A SCALABLE SYSTEM ARCHITECTURE FOR USE WITH FREE SPACE OPTICAL INTERCONNECTS IN A 3-D STACKED PROCESSOR ENVIRONMENT

by

James Fleming Rorie, Jr.

A dissertation submitted to the faculty of
The University of North Carolina at Charlotte
in partial fulfillment of the requirements
for the degree of Doctor of Philosophy
in the Department of Electrical Engineering

Charlotte

1999

Approved by:

____

Dr. Fouad E. Kiamilev

____

Dr. Thomas P. Weldon

____

Dr. Barry Wilkinson

____

Dr. Hassan M. Razavi

____

Dr. Tom Reynolds

ABSTRACT

JAMES FLEMING RORIE, JR.. A Scalable System Architecture for use with Free Space
Optical Interconnects in a 3-D Stacked Processor Environment
(Under the direction of DR. FOUAD E. KIAMILEV)

This dissertation covers the development of a computer architecture to support synchronous serial communications over free space optical interconnects using the 3-D stacked processor technology. The architecture is based on existing implementations of message passing parallel systems, but adapted for the unique needs of an opto-electronic environment. A novel approach to dynamic clock synchronization over a free-space transmission bus is presented. Development of an architectural specification through identifying the components of a functional description and mapping them to an appropriate network model is outlined. By applying these techniques, the designer has the advantage of field tested solutions to the problems that occur in the development of complex, hybrid, hierarchal opto-electronic architectures and protocols.

DEDICATION

To my family for their love and support.

“If we knew what we were doing, it would not be called research, would it?”
Albert Einstein

ACKNOWLEDGEMENTS

I would like to thank my advisor, Dr. Fouad Kiamilev, for his guidance in this project. Additionally Jeremy Ekman, Premanand Chandramani and Jim Rieve provided invaluable assistance at times that it was most needed.

This work is sponsored by the Defense Advanced Research Projects Agency (DARPA) and the Air Force Research Laboratory under agreement number F30602-97-2-0122.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1 - INTRODUCTION

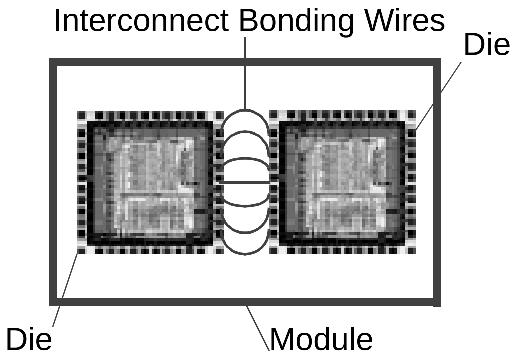
Future trends in computer development will encompass areas where there is a requirement of high data throughput to achieve useful operation. Many applications in these areas, which include real-time and adaptive signal processing systems, are the key benchmarks for performance evaluation of new technologies. These applications present the new challenges for the next decade of computation.

To meet the requirements of these complex applications, the concept of parallel processing must be applied. However, traditional planar VLSI technology supports these parallel implementations at the expense of die area. The increased transistor count and routing can be accommodated through the use of larger die and denser fabrication processes. However, these processes are more expensive due to increased fabrication cost and decreased yields.

The use of advanced packaging strategies can increase total transistor count while staying within common VLSI processes. Multi-chip Modules (MCM's) allow multiple VLSI dies to be combined to form a high density logic unit. The process uses cores fabricated using traditional processes and bonds them on an external substrate. Wire or optical interconnects are used to establish communications between the dies. The result is a slightly larger device that has twice the density of a normal die with a much higher yield than a monolithic approach.

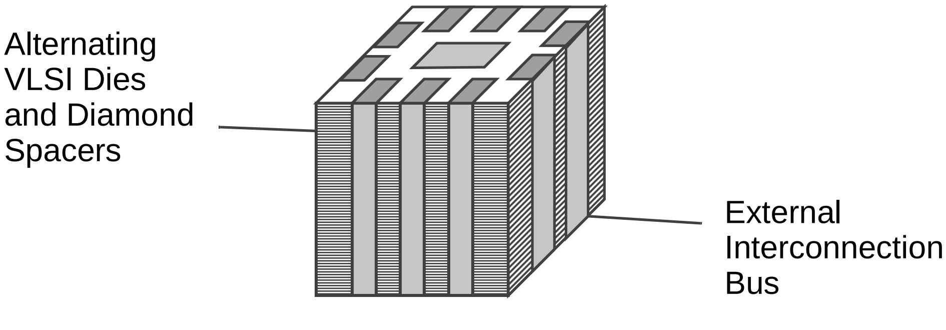
This technology suffers from a few limitations. Parallel systems can require a large increase in routing between functional units. Using more than two cores can increase routing geometrically, assuming a point-to-point connection for each bonding pad. It is in this area where planar VLSI technology becomes inefficient. To increase the number of modules, we must take advantage of the third dimension.

FIGURE 1. Planar Multi-Chip Module



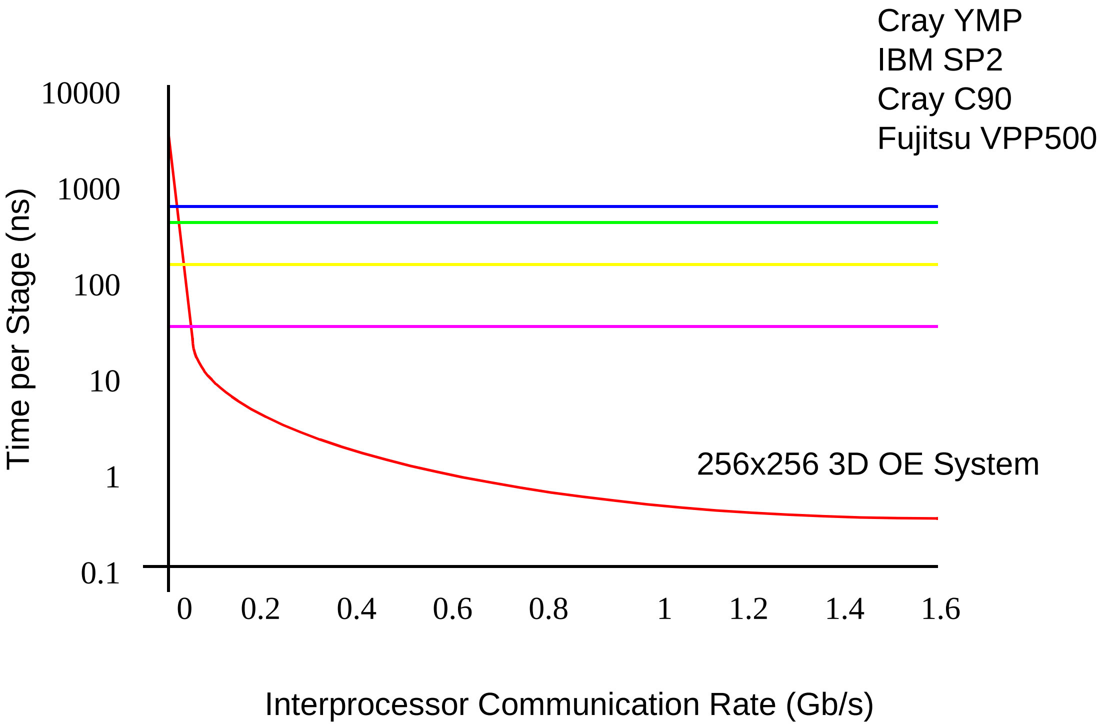
Stacking technology, developed by Irvine Sensors, Inc., is an approach by which separate VLSI dies are stacked vertically to form a three dimensional logic unit. By using the third dimension, the designer can build highly parallel systems in a uniform manner. Each parallel functional unit can be designed discretely in a planar fashion and can be expanded through stacking to meet computational needs.

FIGURE 2. Vertical Stacking


Alternating VLSI Dies and Diamond Spacers
External Interconnection Bus

Evaluations of this new technology have shown the potential for several orders of magnitude increase in processing capability. Figure 3 shows a comparison performed by Betzos[1] of a theoretical 256 x 256 opto-electronic computer with a number of the world's fastest systems. The benchmark used a 3D Fast Fourier Transform with a data size of 256 x 256 x 128 and plotted the time for the execution of the one stage of the calculation. It also assumes a fixed performance of 7 ns per operation using a 64-bit floating point arithmetic. The results showed that for interprocessor communication times greater than 100 Mb/s, the opto-electronic computer out performed all the other systems. For communication rates greater than 2 Gb/s, the opto-electronic system was two orders of magnitude faster.

FIGURE 3. FFT Performance Comparison



However, this new technology presents fundamental problems in the area of routing. Since components are no longer located local to a single die, some novel approach must be taken to allow communication between planes. For low speed signals, this global routing can be performed by adding traces around the die stack. But high speed datapaths require a more robust solution. These signals require electrical isolation to prevent stray electromagnetic interference from affecting the transmission. Current planar technologies implement high speed datapaths through the use of electrical planes as EMI shields. Unfortunately, this approach is not feasible in the stacked processor technology.

There exists a technology suited for dealing the unique constraints of the three dimensional stacked processor environment; Free Space Optical Interconnects(FSOI). This technology uses lasers and optical detectors to transmit information through free space. Previous FSOI technology used discrete lasers and Multiple Quantum Well technology to create an optical I/O channel. But recent advances in fabrication have allowed the use of VCSEL's, Vertical Cavity Surface Emitting Lasers, as a communications medium.

VCSEL's are naturally formed in arrays, thus they have the advantage of being well suited for the fabrication of 2-D optical communications matrices. This format is ideal for communications between 3-D processing stacks. By use of an optical approach, the problems associated with the coupling effects of electrical connections carrying high speed signals are eliminated.

1.1 Contributions

This paper strives to make the following contributions. First, a completely specified design for an opto-electronic stacked processor architecture is presented. This design is based on behavioral/structural simulations and addresses the low-level functional problems associated with the unique technologies that are incorporated.

Second, a potentially useful method for modeling hybrid opto-electronic systems using networking components as models is suggested. Through use of this approach, problems that occur in development can be identified through simple analysis of the known models. The result is a structured approach to opto-electronic design that could provide an aid for the future development of more complex hybrid opto-electronic architectures.

Third, a high speed transmission clock synchronization design is presented. This unit is capable of operating at the full transmission data rate without the need for Digital Phase Locked Loops (DPLL's) or clock encoding by taking advantage of the unique 3D-OESP environment. By assuming a synchronous clock with negligible drift, this high speed technique provides data clock synchronization at a minimal hardware cost.

Finally, a power-up address resolution technique to achieve unique processor identification is explored. While similar techniques exist in more complex architectures, this approach uses the hierarchal nature of the architecture and dynamic interconnection capabilities of hardware description languages to eliminate application specific hardware.

