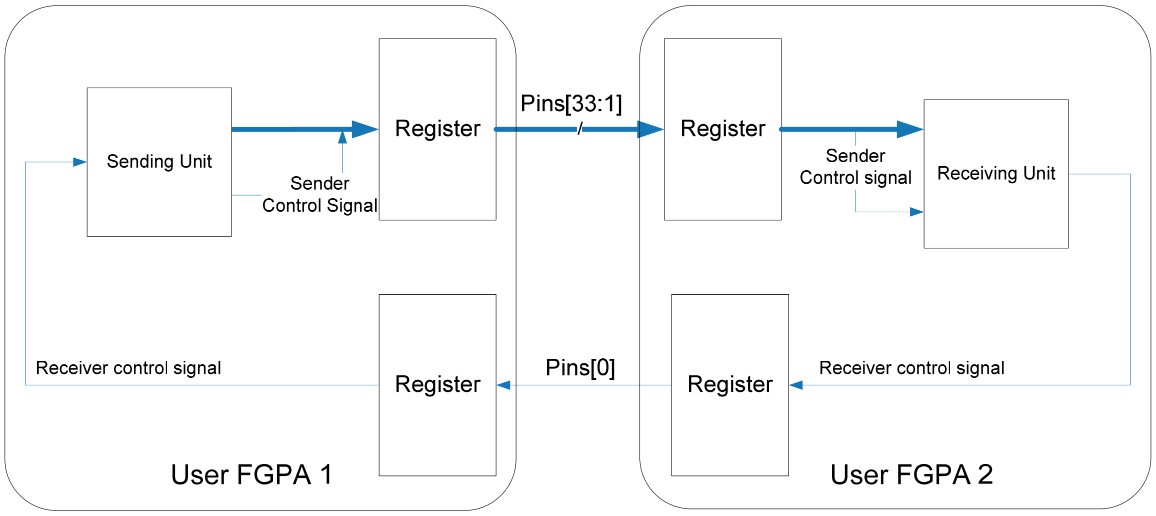


Figure 3: The BEE2 interchip implementation.

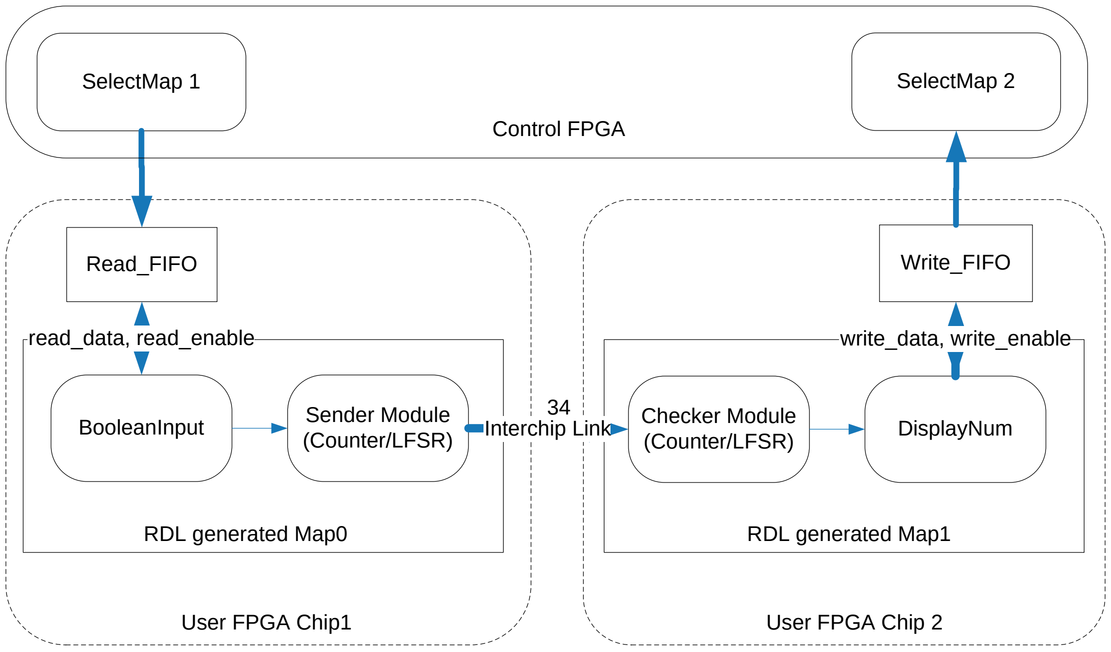


Figure 4: The test framework on the BEE2.

The transmitter instantiates a pattern generator; the Selectmap will alert the transmitter to send a new pattern every time a keyboard input is detected. In our testing framework, this pattern generator can be a counter or a pseudo-random number generator implemented with an LFSR. The receiver module has the same pattern generator instance which checks its next pattern against the received pattern to confirm accuracy. Because the transmitter waits for the Selectmap interface before sending new data, this framework runs only as fast as the Selectmap interface and does not stress the performance. We were unable to make additional improvements to this framework before moving onto the BEE3 board, but we preserved most of the high-level architecture in implementing the testing framework for the BEE3. However, we stopped using RDL and RDLC as we began working on the BEE3. We felt that designing a functional interchip itself was more important for a first step.

2.3 BEE3

The BEE3 board is shown in Figure 5. There are four FPGAs underneath the heat sinks. These are Virtex 5 LX110T speed grade -2 FPGAs. The QSH connectors are labeled in the figure; there is one per chip. A blue connector cable (not shown) connects any two connectors. Each chip is connected to five green LEDs. The BEE3 board used in this project has a default global clock running at 100MHz. This clock can be managed using a DCM to produce a clock signal running up to 500MHz. The operating temperature range of a commercial grade Virtex 5 is 0°C to 80°C [8].

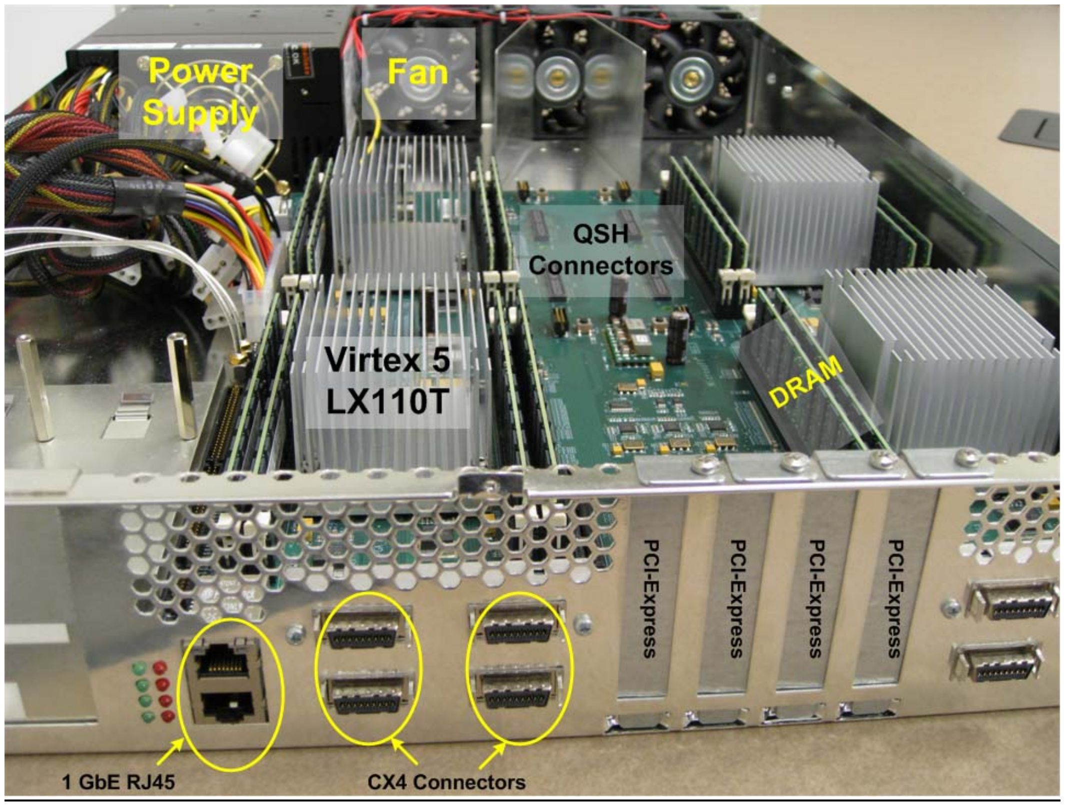


Figure 5: The BEE3 board.

The QSH connectors consist of 19 differential pairs specified to run at a maximum data rate of 1Gbps. Three of the 19 pairs are routed to clock capable pins, and all 19 pairs are on the same local bank. This setup allows clock to be sent through those pins and be used to drive the data lines, which is necessary for source synchronous communication.

2.4 Tools

We use the Xilinx Project Navigator for development and debugging for the BEE3 interchip interface module. XST is used for synthesis and Impact for programming. The PAR Timing Analyzer, FPGA floor planner and Chipscope are frequently used for debugging.