CHAPTER 2 - PREVIOUS WORK

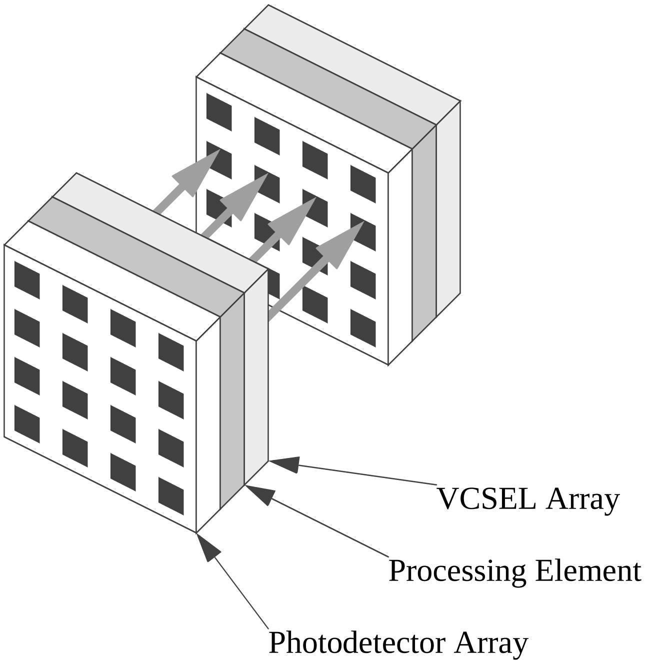
2.1 Overview

To understand the current state of technology in the areas represented in this paper, it is advantageous to examine a few systems that bear some resemblance to the 3D-OESP project. Through examination, strengths and weaknesses of each of the systems can be identified forming a basis for developing a new design

2.2 Opto-Electronic Architectures

McArdle[14] demonstrates an optically coupled system using a single smart-pixel processing element array to implement a feedback-type architecture. As shown in Figure 4, each stage in the computing system consists of three functional partitions. First, a photodetector array receives data that is external to the device. Next, the processing element array performs the actual calculations on the data. Finally, optical outputs are provided by the VCSEL array which flow into the next stage.

FIGURE 4. Smart-Pixel Parallel Optoelectronic Computing System

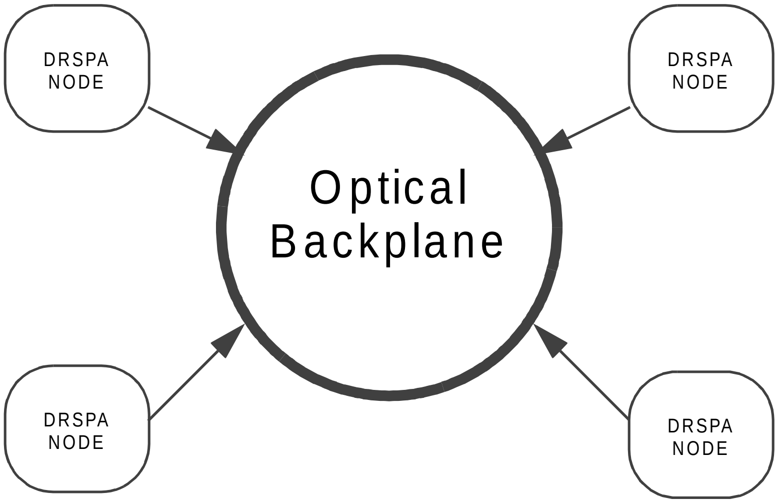


A reflection phase modulating Spatial Light Modulator(SLM) is used to display a computer generated hologram to dynamically change the interconnection topology. The operation of the PE array can be changed on each cycle, and the interconnection topology can also be periodically changed by updating the pattern on the SLM. The machine is therefore completely programmable and is capable of performing a variety of parallel processing tasks.

This system has the advantage of a pipelined architecture which is very efficient for applications that can take advantage of this design. This system also uses VCSEL matrices for communication between processors similar to 3D-OESP. However the number of optical devices is high relative to the number of processors. The additional requirement of one SLM per stage further increases the total device cost. Finally, the number of discrete devices requires considerable effort in post-fabrication alignment.

Szymanski[15] presents a terabit, free-space photonic backplane consisting of a large number of parallel reconfigurable optical channels. The parallel channels are organized as a unidirectional ring with channel access protocols implemented by smart pixel arrays.

FIGURE 5. Hyperplane Architecture



The photonic backplane interconnects 32 Printed Circuit Boards (PCB’s) and has a bisection bandwidth of 1 Tb/s, with each PCB receiving a bandwidth of 32 Gb/s. The backplane can be dynamically reconfigured to support 1024 broadcast channels at 1 Gb/s, 32 broadcast channels at 32 Gb/s, or many intermediate values. The backplane can also embed arbitrary graphs, including meshes, hypercubes, shuffles, etc. This system benefits from a high speed, high bandwidth optical channel. However, the actual processing is performed by discrete PCB’s thus reducing the performance density of the system.

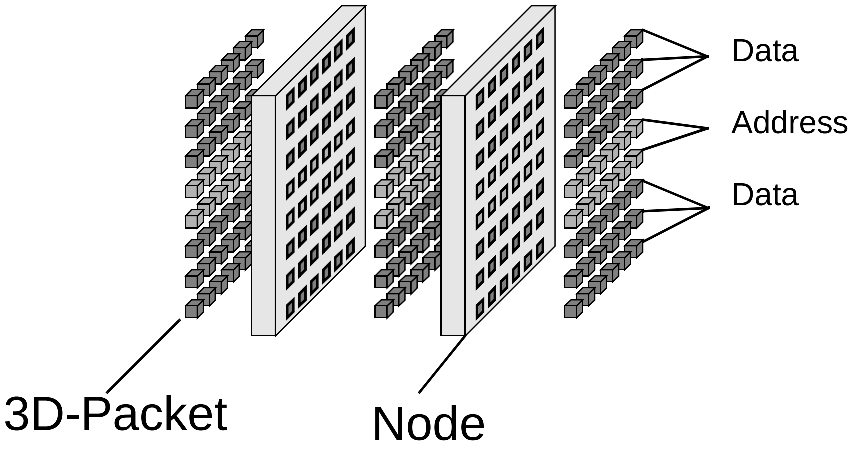
Neff[23] has developed a free-space interconnect system that functions as a board-to-board pipeline for connecting high-speed processing elements. The prototype version employs an 8-by-8 smart pixel array. Each pixel has its own driver, detector, and routing electronics. They also have limited signal processing capability implemented by a 1-bit, 20-MHz processor that can perform 16 logic functions and has 4 kb of off-chip RAM.

Routing is performed through a lenslet array and a holographic optical interconnect element placed over each VCSEL photometer. A four-level hologram containing 64 individual phase holograms (one per VCSEL) diffract the VCSEL beams. Although both smart pixel arrays are identical, the hologram implements a 2-D, nonseparable perfect shuffle interconnection scheme that permits any two channels in the array to talk to each other. This system has potential for simple vector processing applications, however it is limited by the speed and capability of the processing elements. The addition of the holographic routing technique increases the total system cost.

Wu[16] has developed a networking scheme for transmitting three dimensional optical packets. The system uses a Carrier-Sense Multiple-Access with Collision Detection (CSMA/CD) protocol modified for operation over ring networks to transmit Optical Parallel Data Packets (OPDP's). Through the use of the parallel packets, higher data throughput is achieved.

The optical path is provided through the use of a 5x8 array of VCSEL's with accompanying support logic. The ODPD that operates on this channel contains a data payload, address information and a transmission clock. Synchronous transmission is accomplished by aligning data with the optical clock. This allows high speed communication between systems with no external synchronization.

FIGURE 6. TRANSPAR Network Communication



This system shows considerable promise. The parallel optical backbone combined with optical clock results in a very robust design. However, like the other designs, the number of optics per processing element is high.

2.3 Analysis

Each of these systems shows a high degree of functionality and a common limitation, that of a high number of optical stages per processing element. These stages add to the cost through increased fabrication complexity and additional alignment requirements. To strive to minimize cost while keeping functionality, the optics need to be centralized into one unit. If possible this unit should strive to minimize or eliminate the need for specialized routing optics. Thus a high processor to optics ratio can be achieved.

CHAPTER 3 - THEORETICAL BACKGROUND

3.1 VCSEL Device Concepts

The underlying principles that photo-luminescent devices are based upon were developed by Albert Einstein around 1916. His work was based on particles known as photons that could be generated from atoms in a medium under specific conditions. This release of photons is termed spontaneous emission.

The process by which spontaneous emission occurs begins with atoms in the medium at rest or in the ground state. Energy is then added to the medium through in the application of heat or electrical energy, raising the electrical state for some atoms. This process is known as pumping. The excited state is somewhat unstable and will, therefore, allow the atom to return to the ground state. However, to return to this lower energy state, the atom must first lose its excess charge. This is achieved through the emission of a photon.

This process operates continuously in a photo-luminescent device with the resulting photons producing a very efficient light source. This is the process widely used for the creation of light emitting diodes or LED's. Unfortunately, this method of luminance is random and incoherent and thus is not useful for applications such as optical communications. For more uniform light source, we must look further into Einstein's theory.

3.1.1 Light Amplification and Stimulated Emission

Once the basic principles were developed, Einstein examined the possibility of interaction between the excited atoms. He theorized that a photon striking an excited atom would release another photon that was essentially a duplicate of the first. This collision is known as stimulated emission. By being identical, the two photons would have the same energy (hence wavelength), polarity and phase. The result would be a method of generating a coherent light source.

Since the possibility of one photon striking another is relatively small, to generate the reaction in any usable quantity, one needs a large quantity of excited atoms and photons. This is accomplished through the use of two mirrors placed directly opposed to each other. The photons generated in the medium will reflect off of each of the mirrors, striking atoms through successive passes until sufficient energy is obtained to escape through the partially reflective mirror. This process of photon generation is known as light amplification through stimulated emission. Light resulting from this process is coherent, polarized and within sharply defined wavelengths; the characteristics that constitute laser emission.

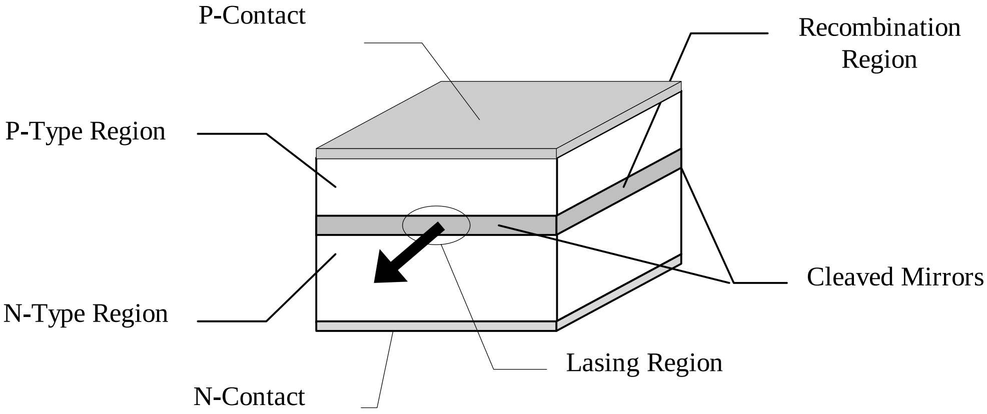
3.1.2 Semiconductor Lasers

In 1953, von Neumann proposed the idea that stimulated emission could be achieved through carrier injection across a P-N junction.[2] This theory opened new doors for the design of high quality laser devices, but was impractical for many years due the stringent technological requirements that were inherent in the fabrication of the device. To fabricate injection lasers properly, a technique must be used that provides a high degree manufacturing accuracy combined with low cost. With the advent of the computer revolution, semiconductor fabrication techniques evolved to the point where injection lasers became both feasible and cost effective.

An injection laser is shown in Figure 7. It operates by applying a forward bias to a P-N junction. This results in the injection of electrons and holes into the active region where they recombine, emitting photons. The energy of the emitted photons is approximately equal to the semiconductor bandgap.

Once the photons are emitted, they must be contained in an appropriate optical feedback system. In a semiconductor device, this is usually accomplished through cleaving the device on a grain boundary. If properly formed, the interface will function as a mirror, reflecting the photons back into the cavity.

FIGURE 7. Illustration of a Semiconductor Injection Laser

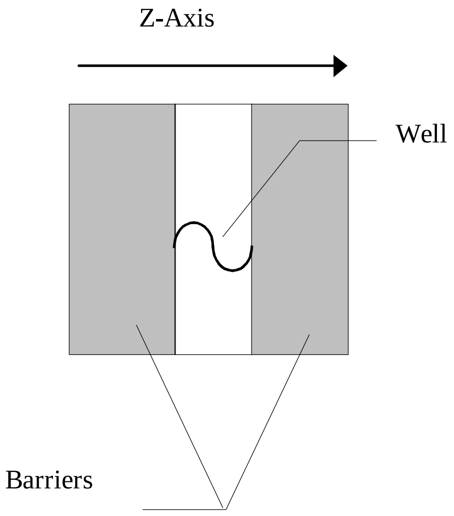


3.1.3 Quantum Confined Semiconductor Lasers

A technique that is used to increase the efficiency of semiconductor laser is the application of quantum structures. Quantum confinement occurs when atomic waveforms are trapped in structures that have dimensions comparable to the De Broglie wavelength of the electron itself.[3] The presence of the confinement structure changes the possible energy states achievable by the electron, thus creating an electronic resonance cavity with a limited number of possible solutions. These cavities can increase the coherence of the confined particles for a given energy input, thus increasing the effective output of the laser device.

Figure 8 shows a diagram of a two dimensional quantum structure, commonly referred to as a quantum well. This type of structure can be extended to one and three dimensional structures termed quantum wires and boxes, respectively. Each additional dimension of the quantum structure increases the coherence of the confined particles.

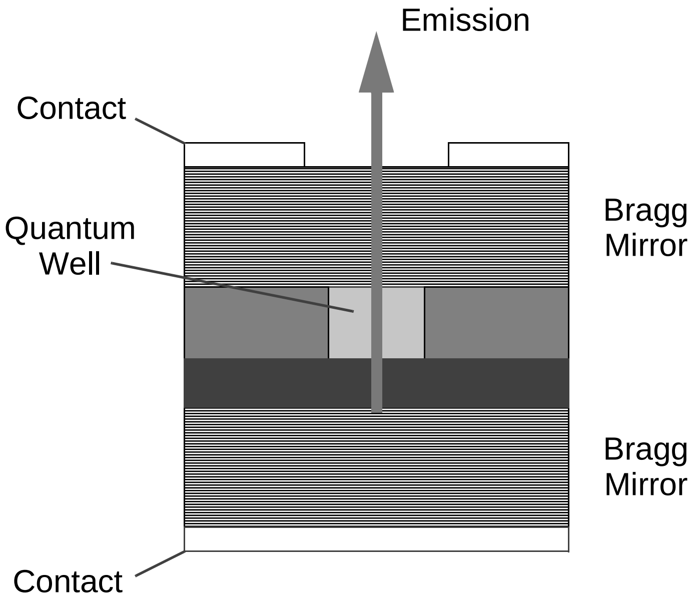
FIGURE 8. Quantum Well Structure



3.1.4 VCSEL's

VCSEL's (Vertical Cavity Surface Emitting LASERS) were first proposed by Professor Kenichi Iga at the Tokyo Institute of Technology during the late 1970s. They are semiconductor lasers fabricated using the GaAs/AlGaAs/InGaAs material system. As shown in Figure 9, the vertical laser cavities are created by surrounding the quantum wells with alternating layers of epitaxially grown film to form Bragg mirrors. The layer thicknesses in the mirror design define the emission wavelength of the laser. Like their name suggests, these devices operate in mode such that laser emission is perpendicular to the plane of the substrate.

FIGURE 9. Vertical Cavity Surface Emitting Laser (VCSEL)

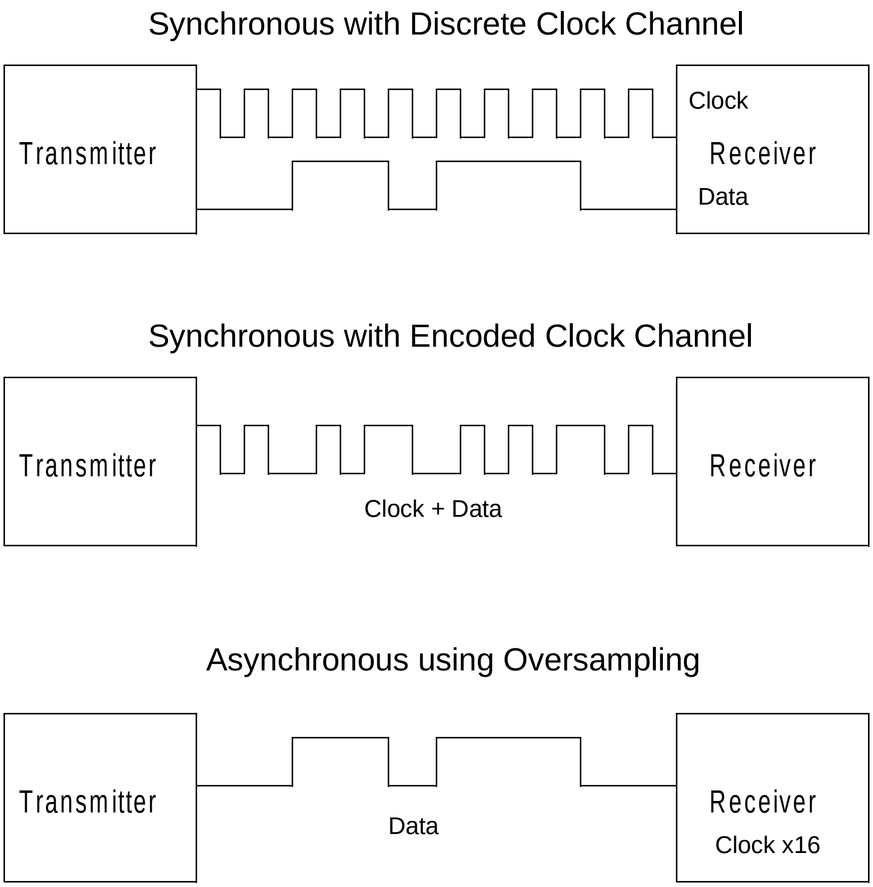


The key difference between the vertical-cavity and the conventional in-plane edge-emitting laser is that the VCSEL can be entirely fabricated and tested at the wafer level, while the edge-emitter is only completed once the wafer is cleaved into individual devices, thereby forming facet mirrors. Thus, the handling involved in VCSEL fabrication is much less than other designs. The laser active region of a VCSEL is typically one wavelength thick, implying that VCSEL’s always operate in the fundamental longitudinal mode. Current VCSEL are capable of transmitting data at 6 Gb/s with a potential for much greater speeds.[3][8]

3.2 Digital Transmission

Work in the area of data clock synchronization during data transmission tends to focus on recovery of clocks from embedded data streams through decoding or oversampling. These approaches require the data being transmitted to either be encoded with the transmitter clock signal, to be transmitted with a separate clock on another channel or the receiver to operate at a higher clock rate than the source, as shown in Figure 10.

FIGURE 10. Traditional Transmission Techniques

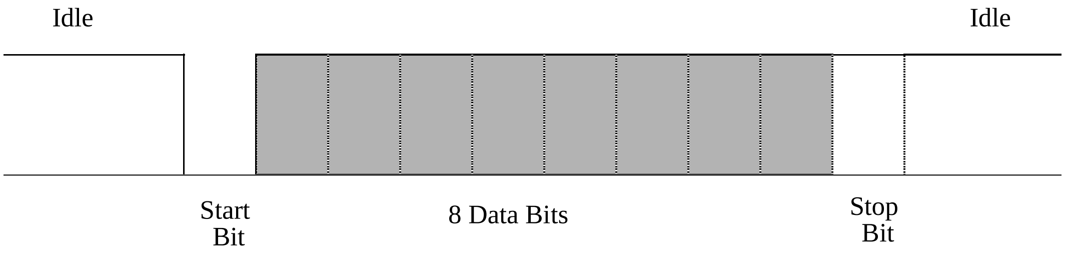

Synchronous with Discrete Clock Channel
Transmitter
Clock
Receiver
Data
Synchronous with Encoded Clock Channel
Transmitter
Receiver
Clock + Data
Asynchronous using Oversampling
Transmitter
Receiver
Data
Clock x16

3.2.1 Asynchronous Transmission

Figure 11 shows an asynchronous serial data frame format. As the name implies, data frames transmitted using this technique arrive asynchronously in relationship to the remote system. The term asynchronous is relative since only the frames arrive in this manner. The data within each packet is arranged synchronously in relationship to the clock signal generated from the start bit on the remote side for each packet.

A data frame consists of three components; the start bit, data, and a stop bit. The start bit consists of a pulse for one baud cycle of an assertion level that is the electrical opposite of the idle level. The purpose of this bit is to serve as a marker for the beginning of a data frame. A change from the idle state is used by the remote system to determine that a new data frame is being transmitted

FIGURE 11. Asynchronous Serial Data Frame



Next in the frame is the actual data. It consists of 7 or 8 bits of binary data. The orientation can be either most significant bit first(MSB) or least significant bit first (LSB) depending on the serial discipline being used.

Finally, the stop bit is transmitted. This designates the end of the data frame by asserting the communication line to the same level as the idle level. The receiving unit can determine a valid frame by verifying that the levels for the start and stop bits are correct. If this is not true, then a frame error is detected and the data is determined to be invalid.