We chose these tools since we are more experienced with direct programming of FPGA with Verilog. Additionally, we felt that using the alternative, Xilinx EDK, would be unnecessarily cumbersome on for developing a small interchip module.

3. Design

The BEE3 interchip link uses a source synchronous design in order to optimize for speed and reliability. This implementation uses the QSH connector, which is wide enough to support bidirectional transmission across the cable. This section will describe the design of the link, the testing framework used to verify and stress the link, and specify how the link can be used in other applications.

3.1 Overview

The source synchronous design follows the one described in the Xilinx application note titled "16-Channel LVDS DDR Interface" [6]. The module sends 16 bits in parallel with a burst length of 8, for a total of 128 bits per burst; each burst takes 4 clock cycles at DDR. As shown in Figure 14, data is latched with a divided clock and sent through 16 OSERDES master/slave pairs to produce 16 differential signals pairs. The transmitter clock is also split into a differential pair with a dual data rate register and sent across the cable along with the data.

The receiver samples each incoming signal pair relative to the incoming clock pair then feeds the sampled bits into ISERDES master/slave pairs to de-serialize at double data rate. Calibration is necessary immediately after reset to ensure that the receiver samples each differential signal accurately relative to the clock. The bit alignment and resource sharing modules shown in Figure 7 are responsible for this initial calibration.

The Xilinx design notes that the link degrades when data is at direct current such that its frequency response is not flat; the application note claims the link can support data of up to 29 consecutive 0's or 1's before link is rendered unreliable at ~900Mbps for a speed grade -2 device [6]. Since this link would be used to send raw data, there is no limit to the spectral content. In order to ensure reliability, we have designed a "forced inversion" module which modulates the data to limit the spectral content of any data set. This module resides on both the transmitter and receiver to ensure that data is inverted back to the original after crossing the wire.

The testing framework controls the transmission rate and captures the error rate at the receiver. The pattern generator module is instantiated in both the transmitter and receiver; the generator at the transmitter uses the divided clock and creates a new pattern every four cycles ensuring new data for each burst; the receiver side generator creates a new pattern every time a new data arrives from the transmitter, and compares the values to determine error rate.

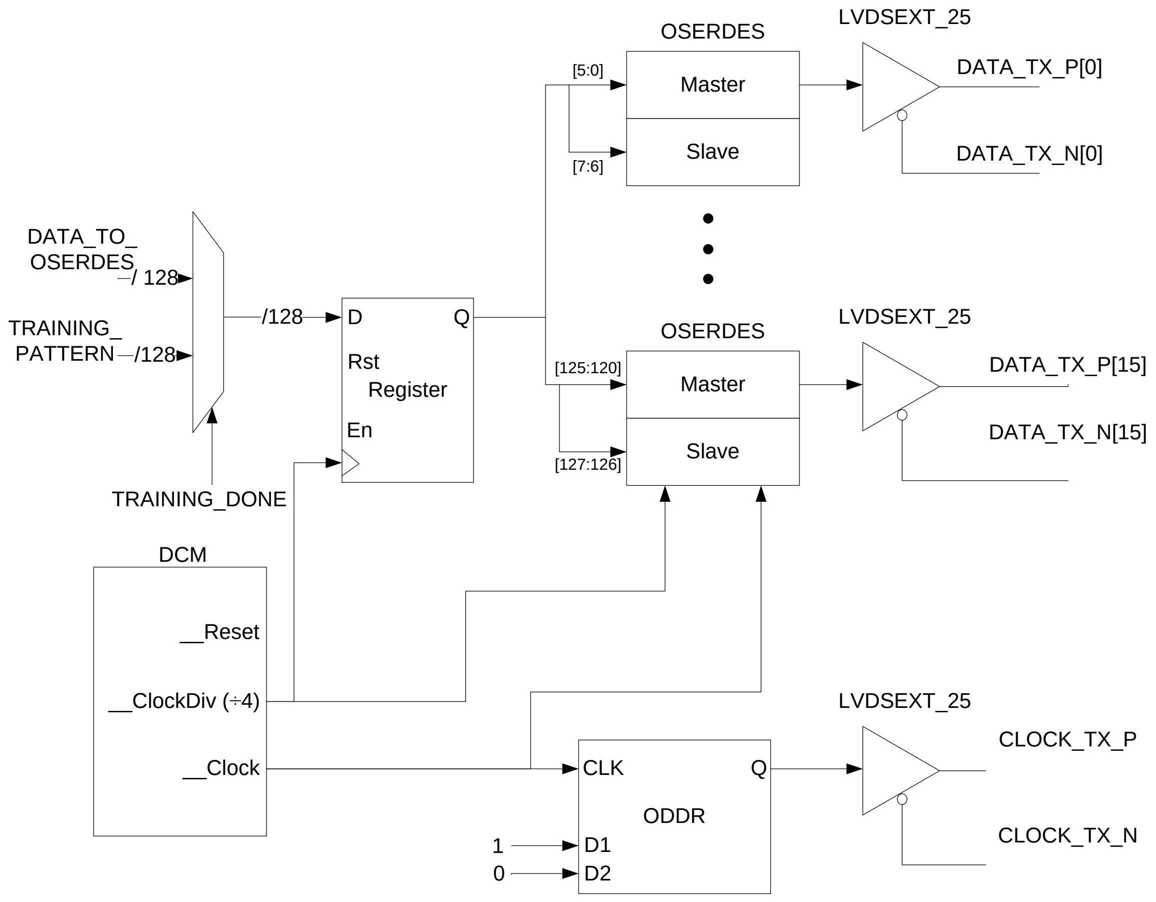


Figure 6: Source synchronous transmitter module

Figure 14 shows the details of the entire application including the testing framework, the interchip modules and the forced inversion module. The sections below will describe each section in detail.

3.2 OSERDES & ISERDES

The Virtex 5 OSERDES modules [8] serialize 2:1, 4:1 and 6:1 at single or double data rate. Since each OSERDES can serialize up to 6:1, two OSERDES are necessary to serialize 8 bits per differential pair. Sending 8 bits with double data rate takes four cycles, hence the transmitter latches new data using a divided by four clock, using the next four regular clock cycles to send the latched data. Also note that a 8:1 OSERDES module has a four cycle delay before data appears on the line, see Figure 8.

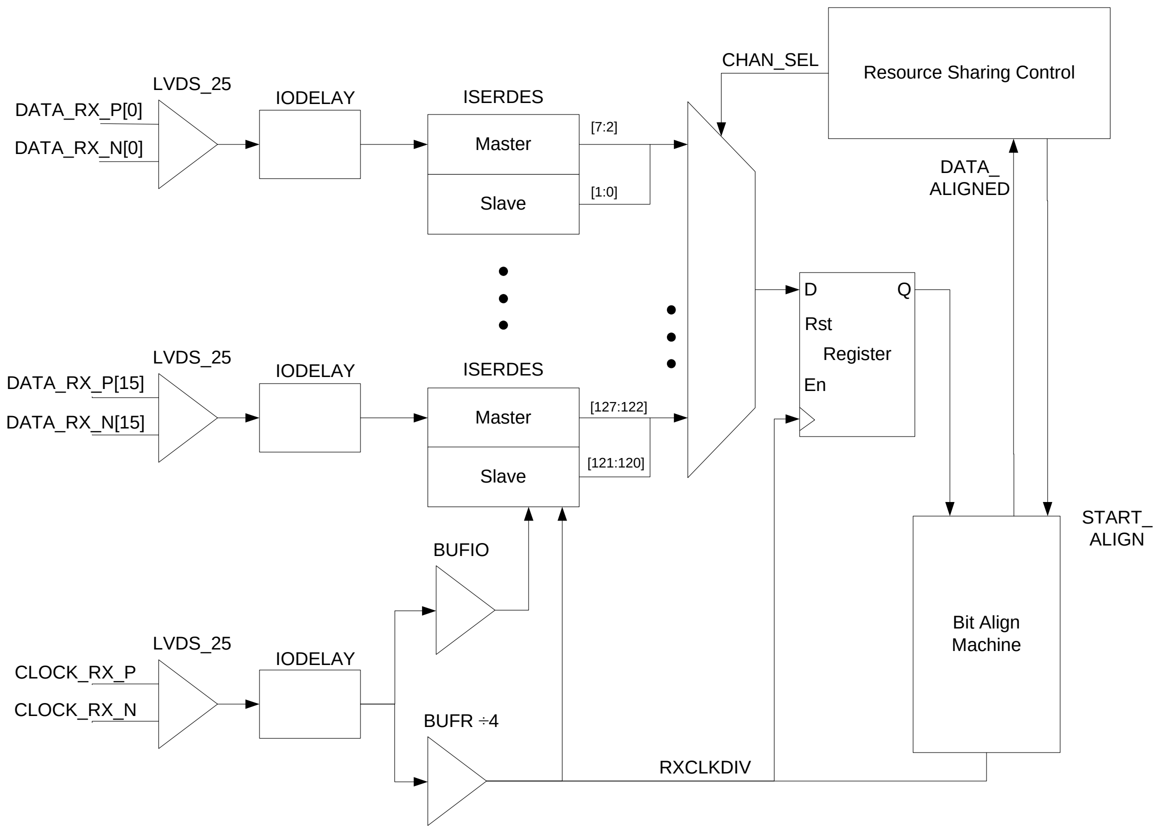


Figure 7: Source synchronous receiver module

The ISERDES is a de-serializer module designed for source synchronous applications [8]. Similar to the OSERDES, each ISERDES module can handle up to 6 bits, so two modules are necessary to de-serialize the incoming bits.