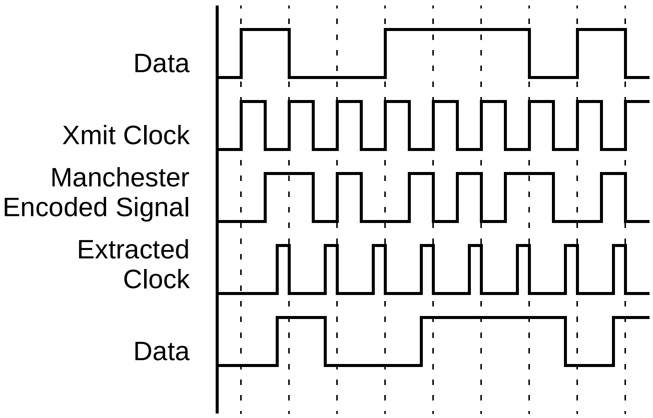
3.2.2 Limitations of Asynchronous Transmission

It is important for the remote system to be synchronized to the speed of the transmitter to properly decode the data stream being received. Even a small variance can affect the frame decode to such a degree that information transmitted will be rendered useless. To resolve the synchronization problem, the remote system oversamples the input typically by a factor of 16, reconstructing the clock on the receiving end. This oversampling technique is difficult to achieve in systems with extremely high system clock rates.

3.2.3 Synchronous Transmission

Synchronous transmission overcomes the limitations of asynchronous transmission. It is characterized by the transmission of a clocking signal simultaneously with the data. This is accomplished in one of two ways. First, the clock is transmitted via a separate channel that is received with the data. This clock signal is used to drive the sequential logic in the receiver. This method is the easiest to decode, but requires a separate communications channel.

FIGURE 12. Manchester Clock Encoding

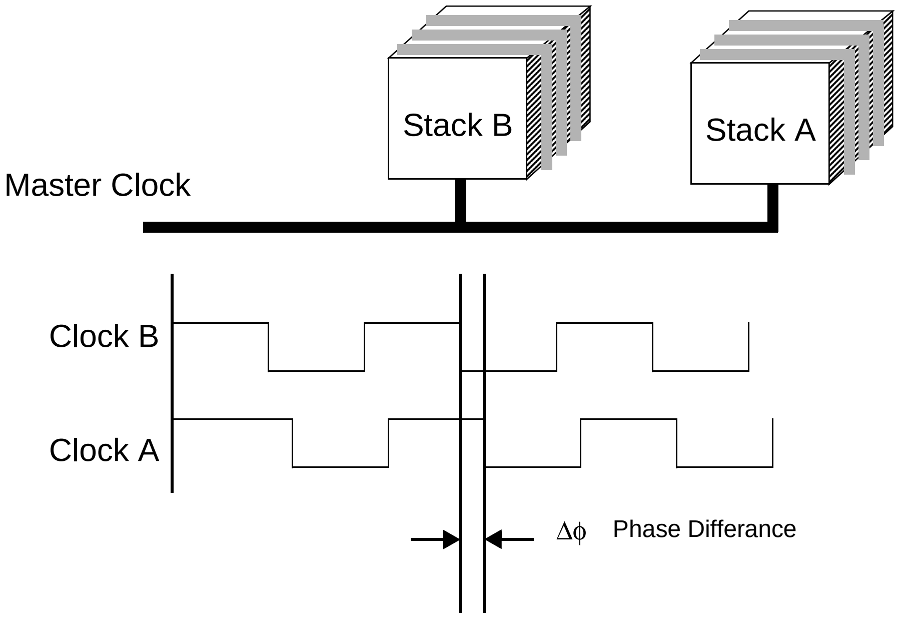


Second, the clock signal can be encoded with the data. Figure 12 shows a data stream that is encoded using the Manchester technique. This approach substitutes a ’10’ and ’01’ signal for the 1 and 0 data transitions. Through the addition of this data, the spectrum of the transmission has an additional regular frequency component added. This signal can be locked onto with a Digital Phase Locked Loop or similar device. Using an encoded clock allows the data and clock to be sent over one channel. However, the effective data rate of the communications channel increases.

3.2.4 3D-OESP Approach

For fully unsynchronized systems with disjoint clocks, these are the only options available. However, the unique environment of the 3D-OESP project provides another possible solution. Each of the stacks is driven by the same external clock source, as shown in Figure 13. Because of this, the signal that reaches each stack will be synchronous with respect to frequency, but some phase shift will exist. To achieve fully synchronous transmission, a method to determine the phase shift must be devised.

FIGURE 13. Clock Distribution Skew



The function to determine the phase difference must operate one step per clock cycle. Due to the high speed of the clock, oversampling cannot be used. It must measure the skew in a manner similar to the operation of the receiver for any meaningful calculation to be performed.

CHAPTER 4 - DEVELOPMENT

4.1 Introduction

Developing systems involving opto-electronic components can present a number of challenges to the hardware designer. There are a multitude of issues that must be addressed when exchanging data between low speed electrical buses to the higher speed optical environment. These hybrid architectures are further complicated when hierarchal design strategies are applied.

In an attempt to reduce opto-electronic multiprocessor design complexities, concepts from related fields have been applied. Among these fields, network architecture bears the greatest similarity. Many networking concepts have been applied successfully in the area of parallel processing[2] and optoelectronic systems[5][6]. It is obvious that computer and network topologies hold a number of similarities.

4.1.1 Impetus

The device technologies used in the optical area exist mainly in small production runs; therefore, their price is considerable relative to other technologies. To maximize capital resources during the development cycle, an attempt must be made to reduce the number of prototypes required. This requires extensive simulation of designs that are developed by using sound operational concepts.

In order to accomplish these goals within the time constraints set for a project, rapid prototyping techniques also need to be applied. But these techniques only speed the actual implementation of the design. Other techniques need to be utilized to quickly develop the architectural specifications.

4.2 Mapping Functional Specification to Network Topology

The first step to in applying this approach is to find an appropriate real world model for the proposed system and map these concepts to an architectural specification. The critical decision involved in applying the network topological concepts is the selection of an appropriate model. This is accomplished through a careful examination of the functional specification.

4.3 The 3D-OESP Architecture

The 3D Opto-Electronic Stacked Processor project consists of the development of a custom system architecture based around vertical chip stacking technology. This technology combines individual VLSI die in a three-dimensional fashion to form a cubic device. These stacks are then used with optical interconnects to create a high-speed, high-density computational platform[21].

The architecture needs to support the calculation of two dimensional FFT's and other digital signal processing functions using VCSEL based free-space optical interconnects for communication between planes (die) and stacks[22]. Major architectural points are the development of reliable high speed serial link and an inter-processor communication protocol. This protocol must account for the unique requirements of the free-space optical interconnects while maximizing the available bandwidth[8]. One such requirement is the need to incorporate many simultaneous serial communication links in order to avoid the problems of synchronization and skew across bits in a parallel word.

Inter-stack communication will be accomplished through the design of a robust full-duplex optical link centered around two-dimensional VCSEL-detector arrays[20]. This link should be self sufficient, containing all necessary logic required for operation including electrical/optical domain transfer circuitry[7].

Figure 14 shows a block diagram of the proposed system. The three major functional blocks are the processor stack, opto-electronic arrays and the crossbar stack. This particular configuration consists of 16 dies in each of the two stacks. Computation is performed in the processor stack, with each plane having four general purpose CPU's. Communication between dies is accomplished through the opto-electronic arrays. Data is transmitted to the crossbar stack, which is rotated 90° relative to the processor stack. This orientation allows the data to be routed to a different part of the optoelectronic array for retransmission to the processors.

FIGURE 14. Block Diagram for the 3D-OESP System

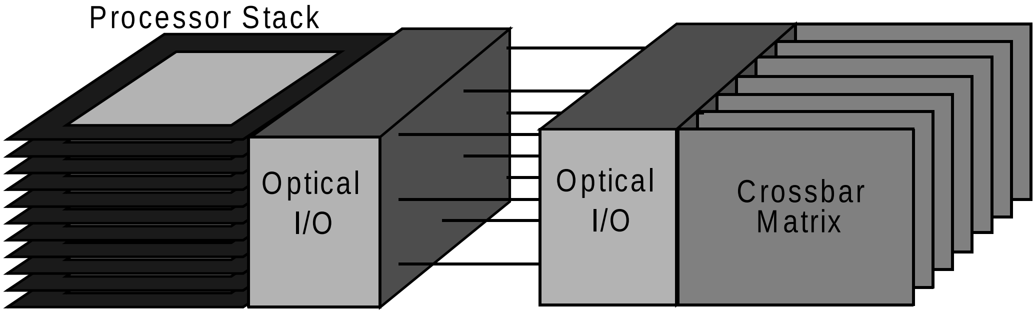
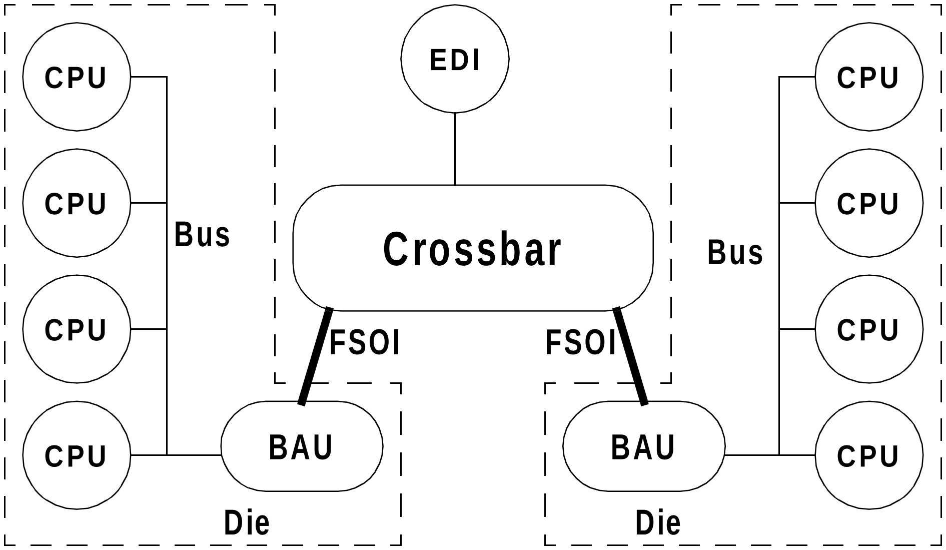


Figure 15 shows a simplified block diagram of the 3D-OESP project implementing a pair of processor and crossbar dies. In the architecture, the CPU's are the source and destinations for all packet traffic. The Bus Arbitration Units(BAU's) provide byte level synchronization and allocate bus access between CPU's. The connection between the CPU's and BAU's is a local tri-state parallel data bus. To provide global communications, a queuing crossbar is included to determine the destination BAU for incoming packets and retransmit to that location. BAU-to-crossbar interconnection is provided by a matrix of free-space optical interconnects(FSOI). Finally, the External Data Interface(EDI) provides an electrical connection for transmission of data to and from the 3D-OESP system.