For more information on OSERDES and ISERDES, please see the Virtex 5 User Guide [8].

3.3 IODELAY

These modules control the delay of the incoming signal relative to the clock. Hence, these modules are tuned during calibration. These modules require a reference clock at 200MHz.

3.4 Interface Clocking

Two clocking schemes are possible in this design, one uses a global scheme with DCMs, and the other uses local clock buffers BUFIO and BUFR to drive the circuit, both shown in Figure 9.

The Xilinx app note suggests using the local clocking schemes to run both the transmitter and receiver to optimize performance. During implementation, we find that the transmitter must use a DCM with the global clock pins to generate a clock to drive the circuit since the BUFIO/BUFR can only be routed to local clock capable pins. The receiver, on the other hand, must route the incoming differential clock signals to clock capable I/O pins, and use BUFIO/BUFR to drive the interchip module. If the receiver uses a DCM to manage the incoming clocks, the routing will cause the module to fail during calibration at higher frequencies of 300MHz or more.

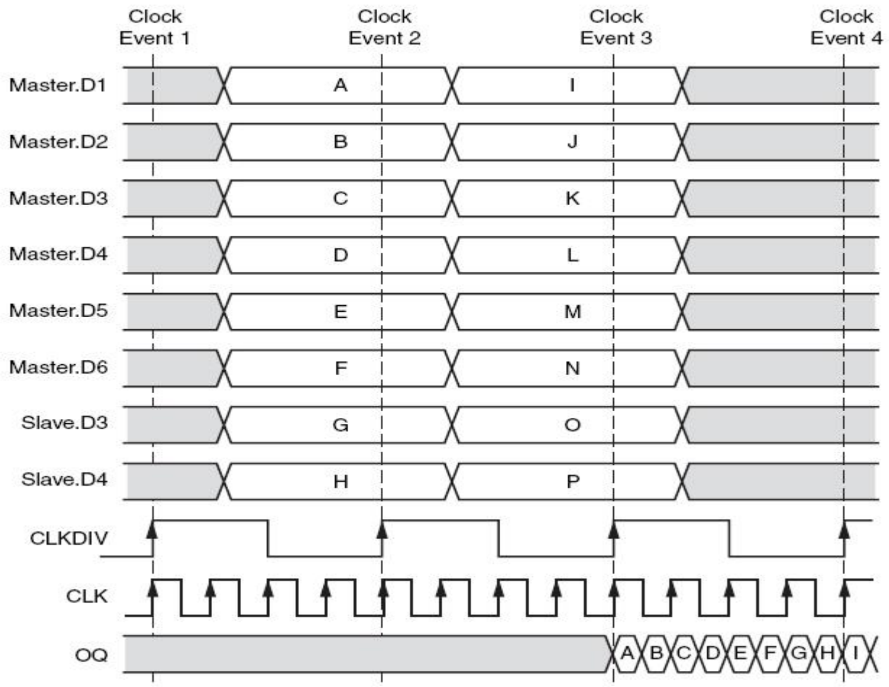


Figure 8: OSERDES timing diagram [8].

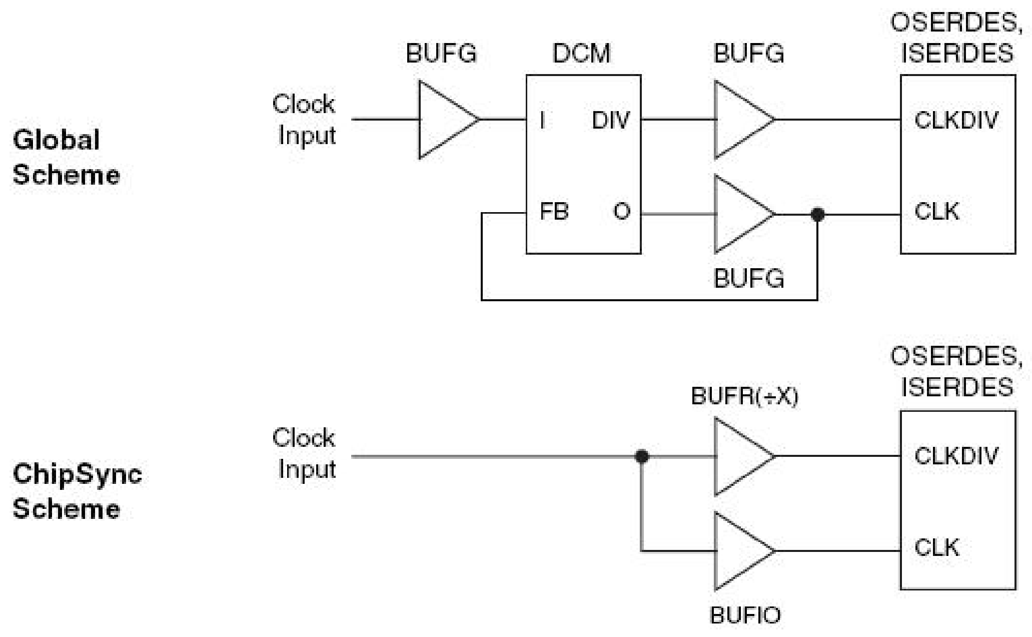


Figure 9: Two schemes of interface clocking [6].

3.5 Calibration

Calibration begins immediately after reset. Because the receiver must align the clock edges to the center of the data eye for each wire, it must do a training run at the beginning to calibrate the IODELAY modules. The transmitter will continually send a training pattern, and the receiver calibrates each channel in a round robin fashion. When the receiver finishes calibration, it sends a signal TRAINING_DONE back to the transmitter to announce that calibration is finished. In an alternate implementation, if there were no pins available for TRAINING_DONE, the transmitter can simply wait for a period of time (e.g. 500ms) then assume that calibration is finished and begin transmission. We decided to keep the TRAINING_DONE signal. Note that the TRAINING_DONE signal is held high once calibration completes.

The calibration setup on the receiver side is illustrated in Figure 7. The resource sharing module contains a state machine which selects the channel to align in a round robin fashion. The bit alignment machine aligns each channel to match the agreed training pattern. When alignment is finished for a single channel, the bit alignment machine outputs the INC, ICE and BITSLIP signals to tune the IODELAY modules. The reference design also allows inputs into the module to manually tune the IODELAY; our implementation does not utilize this.

Currently, the transmitter sends the pattern 00101100 (0x2C) on all 16 channels during training. The receiver bit alignment machine checks for 0x2C on each channel. There can be many good training patterns; ideally, a good pattern would resemble real data in having various bit run lengths; e.g. 10101010 (0xAA) is less optimal compared to 0x2C (00101100).

3.6 Recalibration & Reset

To recalibrate, both the transmitter and receiver interchip modules must be reset correctly. When the transmitter is reset, it will automatically send the training pattern until TRAINING_DONE is received. Therefore, the receiver must de-assert TRAINING_DONE, allow transmitter to reset, wait for IDELAYCTRL on the receiver side to recover then reset the receiver. The IDELAYCTRL module takes up to a hundred cycles to fully recover. This procedure described in the Xilinx app note [6] but not implemented in our testing framework. We did not implement recalibration for the testing application since we wanted to calibrate at one temperature and continue collecting data across varying temperatures.

3.7 QSH

The QSH cable supports 38 differential pins from each chip, 19 differential pairs on QSHA and 19 on QSHB. This module sends from QSHA with 17 differential pairs (16 for data, 1 for clock) on the transmitter, and receives 17 differential signals on QSHB at the receiver. Note that the clock must be sent to a clock capable I/O pin on the receiver chip. On the BEE3, this would be 9, 10, and 11 differential pair on the QSH. For more information, see BEE3 ucf and the Virtex 5 pinout data sheet [9]. One wire in 18th differential pair is used for TRAINING_DONE. The last QSH differential pair is currently unused.

3.8 Forced Inversion

As mentioned before, the receiver performance tends to degrade as the data spectral content widens. The app note evaluates the performance based on various data patterns; the most stressful test uses a pseudorandom generator PRBS29, which has a maximum run length of 29 consecutive 1's or 0's. On the BEE3, the raw data across the link would have no limit on spectral content. To ensure high performance given any data pattern, we designed this forced inversion module to restrict the spectral content for any data pattern.

The goal of this module is to be simple in logic and circuit implementation; ideally, it would be easily parameterized and flexible to accommodate integration into different designs. We considered several implementations for this module:

1. Encoding data with a scheme similar to Ethernet encoding such as 4b5b/8b10b codes [10], which force narrow spectral content for the data going onto the wires.

2. Pulse shaping the data with a hardware filter.

Encoding and decoding data is somewhat complex, requiring at least a translation table; it also complicates debugging. Pulse shaping filters are complex in implementation, and require a lot of state to be stored. Both encoder and filter implementations are inflexible and difficult to parameterize.