FIGURE 15. Simplified 3D-OESP Topology

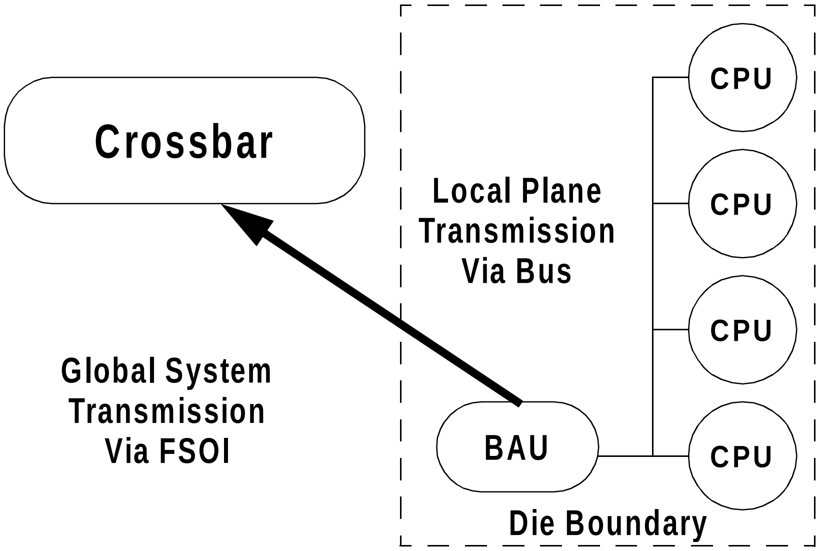


4.3.1 Functional Inferences

Using the 3D-OESP specifications, a few simple comparisons can be made. The CPU's function as endpoints for data generated on a tri-state bus. Thus they would correspond to network nodes transmitting on a broadcast bus. Data can be transmitted locally to the plane or to CPU's that reside on different planes within the system.

Transmission to a different plane would require that the data be forwarded to a higher level component that has knowledge of the physical relationships between the planes. The forwarding concept is used for transmitting between subnets in a network environment. Since subnets are local in nature, this concept could be applied stating the boundaries for a subnet and transmissions within a plane. Forwarding of packets, shown in Figure 16, is provided through the BAU, which is a function traditionally performed by a network router.

FIGURE 16. Applying Subnet Concepts to the Plane


Crossbar
Local Plane Transmission Via Bus
CPU
CPU
CPU
CPU
Global System Transmission Via FSOI
BAU
Die Boundary

Global data in the system is transmitted by the free-space optical link to a central crossbar. The optical link has a much higher bandwidth capacity due to a higher clock rate than the local parallel buses and is functionally different in its operation. It obviously requires some type of data conversion and multiplexing to make maximum use of its capacity. In large networks, high-speed, global fiber connections, termed “backbones” are used to handle the large amount of traffic between subnets. Since we have established that each of the planes roughly corresponds to a subnet, it follows that the global connection between these could be represented by a optical fiber backbone.

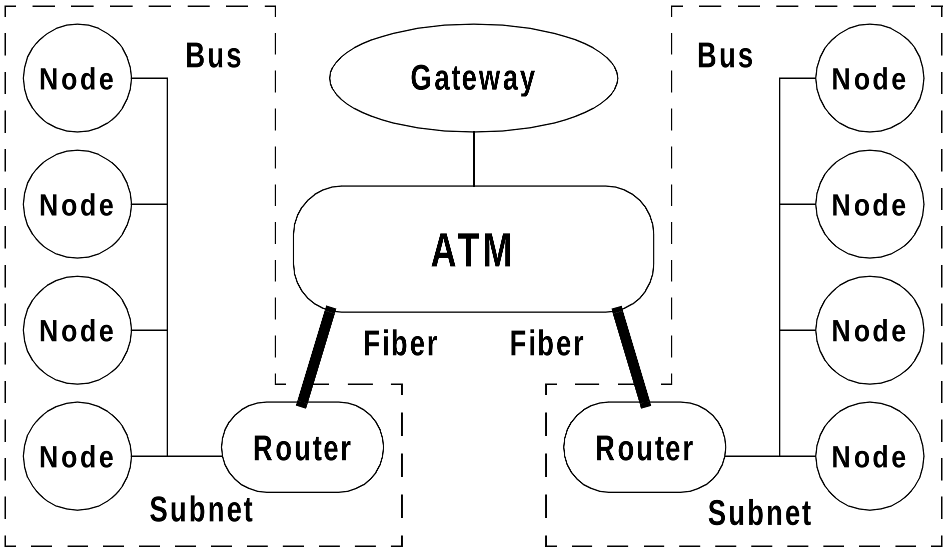
The job of routing global information in the 3D-OESP system is performed by a central crossbar. This device receives information from each of the planes, determines its destination plane and retransmits it on the appropriate optical channel. To perform this task, it must have knowledge of the locations of each of the planes and their associated address ranges. Based on this information, an appropriate model would be an Asynchronous Transfer Mode switch. This device is designed to take network data from multiple input channels, and reroute it to destination ports based on the encapsulated address information.

The final portion of the 3D-OESP architecture to be examined is the External Data Interface (EDI). This is a communications interface that provides a link to systems that are not directly part of the system architecture. Its main tasks are to provide the initialization data for processing and to download the resulting data after computation.

Assuming that we have successfully created a network model for the 3D-OESP architecture as a whole, it would follow that the EDI could be considered a network gateway. This gateway would provide access to the systems external to the basic architecture.

Figure 17 shows an equivalent network topology based on the initial analysis, substituting an appropriate network component for each functional block in the architecture specification. In the model, the CPU’s are represented as traditional network nodes. Each of the nodes is connected on a common bus to form a subnet. Subnets are connected globally through high-speed fiber channels to a central ATM switch. To provide local traffic isolation, each subnet is provided with a router to discriminate between packets based on their destination address. A gateway is provided for external access to the network.

FIGURE 17. Equivalent Network Model



4.4 Application of Topological Concepts

Once an appropriate model is selected, the specifications for the architecture are developed. As decisions concerning functional trade-offs arise, the model is examined to see how it implements a solution. By mapping this to the architectural specification, solutions will become apparent for many problems.

4.4.1 Data Transmission Methodology

The first decision to be examined under the network model is that of a method of transmission throughout the system. Since transmissions on a broadcast network require a destination address, some form of binding is necessary to associate the address information with its data. Networks accomplish this through the use of a structured transmission packet. This packet format specifies the relationship of the address information to the data being transmitted.

Next, a transmission protocol must be determined. This can be provided through the establishment of a virtual circuit or by connectionless packet transmission. Virtual circuits have overhead relating to the creation and destruction of the communication path. They are ideal for extended unicast transmissions by the same source and destination. However, for short bursty traffic, set-up and tear down overhead can be significant.

Connectionless transmissions require the address overhead on each packet and are therefore, inefficient for extended data transmissions. However, they do not incur the initialization overhead of the virtual circuit.

The applications targeted for the 3D-OESP project are of a distributed processing nature. This means that data will be ordered, yet bursty, similar to ethernet traffic. The model in this case dictates a connectionless, packet oriented approach.

FIGURE 18. Packet Transmission Format

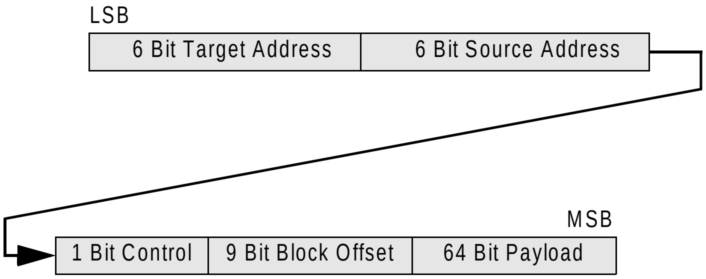


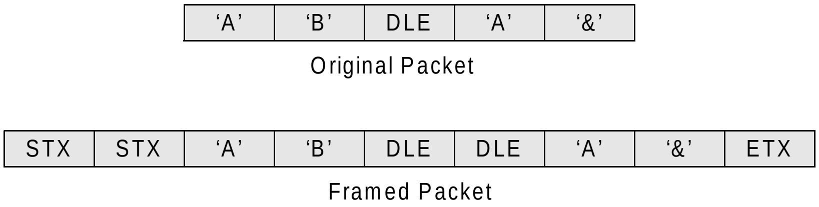
Figure 18 shows a layout of the packet format used in the 3D-OESP specification. The packet transmits a 64-bit payload. With the number of CPU’s in the full specification, 6-bits of address are needed to fully identify a destination. Along with the destination, the source address, offset and mode bits are also specified. This information is mainly used by the destination CPU’s and could therefore be considered to be data. By specifying it in the system level packet, this information would be available for future expansion at a later time.

4.4.2 Packet Framing

Once a packet format has been established, the problem of framing becomes paramount. Data that will be transmitted over a high-speed serial channel must provide some method of explicitly specifying the beginning and end of each of the packets. This framing information should be sufficient to be positively identified in the presence of errors, while not providing significant overhead to the packet.

Packet oriented transmission is classified as synchronous in nature and provides for two approaches to providing frame synchronization. Character synchronization specifies data boundaries through the embedding of STX-ETX (Start Transmission, End Transmission) characters in the data stream to identify the boundaries of the data. To provide data transparency, STX-ETX characters that appear the in the data packet are preceded, as shown in Figure 19, with a DLE (Data Link Escape) character to differentiate them from the packet boundaries.

FIGURE 19. Character Synchronization



This concept can also be applied at a lower level through the use of bit synchronization. Figure 20 shows a technique known as bit stuffing that uses a consecutive string of 1’s or 0’s as a frame boundary sequence. The length can be sized for greatest transmission efficiency as the packet is considered a bit stream and character boundaries are ignored. If the data stream contains a sequence of 1’s or 0’s that is equal in length to the frame sequence, a 0 or 1 is inserted to break the run. In this way data transparency is achieved.

FIGURE 20. Bit Synchronization

110010101100011111011111

Original Packet

01111110**11001010110001111**0**101111**0**1**01111110

Framed Packet

Either approach is acceptable in the network model that we have created; therefore, the decision should be based on the factors relevant to the architecture. Since the actual nature of the data will vary from each application, some flexibility will be desired. Because of this, the bit stuffing scheme is preferred since it handles the transmission of binary data or printable characters with similar efficiency.

4.4.3 Arbitration and Collision Avoidance

Since we have several processors that have need of the local bus, some type of access protocol is necessary to prevent collisions. This protocol must be robust, easy to implement and minimal in resource costs. The three main types of network access protocols are Token Ring, Carrier Sense Multiple Access with Collision Detection (CSMA/CD) and Token Bus.

Using a token ring approach, a token would be transmitted to each CPU. Once received, that CPU would have access to the bus. Data is transmitted sequentially from each CPU in the ring until it is forwarded or reaches its ultimate destination. Token ring provides the best utilization of the bus, but it would require considerable VLSI real estate for the routing of the ring.

CSMA/CD would allow each CPU to transmit on the bus as necessary. The CPU would then listen to its transmission to see if another CPU transmitted simultaneously, destroying its data. If a collision was detected, each CPU would wait a random length of time and retransmit. CSMA/CA is easily implemented through the use of an open-collector interface for each of the CPU's. However, it only allows up to 20%-40% of maximum bandwidth before collisions become a problem. Since the parallel bus is the probable bottleneck in the design, this would be inadvisable[10].

Token bus provides the high speed of the token ring with the routing simplicity of a broadcast protocol. It transmits the token to each of the CPU's and data appears on the central bus. Token bus also provides excellent throughput, achieving 80% bandwidth utilization of a communications channel[11].

For the given architecture, token bus appears to hold the best performance. However, a full implementation is not required because of the close proximity of the CPU's. The concept can be distilled to its basic concepts, a broadcast based protocol with a central arbitration authority. In the modified protocol, the BAU would serve as a bus arbiter. Through control lines, it would signal to each of the processors when they have access to the bus.

4.4.4 Optical Backbone

In the network model, a traditional broadcast connection is implemented for all local communications within the subnet. Global communications are accomplished over a high-speed fiber backbone. While not capable of transmitting the aggregate of all subnets, it does provide a higher capacity channel. This channel is capable of processing the additional addressing overhead that is required.

A single plane would use a fraction of the FSOI throughput. To take full advantage of the higher bandwidth of the optical channels, multiplexing is used to combine these low speed data paths. Time domain multiplexing over multiple channels is well suited for free-space optical interconnects due to their digital nature, high switching speed and relative immunity to channel crosstalk

Since addressing information is encapsulated into the global packets when they pass through the local router, the CPU's must know something of the topology of the network. This requires a processor that is larger and more application specific. The result is that a general purpose CPU would need to be modified to work in this architecture.

4.4.5 Routing

To achieve the lowest transmission latency possible, it is desirable to have all network nodes on a common broadcast bus.[12] However, this can drastically reduce performance in hierarchical topologies. In these systems it becomes necessary to filter irrelevant data, preventing it from being needlessly retransmitted.

To achieve this selective retransmission, the components within the system must have some knowledge of the address ranges for each of the transmission channels. In a network environment, routers perform this function, reducing overall traffic by intelligent filtering of packets. Transmissions with local addresses are destined for nodes within the local sub-net and are prevented from joining global backbone traffic.

In the network model, the ATM knows of the existence of each of the subnets, but nothing of the addressing scheme within them. Therefore data is routed to the subnet with the appropriate address range. Since each subnet is broadcast based, the data will appear on the bus allowing the appropriate CPU to recognize its respective data.

Applying this to the system architecture, the crossbar would have knowledge of the address ranges for each of the planes. Data that flows through it would be sent to the appropriate optical channel. But some mechanism must exist to prevent local data from being transmitted to the crossbar.

The BAU performs this filtering function locally. If a packet appears on the bus that has an address that does not correspond to the allowed range within that plane, it will be forwarded to the crossbar for redirection. However, when a packet is transmitted that has a local destination, it appears only within the plane.

4.4.6 Node Address Assignment

One of the key problems in the 3D-OESP architecture was the determination of how CPU's addresses were assigned. Two approaches are used in a network environment, hardwired addressing and software update.

Hardwired addresses have the advantage of simplicity. Network addresses are assigned by manually configuring the adapter. This is a good approach for a static network topology, but in order to implement a dynamic architecture, a more robust approach is required.

Software-updated addresses allow a node to obtain its network address during some initialization phase. The address is received from some central network authority after an initial query. By allowing a system to start in an unknown state, local configuration is not necessary.

Either approach is supported by the model. The software approach is preferable since it allows each processor to be general purpose. However, software addresses are far more complex to create. Ethernet networks use a central authority to assign node addresses during adapter initialization using the BOOTP protocol. Thus the configuration, within certain limitations, doesn’t require hardware changes.

Using the centralized approach, an address server of some description must exist on the network. This is accomplished in a number of levels. First, the central authority for the subnets would obviously be the ATM switch. Due to its location, it would be able to provide high order addressing bits, roughly analogous to a subnet mask. The second level of authority must reside within the subnet. This task is performed by the BAU. With its proximity within the subnet, it can assign the final bits of addressing to each of the CPU’s.

From this, it follows that the initialization process would be two step. In the first step, the crossbar transmits an information packet to each the of the BAU’s, informing them of their respective high order addresses. The second step would involve the BAU’s transmitting the complete address to each of the processors on the local bus. At the completion of this step, each of the processors is initialized relative to its location in the bus and is capable of sending and receiving data.

4.4.7 Multicasting

One of the functions that would be required during start-up of the 3D-OESP system is the broadcast of initialization data from the EDI to all the CPU’s. Data that is transferred will be identical, differing only in address. The simplest approach is to create packet with a destination for each of the CPU’s but this uses a large amount of bandwidth, on the order of:

$$E = nmp \qquad \text{(EQ 1)}$$

where,

E = Bytes transferred through system

n = Number of planes

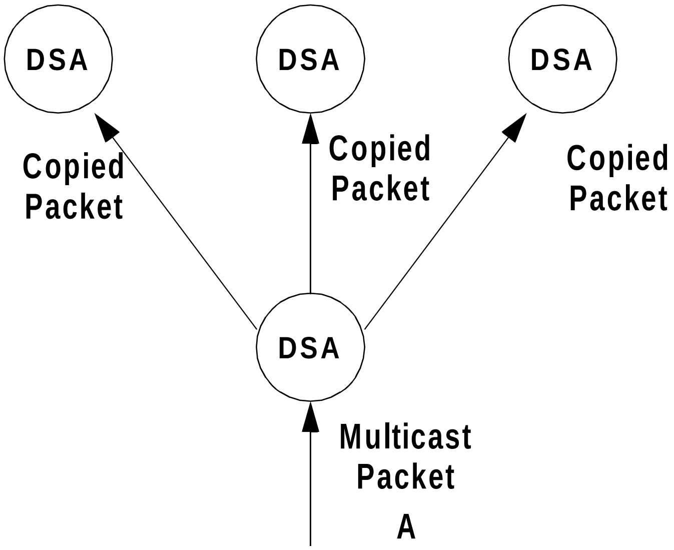
m = Number of CPU's

p = Size of packet in bytes

Given that this operation could be performed numerous times during a computational cycle, it would be desirable to reduce this overhead.

A technique employed by networks to deals with packets of this nature is known as multicasting is shown in Figure 21. This allows a packet to be transmitted to a group address. The packets are received by Directory Service Agents (DSA's), which attempt to determine the destination for each of the packets. If the local DSA doesn't have enough information to determine an address, it forwards a copy of the packet to all DSA's that could provide more information.

FIGURE 21. Multicasting Data Flow



Using a multicast packet in the network model, we follow the above example. As the packet is transmitted from the EDI, it first encounters the ATM switch. Since this is a switching point, it would follow that it operates as the first level DSA. The ATM, being unable to completely decode the destination for the packet, would forward a copy of the packet to each subnet. These copies would be received by each router, which is also operating as a DSA. The local router would complete the address and place the data on the local bus for each node to receive.

In the architecture specification, the ATM and BAU would operate as the DSA for the system. Each would copy the packets and forward them to the next level in the broadcast tree. The resulting bandwidth usage for this transmission would be P at the EDI and N*P over the FSOI with no increase in processor complexity.

CHAPTER 5 - OPERATION

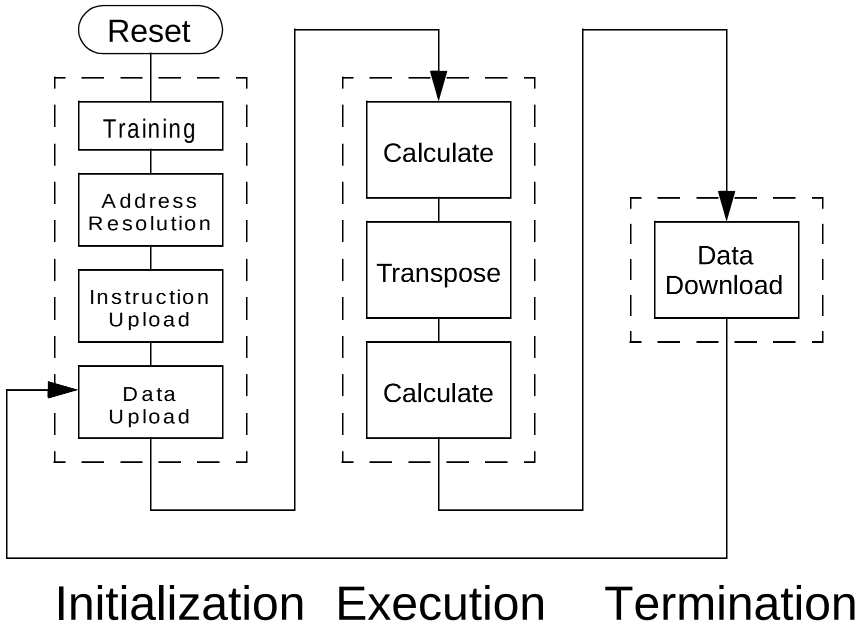
5.1 Operational Flow

The system has three different phases in its operation. Boundaries of each of these phases are not tightly defined as the they are determined by the processing units being used in the architecture and their associated application. For the purposes of this dissertation, we will assume the processing elements are general purpose computational units and the application is a high speed FFT. In the 3D-OESP project, the three phases are:

- Initialization
- Execution
- Termination

Figure 22 shows the relationship between the phases during runtime. In the initialization phase, the system is configured for the task at hand. The execution phase consists of all operations necessary to perform the actual calculations. During the termination phase, all products of the calculation are off-loaded to an external processor for evaluation or storage.

FIGURE 22. Operational Flow



5.1.1 Phase I - Initialization Sequence

Immediately after power-up, the system must perform a series of steps to initialize the processor and support hardware. The assumptions being made are that the processors are capable of asserting a ready signal once they are powered up and ready to receive data.

5.1.1.A Training

Before any data can be transmitted through the optical channels, the receivers must be synchronized with respect to the data clock. The inherent process and thermal variations that can occur in the system could be sufficient to skew the clock and cause transmission errors. Given these changes, the output of a given transistor can change greatly depending on its location within the stack. In a communication circuit this can result in errors introduced in the data stream. Therefore, steps must be taken to compensate for this variation in output.

To account for potential shift in the data clock, a technique for determining the amount of skew present and compensating for it must be implemented. The process is referred to as training. The key to the successful use of this technique is that both the transmitter and receiver have reliable clocks and that there will be no drift relative to each end of the optical channel.