Our implementation (Figure 10) uses a data modulation technique using simple circuit elements. We use a shift register cascaded with a register to produce a regular clock-like signal to modulate the data. The shift register would produce a single pulse every X cycles, and the register would be enabled to take as input the inverted value from output. This creates a signal which flips from 0 to 1 every X cycles. This signal is used to invert the transmitted data such that it would send the raw data for X cycles then send the inverse for the next X cycles.

This scheme works to modulate the data for many data patterns except for one: a signal which coincidentally flips from 0's to 1's every X cycles, e.g. 1,1,1,1,0,0,0,0 if X=4. For this pattern, the data on the wire would still be 1,1,1,1,1,1,1,1 which maintains a direct current, resulting in a very uneven frequency response. To address this issue, a hold signal is sent to the transmitter every time the control signal flips; this hold signal notifies the transmitter to hold the current pattern, and allows the transmitter to send the inverse of the current pattern. In the example above, the module would send 1,1,1,1,0,1,1,1,1. The timing diagram in Figure 11 illustrates this procedure where X=3.

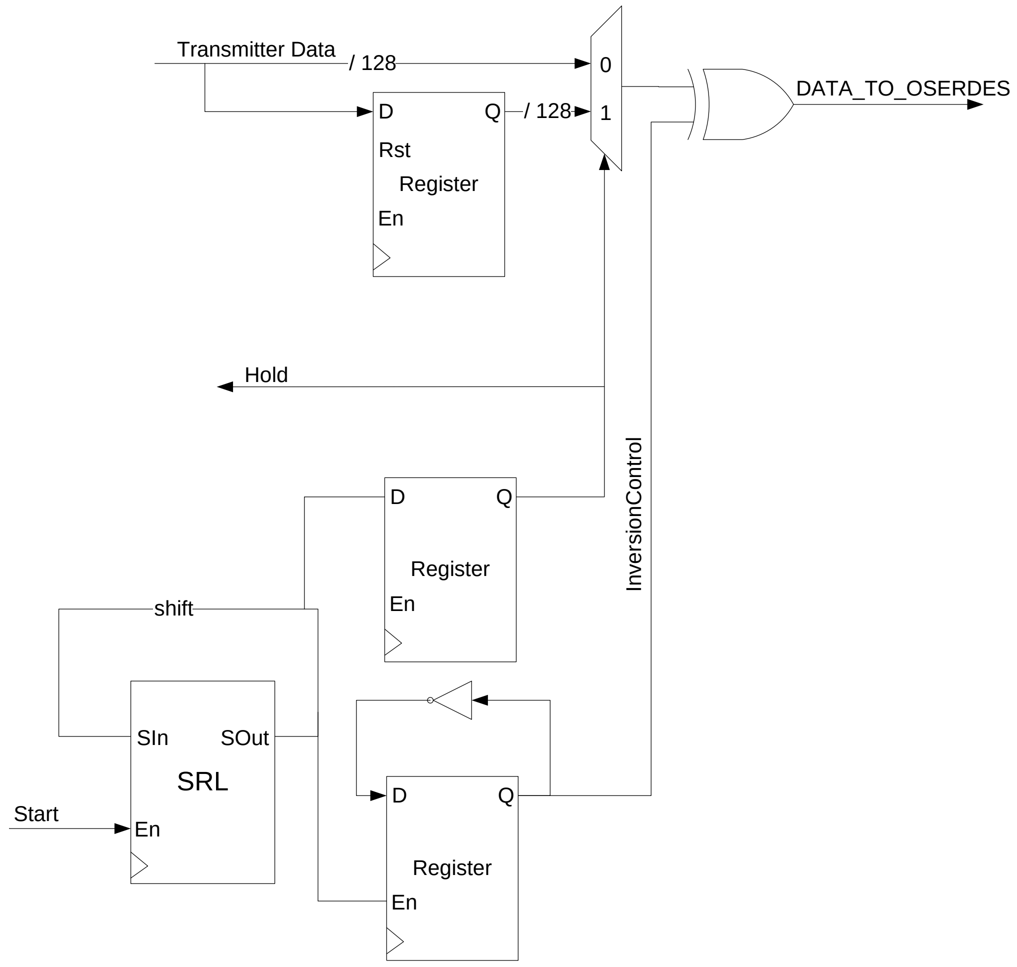


Figure 10: Forced inversion module

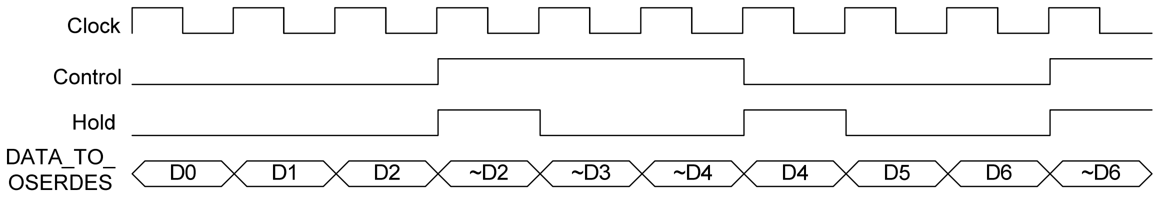


Figure 11: Forced inversion timing diagram where modulation period is 3

The same module is instantiated on the receiver side, but the hold signal becomes a “drop” signal. After X cycles, one data is dropped and ignored by the receiver and the following X data are inverted to recover the original data. Note that this module is instantiated between the interchip module and the asynchronous FIFO, such that dropped data never reaches the FIFO and the receiving test module knows nothing about the forced inversion.

The only caveat with this implementation is that both instantiations of this module have to start at the exact same cycle on both the transmitter and the receiver. Any phase shift would cause high error rates. The calibration completed signal "TRAINING_DONE" is not good enough since the transmitter receives this signal after an unknown delay. Hence, some handshaking is necessary after TRAINING_DONE is asserted to ensure that the transmitter and receiver start at the same time. A "Start" signal is input into the forced inversion module to begin the modulation. Figure 12 shows the transmitter and receiver state machines used to generate the "Start" signal on both chips.

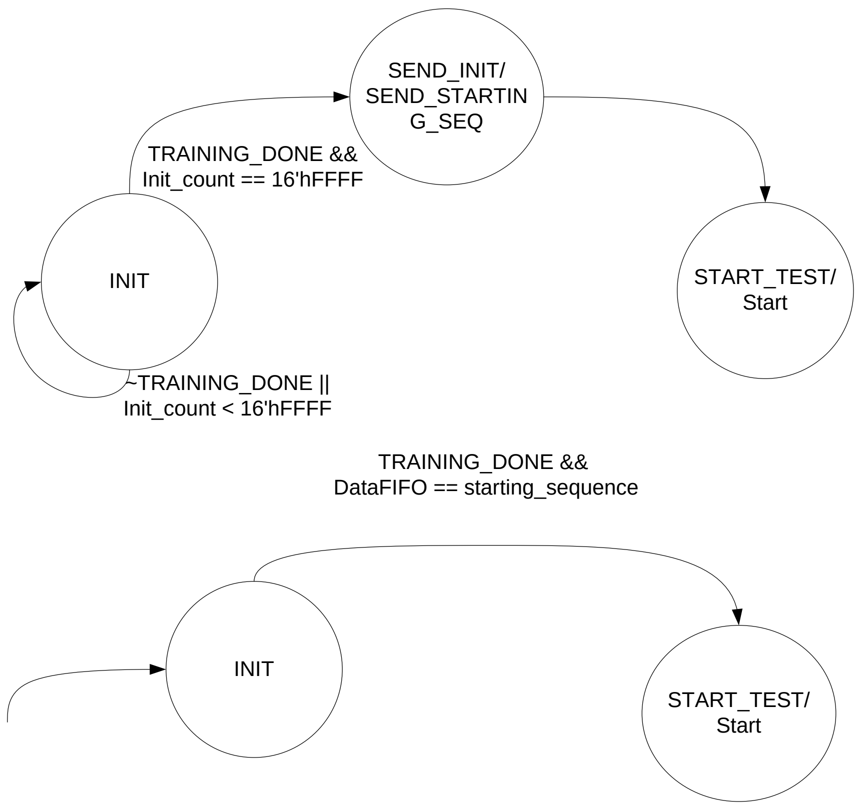


Figure 12: Top shows the transmitter state machine to start the test; bottom shows the receiver state machine.

This forced inversion module is simple in concept, simple in implementation: just a few registers with a built-in shift register LUT; more importantly, it can handle all data patterns. Except for the "Start" and "hold" handshaking signal to the transmitter, the module is integrated as a part of the interchip interface module. This module is very easy to parameterize: change X to any value in the shift register, and the modulation period would change accordingly. This means that the forced inversion can be changed to allow the maximum data rate while ensuring reliability for any application sending different data patterns at various frequencies and temperatures.

3.9 Testing Framework

The testing framework must not only be flexible to accommodate various patterns under various speeds, but also report the current temperature and error rates. Our testing framework tests five different patterns, counts the errors and uses the system monitor block to report the current temperature. This information is reported in hexadecimal format through the RS232 serial port from the receiver every 20 seconds.

As shown in Figure 14, the transmitter tester and receiver tester both instantiate a pattern generator module. The transmitter sends a pattern every four cycles, and the receiver compares the pattern with the output of its own pattern generator.

The five patterns our framework supports are:

1. All 0's
2. All 1's
3. All 0's for X cycles, then all 1's for the next X cycles, and so forth where X is the forced inversion period.
4. Counter
5. Pseudorandom number generated using LFSR

Patterns #1, 2 and 3 stress-test the forced inversion to check that it works for the more extreme cases. The counter and pseudorandom number patterns test the link. If the patterns don't match, the error counter is incremented, and the pattern generator is adjusted to the incoming value such that the next pattern would match the next incoming value. For example, if the receiver counter is at 3 but receives a 4, this counter is set to 4 such that next time it expects to see a 5. This guarantees that any errors are accounted for only once.

As mentioned before, the application using the interchip must use correct handshake with the forced inversion module to ensure that both the transmitter and receiver starts the inversion at the same time. In this testing framework, this is achieved through a state machine that runs immediately after calibration completes. Figure 12 shows the transmitter side and receiver side start state machines. The transmitter waits for the TRAINING_DONE signal then sends an agreed starting pattern after a few cycles; the receiver asserts TRAINING_DONE then waits for the starting pattern. There are other alternatives to this scheme, but we found this to be the most explicit and easiest to debug.

Note that the "Start" signal is sent into the forced inversion module, and is held high once the starting pattern is sent; this conforms to the expected handshaking shown in Figure 10.

In this testing framework, the transmitter application uses a DCM to generate a clock frequency of up to 500MHz. Since data is sent every four cycles, each 128 bits of data arrives at the receiver at the frequency of 500MHz/4≈125MHz. Hence, the receiver application can run at a much slower rate; we use a 200MHz clock to read from the FIFO (Figure 14). Also note that the IODELAY modules require a reference clock at 200MHz, so the receiver application must generate a 200MHz clock to drive the IODELAY, independently of the rest of the application.

The system monitor block [11] is generated with Xilinx core generator. Using the dynamic configuration ports, the temperature ADC code can be obtained. Using the ADC code, the temperature in Celsius is calculated below [11].

$$Temperature(°C) = \frac{ADCCode \times 503.975}{1024} - 273.15$$

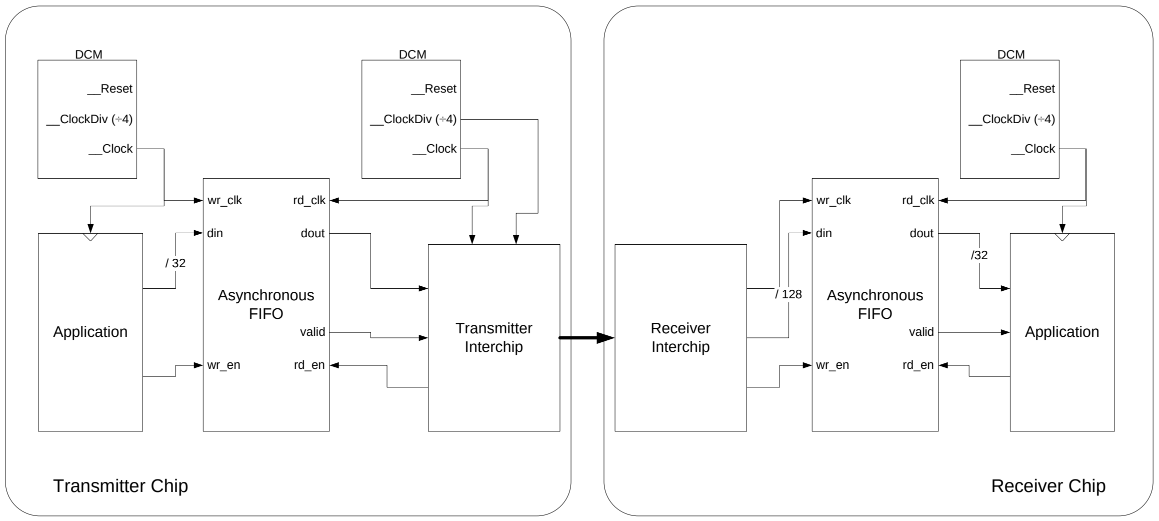


Figure 13: Example application using the interchip interface module.

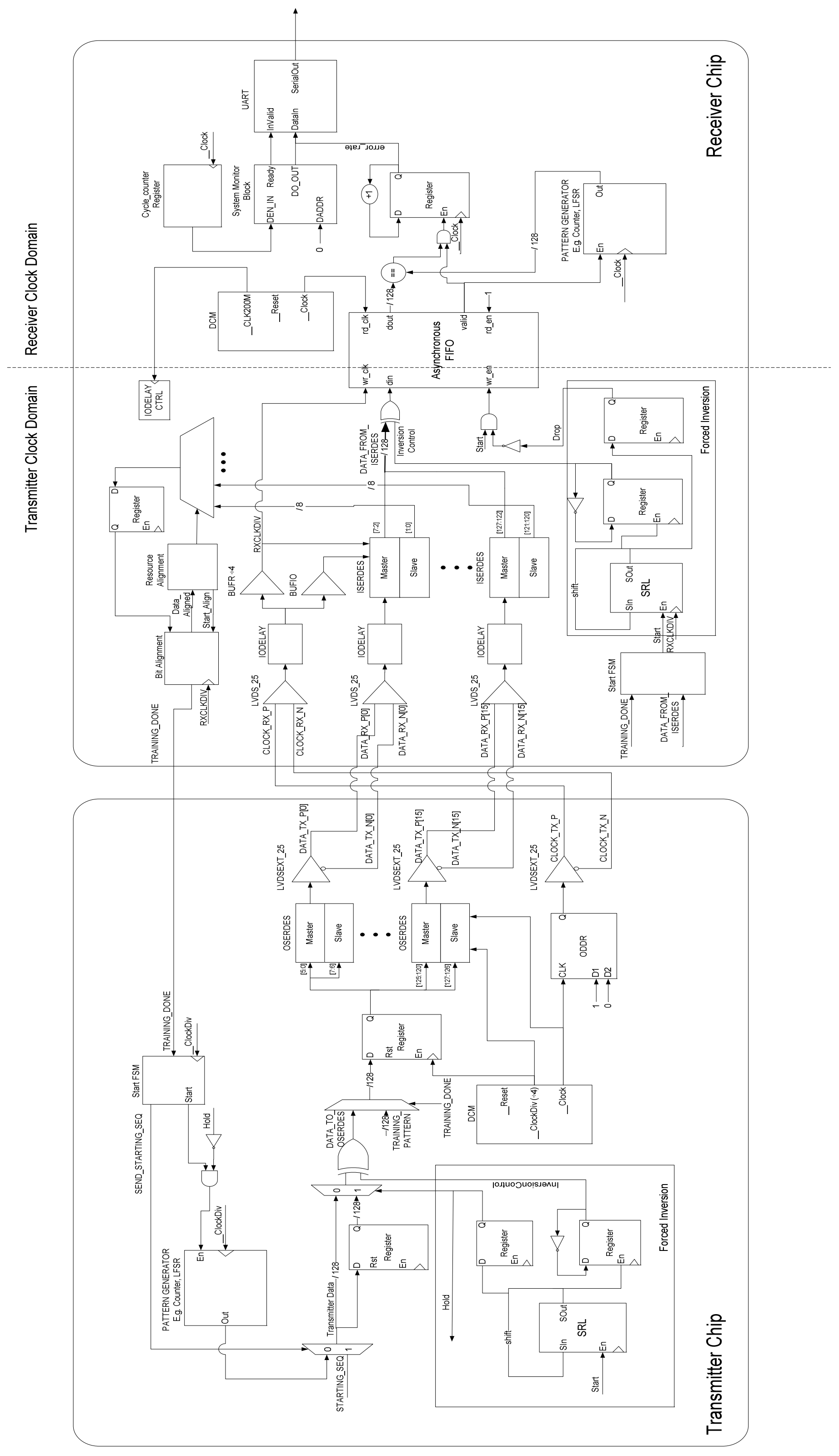


Figure 14: The entire BEE3 testing framework.

3.10 Using the interchip Interface

To integrate the interchip interface module into any applications running at various speeds, asynchronous FIFOs should be instantiated between the application and the link. Our testing framework instantiates one asynchronous FIFO between the receiver interchip and the testing application. The transmitter interchip is synchronous to the testing framework, so no asynchronous FIFO is necessary. Since there can be many ways of using the interchip, the FIFOs are not included in the interchip interface module.

For example, some application using the interface can drive the interchip at a very high frequency, and run the application itself with a much slower clock. If the application wants to continually send 32 bits of data, it would write 32 bits into the asynchronous FIFO; the interchip would read 128 bits whenever it becomes ready, and send the aggregated 128 bits in four interchip cycles. The receiver would receive all 128 bits, input into another asynchronous FIFO, and the application could read 32 bits at a time when it's ready. Figure 13 shows the set up for such an application.