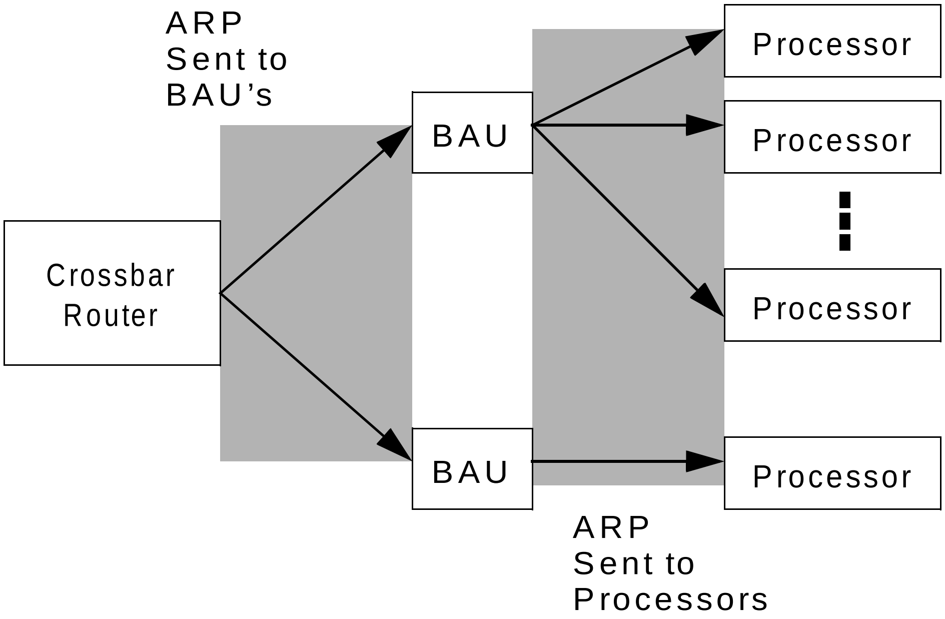
The training sequence begins by transmitting a known sequence to each of the receivers through the FSOI. Each receiver begins decoding this sequence and comparing it to the expected result. If the transmission is acceptable, that clock channel is marked as good. Next, the receivers switch to a slightly more delayed clock and receive the same sequence again. This continues until all possible clocks are tested. Once completed, the receivers examine the results and select a clock that is most distant from any transmission errors. This clock is then selected as the primary reception clock.

5.1.1.B Address Resolution

To communicate with the others in the stack, each processor must both know its name and the name of the destination. This address must exist in one of two ways; Either it is pre-existing(Hardwired) or it is told by a central authority during power-up. Hardwiring the address into each unit is both expensive and time-consuming. It does not lend itself well to a system that is flexible in design. Therefore, it was decided that an initialization protocol would be designed to inform each device of its address.

Since each processor must be wired to the BAU for reception of interrupts and each line had to have priority to prevent deadlock, it was determined that this would serve as local processor address for the die. This process, shown in Figure 23, consists of the following. Immediately after reset, the BAU will transmit a packet to a selected processor with an ID in it. This Address Resolution Packet (ARP) would be used by the processor as its source address. This process would be repeated for each processor on the die.

FIGURE 23. Address Resolution During Power-Up



An additional identification must be made with respect to the individual planes in the stack. Each must be uniquely identified for routing through the crossbar. This address must be generated external to the processor stack and reside with an authority local to the die. The crossbar matrix is connected to each die plane in a fashion similar to the BAU. However, there is no compelling reason to prioritize the connections. Regardless, this unit can assign an ID through a unique connection established via the FSOI.

After reset and until the BAU receives its plane address, it is in a hold state. It cannot process any information from the external data interface. During this same period, the crossbar will transmit a packet to each plane. These packets will be addressed to the BAU and also have the address for that plane. Once this packet is received, the BAU saves the plane address, exits the hold state. Additionally, if it is BAU 0, it instructs the EDI to begin transmitting data.

5.1.1.C Instruction/Parameters Upload

For compatibility, there is support for uploading instructions to each of the processors. By using an external program download, the system can be reconfigured dynamically for different functions. However, this step is optional for processors with embedded programs.

The architecture supports two types of control instructions that provide for this functionality. The address space for these instructions is shown in Table 1. First, the broadcast instruction(Control=1, Mode=1) will send the same packet to all processors. This is useful for synchronization instructions. Second, the processor control instruction (Control=0, Mode=1, Address=Target) allows point to point transmission of information to another processor in the system. This would be targeted for configurations that have a master processor and multiple slaves. Specifics pertaining to the nature of the instructions transmitted are processor dependent.

TABLE 1. Packet Address Bit Mapping

Unit	**Control Bit**	**Mode Bit**	**Plane**	**Address**	**Data**
Processor	0	0	XXXX	XX	FFT Data
Processor	0	1	XXXX	XX	Processor Control Instruction (Processor Dependent)
Processor	1	0	XXXX	XX	FFT Instruction/Parameters
BAU	1	0	XXXX	01	See Control Table
EDI	1	0	0000	00	Data for Off-load
Crossbar	1	1	0000	00	Broadcast data

5.1.1.D Data Upload

Once the instructions to the processors have been transferred, the actual data necessary for the calculation must be uploaded. Each packet is generated with the appropriate processor address and placed on the EDI bus. The EDI will then inject the data onto the plane 0 bus which will route it similar to normal internal traffic.

5.1.2 Phase II - Execution

Once all data has been transmitted, the executional phase can begin. This can be signals by a broadcast command that is transmitted to each of the processors to begin execution. If necessary, each of the processors can transmit an acknowledgment, depending on the application being performed. At this point the system becomes autonomous, operating independently of the EDI. Messages and data are exchanged between processors as the application requires.

5.1.2.A Transpose

The FFT transpose can be considered a component of the execution phase. Key issues relation to this part of execution are the need to provide synchronization or tolerate desynchronization within the system. As the applications can vary greatly, no suitable hardware can be designed to provide this level of control in all applications. It will be up to the processors to address the issues of deadlock.

One approach is to have each processor transmit a broadcast message to the other processors to signal that it is ready to transpose. This had the advantage of enumerating the current phase of system, but the system could enter a blocking state if reliance on these messages is enforced.

Second, a robust transposition algorithm could be developed that could perform the transpose with the processors in various states of readiness. The software overhead of this approach is much larger, however the possibility of deadlock is reduced.

5.1.3 Phase III - Termination

This phase consists of off-loading the data through the EDI. The transmission is accomplished through the use of EDI data packets (Control=1, Mode=0, Plane=0000, Target=00). These packets will be routed by the BAU to the EDI and off the system. Operation will continue with the downloading of data in the initialization phase.

5.2 Serial Data Format

This information consists bit level encoded packet consisting of a leading and trailing violation sequence to precisely mark the starting and ending boundaries for the serial word. Transmission will be synchronized due to fact that the same clock will be driving both the transmitter and receiver, removing the need for clock synthesis on the receiving end. However, some clock skew will be present. This skew will be compensated by the aforementioned training sequence

5.2.1 Error Detection and Correction

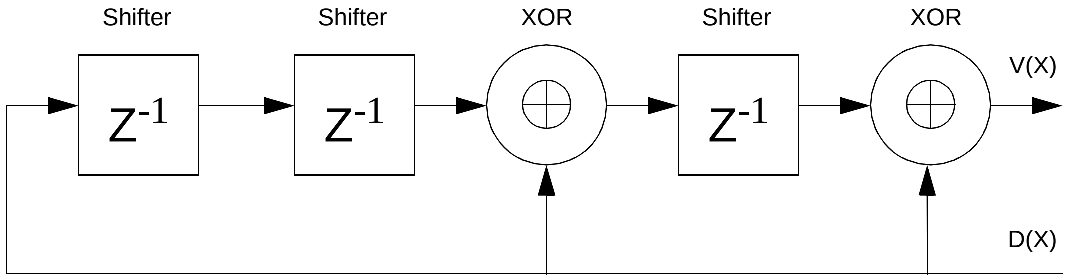
Errors existing in a transmission, coupled with the need for a reliable link, necessitate the use of Error Detection and Correction (EDC). This technique allows communication lines with less than perfect characteristics to operate as if they were an error free link.

The first step in ensuring a reliable link is error detection. Two methods are used based on the type and amount of data transmitted. For transmissions through FSOI, a high level of reliability must be maintained.

In order to reduce the complexity involved in implementing retransmission logic, redundant information is added to the outgoing data to allow the FSOI receiver to correct a limited amount of errors. This is accomplished through the use of a Cyclic Redundancy Code(CRC). CRC's are well adapted for use in this type of transmission due to the fact that they are calculated in a serial fashion. A CRC code is calculated by repeated modulo 2 multiplication of a data stream by a generator polynomial.

As shown in Figure 24 the data stream, represented by D(X), is shifted through the generator circuit. Exclusive-OR's perform the modulo 2 arithmetic, the result appearing serially as V(X). The resulting code polynomial is then transmitted to the remote receiver. Using the same generator polynomial, the receiver then decodes the received information giving the actual data. The remainder, also called the syndrome, gives the status of the transmission. A non zero value is returned when a single bit error is detected[13].

FIGURE 24. Cyclic Redundancy Code Block Diagram

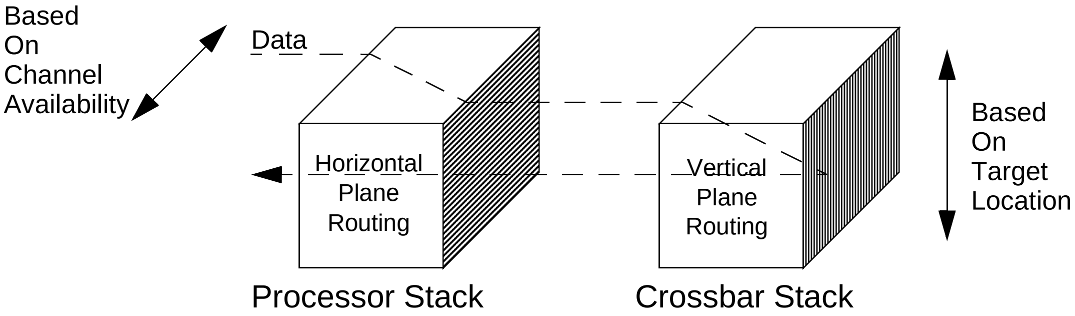


5.3 Packet Routing

The 3D OESP system employes a two stage routing algorithm divided into horizontal and vertical planes. The differentiation of these planes is due to the 90° rotation of the stacks relative to each other. To take advantage of this relationship, an asymmetric routing methodology has been adopted, shown in Figure 25. Because of this approach, movement through each plane has a different effect on the packet being routed.

The packet's horizontal plane destination is based on channel availability. Since it is possible to have multiple packets arriving on each plane simultaneously, it is very likely that there will be variations in the time to service a particular channel once the packet arrives. Given this, a simple algorithm is employed to keep track of the number of packets sent to a plane, and base a channel's future availability on this factor.