One issue with using this interface is that if the sender wants to send only 32 bits for a long period of time, the transmitter should make sure to send the 32 bits even though there are fewer than 128 bits of data. The transmitter should use a timer to send a padded 128 bits at timeout, or the application can artificially pad garbage data into the FIFO to avoid deadlock. Section 6.2 also describes methods to parameterize this interchip interface such that the link can send 32 bits of data with shorter burst times to accommodate different applications.

4. Analysis

4.1 Performance

Because the QSH cable claims to support only 1Gbps maximum per wire, the module has only been pushed to 1Gbps (500MHz DDR). However, because of the forced inversion, some of the data on the line is repeated and dropped at the receiver. To accurately calculate the transfer rate from a transmitter application's perspective, assume that the forced inversion period is X, and clock is running at 500MHz:

$$R = \frac{8X - 8}{8X} \times 1Gbps = \frac{X - 1}{X} \times 1Gbps$$

where *R* represents the effective data rate, and *X* is the inversion period. Note that this is the effective data rate across a single wire, not the entire cable. This equation accounts for the fact that the forced inversion runs in the domain of the divided clock. Hence, the DDR wire actually sends $8 \times (X - 1)$ bits of real data before sending 8 bits of repeated data. In our implementation, *X*=29 according to the worst case specified by the app note. In this case, the effective data rate is 966Mbps per wire.

The data rate can be optimized by changing the modulation period. Naturally, as *X* increases, the effective data rate increases. Another way of optimizing data rate without changing *X* is to alter the forced inversion module such that it does not invert the entire 128 bits of data, but only the last 16 bits sent over the 16 wires. So instead of repeating 128 bits of data every *X* cycles, we would repeat only 16 bits of data - only 1 bit for each wire. The equation for data rate under this implementation would be:

$$R' = \frac{8X - 1}{8X} \times 1Gbps$$

With *X*=29, *R'* = 996Mbps per wire.

The tradeoff to this implementation is logic and circuit complexity. We would need to add another FIFO to keep track of which bits were to last go out on the wire, and only invert those bits to be sent out next cycle. This wasn't implemented since we did not feel that the extra logic complexity is a worthwhile tradeoff for the improvement in performance during link characterization; an application designer may choose differently.

Accounting for all 16 wires, the total data rate over the entire module at *X*=29 is $16 \times 966Mbps \approx 15.5Gbps$.

4.2 Resource Utilization

One of the goals of the project is to keep the module simple and small while optimizing the performance. The table below table reports the synthesis resource utilization of the transmitter and receiver interchip modules with forced inversion. This does not include the test framework logic and asynchronous FIFO.

Component	Quantity	Usage Description
Slice Flip-Flop	128	Multiple uses
LUT	0	-
IOB	35	17 LVDS pairs + TRAINING_DONE
OSERDES	32	Master and slave OSERDES
BUFG	2	Clock, ClockDiv
DCM	2	Clock, ClockDiv

Table 1: Resource utilization for transmitter interchip interface module

Component	Quantity	Usage Description
Slice Flip-Flop	131	Multiple uses
LUT	253	Multiple uses
IOB	35	17 LVDS input pairs + TRAINING_DONE
ISERDES	32	Master and slave ISERDES
BUFIO	1	RXCLK
BUFR	1	RXCLKDIV

Table 2: Resource utilization for receiver interchip interface module

4.3 Reliability

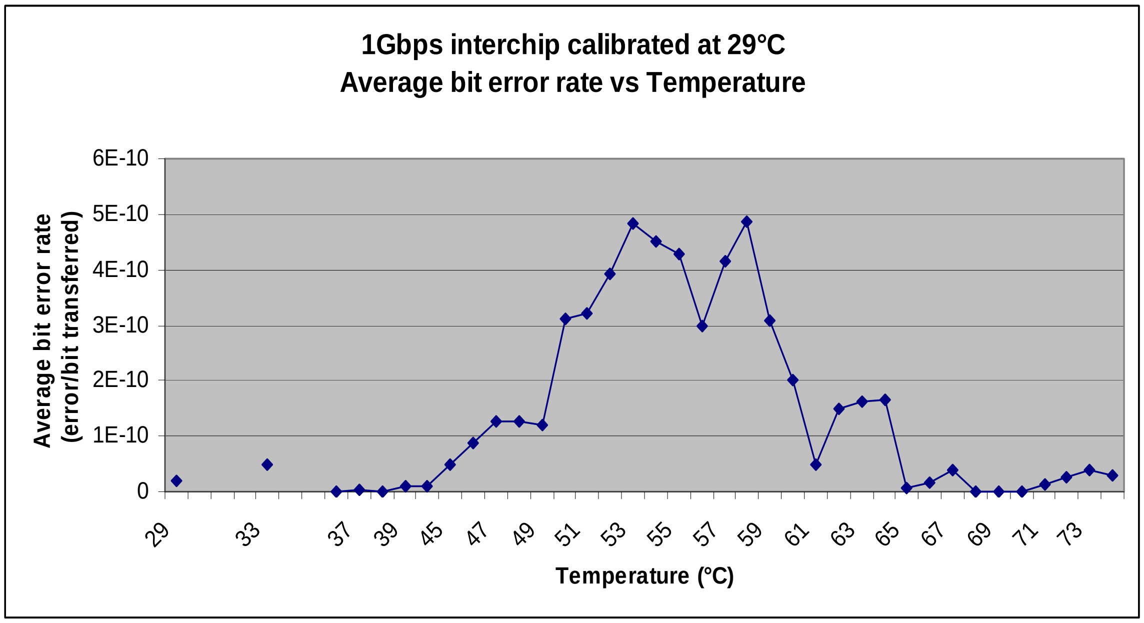
To evaluate the reliability of the link, the testing framework monitors the temperature and counts the error rate and prints it through the RS232 every 20 seconds. The link is calibrated at several different temperatures, and a heat gun is used to artificially heat the chips during the test. An environmental chamber would have been more ideal to heat or cool down the chips, but this equipment was unavailable. Both the transmitter and receiver chips were heated in a somewhat uniform fashion such that both the transmitter and receiver operated at around the same temperature. After a few hundred samples have been collected, the raw data is parsed using a script, fed into excel and graphed.

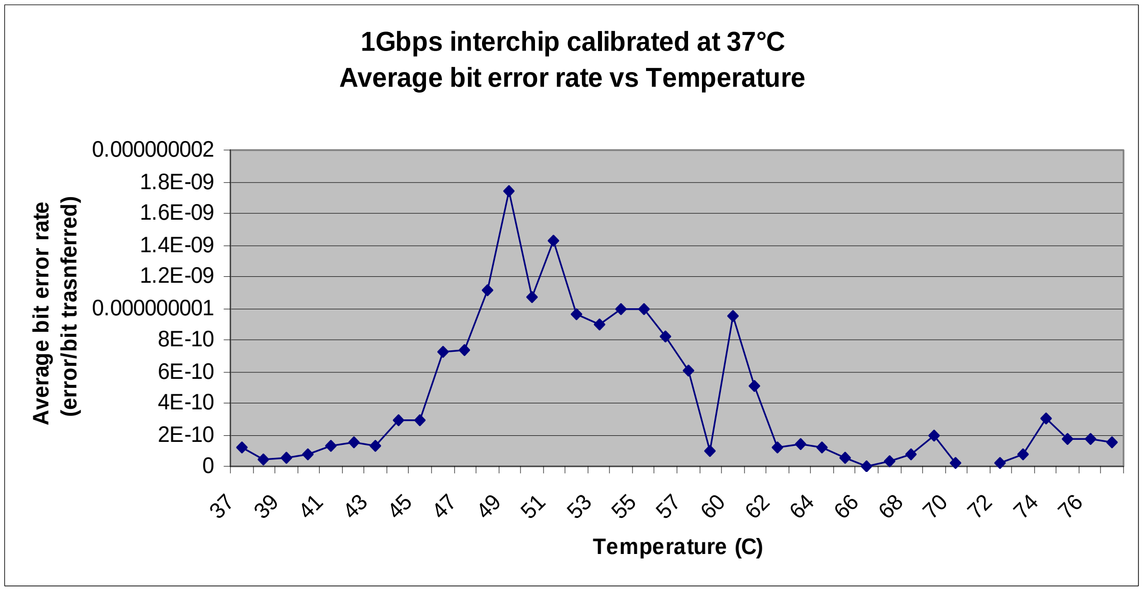
In this section, we refer to the raw data rates (as opposed to effective data rates) over one wire as just “data rate”, for simplicity. We refer to average bit error rate as the error encountered per bit transferred. Even though the test application records if one or more bits of the received 128 bits are incorrect, we assume that there is only one incorrect bit for every error encountered. In the absence of better data, this assumption stands on the observation that since the link is mostly reliable across various data rates, the chance that many bits are failing at the same time is low.

Table 3 shows the average bit error rate across all temperatures for each data rate. Note that for all data rates below 1Gbps, the bit error rate is found to be constantly zero. At 1Gbps, the link starts to fluctuate at varying temperature. All tests are run at approximately 15-25 minute intervals each, with five trials per test. The reported error rates are the average of all five trials. Figure 15 shows the average bit error rate at each temperature, calibrated at three different temperatures.

Data Rate	Bit error rate (error/bit transferred)
800Mbps	0
900Mbps	0
950Mbps	0
1Gbps	2.5×10^{-10}

Table 3: The average bit error rate at various data rates.


1Gbps interchip calibrated at 29°C
Average bit error rate vs Temperature
Average bit error rate (error/bit transferred)
6E-10
5E-10
4E-10
3E-10
2E-10
1E-10
0
29
33
37
39
45
47
49
51
53
55
57
59
61
63
65
67
69
71
73
Temperature (°C)


1Gbps interchip calibrated at 37°C
Average bit error rate vs Temperature
Average bit error rate (error/bit trasnferred)
0.000000002
1.8E-09
1.6E-09
1.4E-09
1.2E-09
0.000000001
8E-10
6E-10
4E-10
2E-10
0
37
39
41
43
45
47
49
51
53
55
57
60
62
64
66
68
70
72
74
76
Temperature (C)

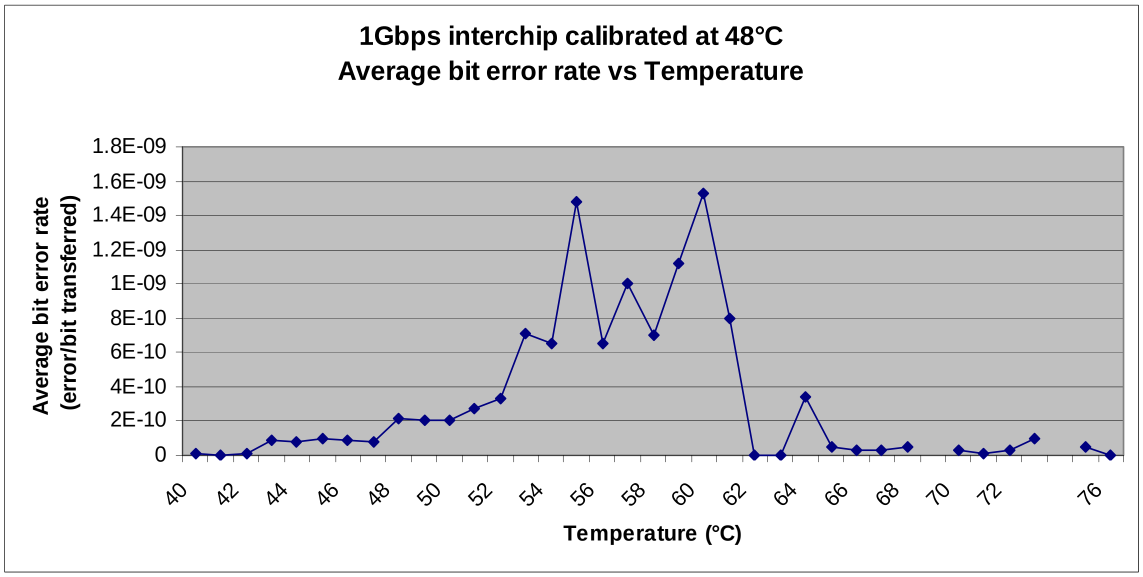


Figure 15: Three graphs showing the average bit error rate versus temperature, calibrated at three temperatures.

For all three calibrations, the error rate would increase exponentially up until 10-15°C above calibration temperature then drop exponentially as the temperature continues to increase. It was expected that the error rate would increase exponentially as the operating temperature increases above the calibration temperature, but we did not expect the error to come back down at higher temperatures. One possible explanation is that the temperature causes a skew in data sampling; the bigger the delta in temperature, the greater the skew. Naturally, when the temperature rises higher and higher, the clock would shift away from the center of the data eye, causing sampling errors. But as the clock shifts more and more away from the center of the current data, it comes closer and closer to the center of the next data eye. At that time, the clock is once again aligned with the data. Although the clock is shifted from the original position, the data is still valid.

To verify the theory that the clock shifts by an entire phase at a temperature delta of +30°C, we can run the test with a larger temperature range: calibrate at close to 0°C, and heat the chips up to 80°C. We can also see if a similar pattern persists at a different data rate with a slightly different curve shape. However, we have no resources to dramatically lower the temperature to 0°C, and it is clear from Table 3 that this behavior only happens at a data rate very close to 1Gbps. Even though we are unable to verify this theory at this time, we explain the testing procedures necessarily in the future works section.

Results in Table 3 show that the link can run reliably at all temperatures at data rates less than 1Gbps, regardless of the calibration temperature. To run the link reliably at 1Gbps, we should implement Error Correction Codes, and enforce re-calibration when the operating temperature rises to ~10°C above the calibration temperature. To avoid re-calibration due to varying temperature, a real-time monitoring module should be integrated into the interchip interface. Section 6.6 describes this module and points to the Xilinx reference design. We decided not to use this module since it uses twice the chip area as the current module, but only improves the data rate to 1.2Gbps. Since the QSH cable claims to support only 1Gbps transfers, we thought the real-time monitoring module was not necessary for the BEE3.

4.4 Forced Inversion

To test the forced inversion module, five different data patterns are generated by the testing framework. The bit error rate is 0 for all five patterns run at the calibration temperature for all data rates. To further verify the effect of the forced inversion module, tests are run on an interchip module without the forced inversion. Using just the pseudorandom number generator (LFSR), the error rates are high and consistent across varying temperatures. Table 4 shows a comparison of the average error rates at various speeds with the two implementations. The link breaks down completely at 500MHz without the forced inversion, and shows a significant error rate at 400MHz. It is clear that the forced inversion was necessary and effective in addressing the spectral content.

Data Rate	With Forced Inversion	Without Forced Inversion
800Mbps	0 error/bit	3.4×10^{-9} error/bit
1Gbps	2.5×10^{-10} error/bit	0.08 error/bit

Table 4: The average bit error rate across all temperatures at 400MHz and 500MHz with and without the forced inversion.

5. Experience

5.1 Development & Tools

We used Xilinx tools for both development and debugging. We found the tools to be surprisingly painless to use. ChipScope was somewhat helpful as a basic debugging tool, but since it is synchronous and can only be run after the design is programmed, we could not fully debug the subtle timing bugs with ChipScope. On the other hand, we would have preferred to have more LEDs per chip for debugging purposes. Even though LEDs are not useful for fine grained signals, they are a convenient way to display what's going on in the system. For example, to verify that the clock manager is producing the correct frequencies, we blink the upper bits of a counter incremented with the clock.

Our development experience also taught us to be pessimistic about large modifications. As we were building an increasingly complex framework, we learned to make incremental changes and do regression tests often. Having multiple clock domains in the system also contributes to some nondeterministic behavior that becomes hard to track down as the system becomes larger and more complex. We spent most of our time not creating new modules or tests, but tracking changes and locating where the new errors originated throughout the course of our many modifications.

We are also pleased to say that we found no blatant bugs and errors in the Xilinx reference design. Although we found some specifications to be lacking in detail on the application note, the module itself turned out to work as designed. Most of our problems stemmed from using it in context of a larger framework, in particular, how to use different clocks and deal with timing issues.

5.2 Clocks & Debugging

The biggest issues we encountered in this project are timing errors. Not only did we lack a fundamental understanding of various issues with clocks, but it was also difficult to debug timing errors. Sometimes the module seemed unstable at the beginning, but started working after a while, other times tying up LEDs to certain signals would make or break the entire design.

Some of the lessons we learned regarding timing errors include:

- When crossing clock domains, make sure the control signals are synchronous to the respective clock domain. For example in the asynchronous FIFO, the read enable signal should be synchronous to the read clock, and the write enable signal should be synchronous to the write clock. Even if the enable signal is held high for a long period of time, it is more accurate to latch it with a register with the correct clock to enforce a synchronous signal.
- Enforce timing constraints on all clock signals. This is intuitive since the tools cannot evaluate the timing constraints of the circuit without explicit requirements from the programmer. Forgetting to do this may be acceptable at a slower speed, but will start to display strange behaviors at higher speeds.
- If working with a design at high clock frequencies and multiple clock domains, always use the PAR timing analyzer to confirm that the timing constraints are applied correctly.

5.3 Working with BEE3

Working with the BEE3 had its up and downs. On one hand, we were lucky to get almost exclusive access to one of the few BEE3 boards; on the other hand, the board often went away for show, which cut into our productivity. But the biggest problem with the BEE3 is that it is poorly documented: there is no central resource that lists all the specifications and components on the BEE3. Especially at the beginning of the project, we resorted to emailing individuals, flipping through raw data sheets and browsing PowerPoint presentations with facts littered across the slides. This caused so much confusion that culminated at our wasting a week debugging the interchip module because it was not doing anything whatsoever, only to realize that we did not plug in the external QSH cable to connect the pins.

6. Future Work

The current module runs reliability at a reasonable speed, but there is still a lot of work to increase speed and reliability, and parameterize the module to conform to various designs. There are also many other interesting tests to run on the link to analyze the behavior of source synchronous communication at high data rates.

6.1 Using Ring in place of QSH

There is a ring of wires called the RING that connects each FPGA. These can be used for interchip communication instead of the QSH connector. The RING connections on the board are specified to run at a maximum data rate of 800Mbps.

Since the RING connections are designed to be single ended, the interchip must be modified to send and receive single ended data for both data and clock. Note that the clock should still be routed to clock capable I/O to ensure correct timing.

6.2 Parameterization

The module can be parameterized to send variable length data at different burst lengths. The current module is hard-coded to send 128 bits in parallel across 16 channels with 8 bits burst length. Currently, the OSERDES and ISERDES are set at 6:1 DDR serialization; but these modules can be configured at 4:1 DDR or 2:1 SDR modules to support a smaller parallel width and shorter burst lengths. Although the parameters are not totally flexible, they can be set at different levels by generating fewer and differently configured OSERDES and ISERDES. For example, the module can be 64 bits wide at 4 bits burst length using 16 channels; or 16 bits wide at 8 bits burst length, using only 2 channels.

To achieve this, the OSERDES and ISERDES have to be instantiated inside generate blocks. Using if statements conditioned by the burst length and data width, the block can generate the number of OSERDES and the appropriate configurations. The DCMs generating the clocks must also evaluate the burst lengths to send in the correct divided clock. For instance, if the OSERDES are set to 4:1 DDR, the divided clock should be divided by 2, not 4. The FIFOs are less easily parameterized since they are generated by coregen and the read and write widths are configured beforehand. In order to keep the module parameterizable, multiple FIFOs should be generated using coregen, and the Verilog generate statement should choose the correct one based on the parameters.

6.3 Reset

Our module cannot be reset after it is programmed onto the board, but this is possible. The reset procedure is outlined in the Xilinx application note [6]. According to the note, transmitter should be reset before the receiver; the IDELAYCTRL module must wait for a few hundred cycles before it recovers from the reset. The receiver must detect a drop followed by a rise in IDELAYCTRL_READY signal before resetting itself. From this sequence, it seems that the transmitter and receiver must communicate by sending a control signal indicating that a reset is taking place. This can be initiated by either the receiver or the transmitter.

6.4 Modify test framework, add random error insertion

In retrospect, we realized that the testing logic can be improved by having the transmitter insert error once in a while to ensure that the test is running correctly, and that the receiver is reporting the correct results. This can be achieved by instantiating a module very similar to the forced inversion: every X cycles, the inverse of the current data is sent over the line. The receiver should report a consistent error rate for each interval, and any actual errors would still be accounted for on top of the random error insertion. A disadvantage of this approach is that it is possible to drop an inserted error, causing the receiver to fail to account for an error. This would result in an inaccurate report of error rates. But assuming that X is large and the link is reasonably reliable, the average error rate should be accurate.

This would be helpful to determine whether: 1. the test is actually running after calibration, 2. whether the testing logic is correct even if it reports no errors and 3. whether the error count is accurate. Our current test framework can use the serial port to print out valuable information for the human tester to check that the tests are functional, but random error insertion would a helpful systematic tool to ensure that each test run is valid.

6.5 Running additional tests

The test suite we created is rudimentary in determining the general reliability of the link at various temperature settings. Because we had just a hot air gun and limited time to perform the tests, we were unable to run many interesting test cases across the entire temperature range. It may be valuable to calibrate the module at extreme low temperatures of close to 0°C, and gradually heat both chips up to 80°C to evaluate behaviors across the entire temperature range. This test may reveal some interesting patterns in error rates and help validate the theory regarding clock shift proposed in section 4.3. One could alternatively calibrate the module at 80°C, and cool it down to 0°C to check whether the reciprocal occurs.

Another interesting test would be to tweak the forced inversion configuration to see how the modulation period affects the performance of the link at various speeds. Currently we compare the performance with the link at a modulation period of 29 cycles, to the unmodulated link. We chose 29 cycles because the application note implies that this is the longest period of consecutive 0's and 1's seen in the pattern before the link breaks down. But it is not clear how the forced inversion implementation affects the performance under different settings. To enable this test, the receiver must read from the serial port the desired modulation period prior to starting the test; then it must communicate with the transmitter through one pin using a standard UART module. Both the transmitter and receiver must then configure the shifter to the desired period. After this initial configuration finishes, both modules may begin the test.

6.6 Add real-time monitoring

The module used in this project is the basic LVDS DDR link; Xilinx has another application note [12] for a similar implementation with a real-time monitoring module that detects sampling skew in real time and adjusts accordingly. We chose not to use this module because the resource utilization is twice as large as the current module, while the data rate is only ~200Mbps more. Additionally, since the BEE3 hardware can only support up to 1Gbps, there is no need for a complex interchip module that runs at data rates greater than 1Gbps.

For future applications, however, real-time monitoring may be useful. If the large resource utilization is acceptable and a higher data rate is needed, this module can be instantiated in the same testing framework. The transmitter is exactly the same, and the receiver has an additional monitoring module which dynamically adjusts the sampling window. The input and output netlists are almost exactly the same, except that the receiver takes as input RT_MANUAL_DISABLE, which is used to disable dynamic monitoring and adjustments. By setting this signal high, the link should exhibit the same behavior as the current link.

6.7 RDL source synchronous link

The interchip and testing framework are written in Verilog. It is possible to integrate this into RDL in the future. Ideally, an RDL unit would be able to instantiate a source synchronous link and parameterize the data width, burst length, and forced inversion period. As described above, these parameters are feasible. The RDL unit would also need to specify the data rate of the link, and specify the asynchronous FIFOs necessary to interface with the interchip module.

7. Conclusion

We implemented a source synchronous FPGA to FPGA link and developed a "forced inversion" module which modulates the data to ensure that any raw data pattern is transmitted reliably. We have analyzed the error rates at varying temperatures and shown that the link runs reliably at data rates of up to 966Mbps per wire – 15.5Gbps total. The tests also indicate that the forced inversion is crucial in maintaining the reliability at high data rates. We have also documented the development and testing experience, along with any potential future work for this module.

This project has created a parameterized, high speed and reliable link that can be used in many applications on the BEE3. Many components in this interchip such as the forced inversion module are used for the first time on an FPGA and can be independently utilized for other FPGA applications.

8. Acknowledgements

The entire RAMP team and other members of the BWRC have been instrumental to the success of this project. I would like to thank Professor John Wawrzynek for his guidance and advice throughout the project.

Greg Gibeling, who developed RDL and RDLC, has not only provided valuable guidance and mentorship, but has also been a savior with resolving the most obscure and troublesome issues. I would like to thank Alex Krasnov for helping me with various issues with the tools and the debugging. I am grateful to Adam Megacz for providing a working Selectmap interface crucial to testing the BEE2 interchip module. Chen Chang, one of the designers of the BEE3 board, introduced me to the interchip design and provided much valuable information about the BEE3. Dan Burke was very attentive and responsive to all of my questions both large and small. Members of the undergraduate students in the RAMP group have also contributed enthusiastic suggestions and insights during the course of this project.

Last but not least, I would like to thank Professor John Wawrzynek, Professor Krste Asanovic and Greg Gibeling for proofreading this paper.

References

[1] Chen Chang, John Wawrzynek, and Robert W. Brodersen. BEE2: A High-End Reconfigurable Computing System. 2005. IEEE Design and Test of Computers, March/April 2005 (Vol.22, No.2).

[2] Arvind, Krste Asanovic, Derek Chiou, James C. Hoe, Christoforos Kozyrakis, Shih-Lien Lu, Mark Oskin, David Patterson, Jan Rabaey, and John Wawrzynek. RAMP: Research Accelerator for Multiple Processors – A Community Vision for a Shared Experimental Parallel HW/SW Platform. Technical Report UCB/CSD-05-1412, 2005.

[3] Laxmi Vishwanathan, Dan Schaffer, Jock Tomlinson, Lattice Semiconductor Corp. *Overview of Memory Types and DDR Interface Design Implementation*. http://www.fpgajournal.com/articles/20041109_lattice.htm.

[4] Greg Gibeling. RDLC2: The RAMP Model, Compiler & Description Language. Master's report, 2008.

[5] Xilinx. *Synchronous Timing.* http://toolbox.xilinx.com/docsan/xilinx7/books/data/docs/cgd/cgd0042_7.html.

[6] Xilinx. *16-Channel, DDR LVDS Interface with Per-Channel Alignment.* http://www.xilinx.com/support/documentation/application_notes/xapp855.pdf.

[7] Andrew Schultz. RAMP Blue: Design and Implementation of a Message Passing Multi-processor System on the BEE2. Master's thesis, 2006.

[8] Xilinx. *Virtex-5 FPGA User Guide.* http://www.xilinx.com/support/documentation/user_guides/ug190.pdf.

[9] Xilinx. *Virtex-5 FPGA Packaging and Pinout Specification.* http://www.xilinx.com/support/documentation/user_guides/ug195.pdf.

[10] IEEE LAN/MAN Standards Committee, "IEEE Std 802.3-2005 Carrier sense multiple access with collision detection (CSMA/CD) access method and physical layer specifications," IEEE, December 2005.

[11] Xilinx. *Virtex-5 FPGA System Monitor User Guide.* http://www.xilinx.com/support/documentation/user_guides/ug192.pdf.

[12] Xilinx. *16-Channel, DDR LVDS Interface with Real-Time Window Monitoring.* http://www.xilinx.com/support/documentation/application_notes/xapp860.pdf.