FIGURE 25. Two Stage Routing Diagram



Movement through the vertical plane selects the physical plane destination. The crossbar determines the target plane for each packet by examining the destination address contained within it. Once the target is determined, the packet is then routed to the appropriate channel. Assuming the channel is free, the packet is immediately transmitted to the destination plane. In the case of channel contention, the packet is placed in a queue to await transmission at a later time

5.4 System Scaling

There are two physical topologies for the 3D-OESP system when operating with multiple processor stacks. Figure 26 shows the system operating in a star configuration. Each processor stack is arranged around a central crossbar stack. The crossbar stack contains one crossbar die for each processor stack per plane. This dies are interconnected on each plane to facilitate communications between the processor stack. The address resolution is between crossbars is carried out on this bus, as well as runtime data transfer. Also of note is the clock phase variation between stacks. The crossbar operates on the opposite clock period as each of the processors. This provides a further level of synchronization.

FIGURE 26. System Scaling - Star Configuration

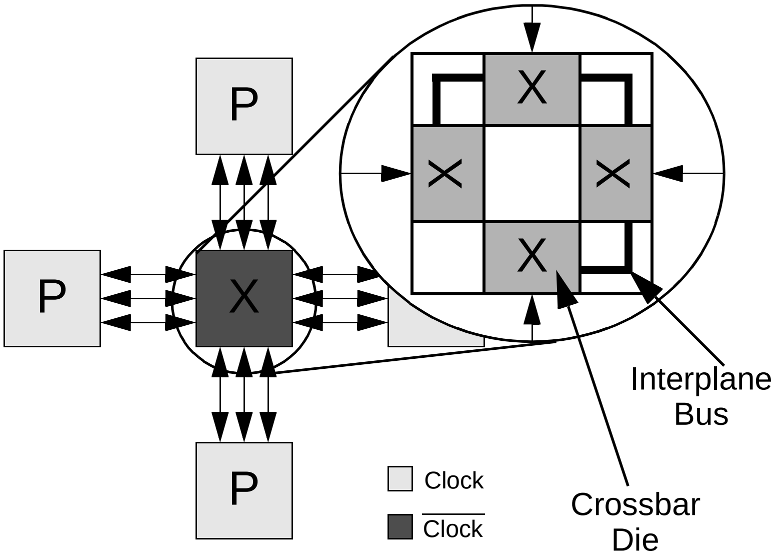
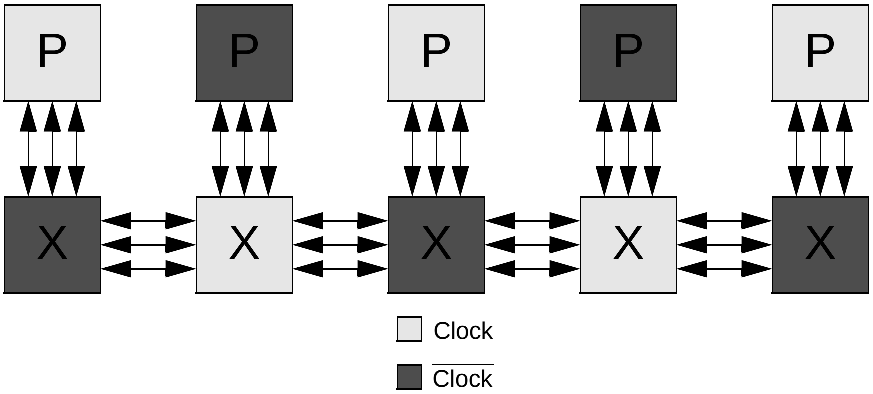


Figure 27 shows the system in a linear topology. The crossbar stack are arranged in a line with one stack per processor stack. The clock phases alternate between crossbar/processor stack pair. Each adjacent stack being opposite in phase. The primary advantage of this configuration is that is can be scale indefinitely.

FIGURE 27. System Scaling - Linear Configuration



The addressing of multiple processing stacks are supported through an extension of the power up address resolution protocol as shown in Figure 28. Operation begins by the crossbar transmitting a query packet to any potential recipient on the Least Significant Databus(LSD). The crossbar will then wait for a period of time equal to:

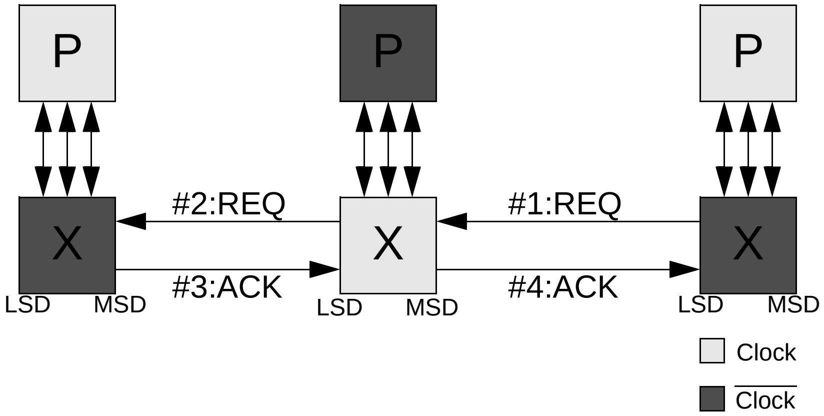
$$t(x) = 2B \qquad \text{(EQ 2)}$$

Where B is the number of bits in the address space. If during this time any requests are received on the Most Significant Databus (MSD), they are queued.

If an acknowledgment packet is received, it will contain the source address of the sending crossbar. From this, the receiving system, can determine its address. If a query packet was received from the MSB, the crossbar must transmit an acknowledgment with its address as the data.

If the time-out expires before an acknowledgment is received, such as the case of the last device on a chain, the crossbar assumes the value of the lowest address possible. If a request was received, it is acknowledged with the proper value.

FIGURE 28. Crossbar Address Resolution Flow



CHAPTER 6 - ARCHITECTURE

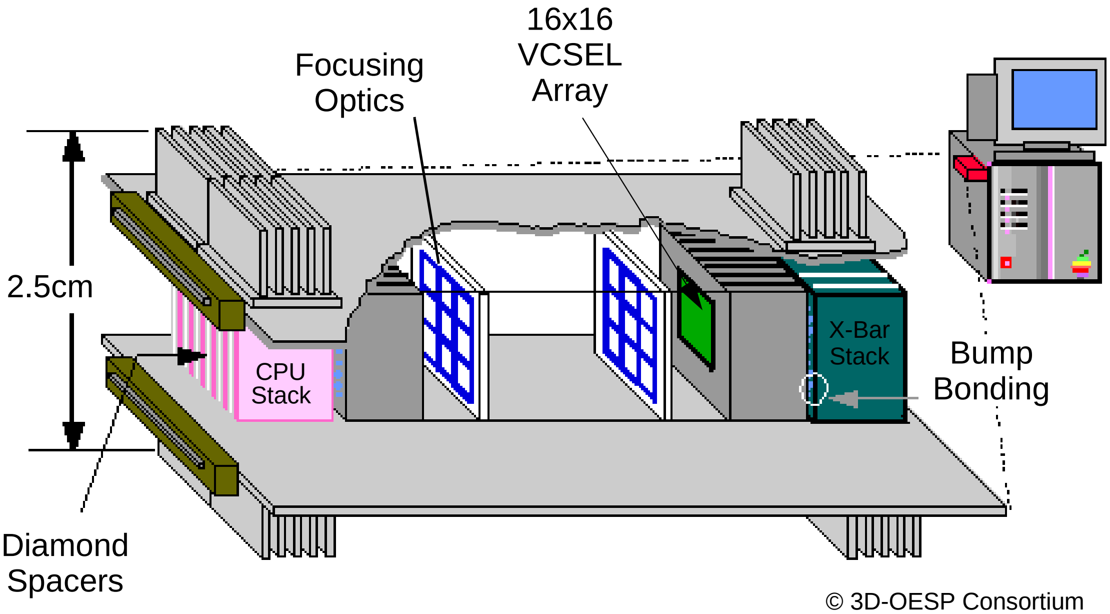
6.1 Overview

The 3D OESP system consists of the following components:

- Processors
- Bus and Associated Control Logic
- Free Space Optical Transceivers (FSOT)
- Crossbar Routing Matrix
- External Data Interface (EDI)

Figure 29 shows the system in its proposed silicon testbed. The testbench allows alignment of the optics and supplies power to each of the stacks. Additional optics are present to provide the necessary gain at the receiving stack.

FIGURE 29. 3D-OESP System Test Environment

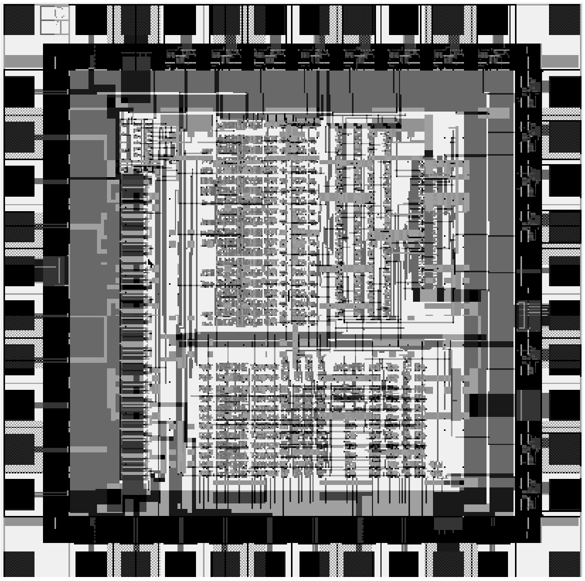


© 3D-OESP Consortium

6.2 Processors

The architecture is designed to support a number of general purpose processors. However, for the purposes of this paper the processor being used to support the application of the 3D-OESP consortium is presented. This processor, shown in Figure 30, is designed for general purpose calculations such as high speed FFT transforms, FIR and IIR filter calculations. Developed with a scalable internal architecture, this device is designed for operation in highly parallel configurations such as 3D-OESP.

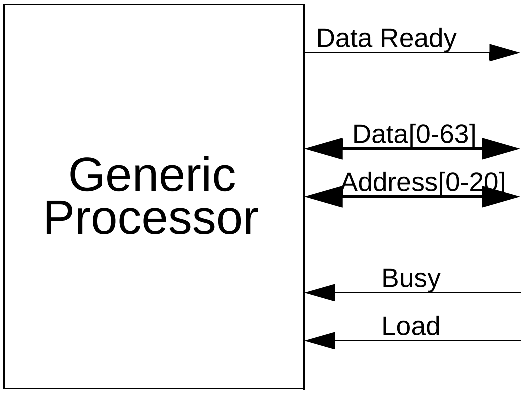
FIGURE 30. FFT Processor



6.2.1 Processor Interface

The processor communicates through a 85-bit bidirectional parallel data bus as shown in Figure 31. This bus is divided into data and address vectors. The bus supports interrupt driven I/O with bus contention being resolved externally.

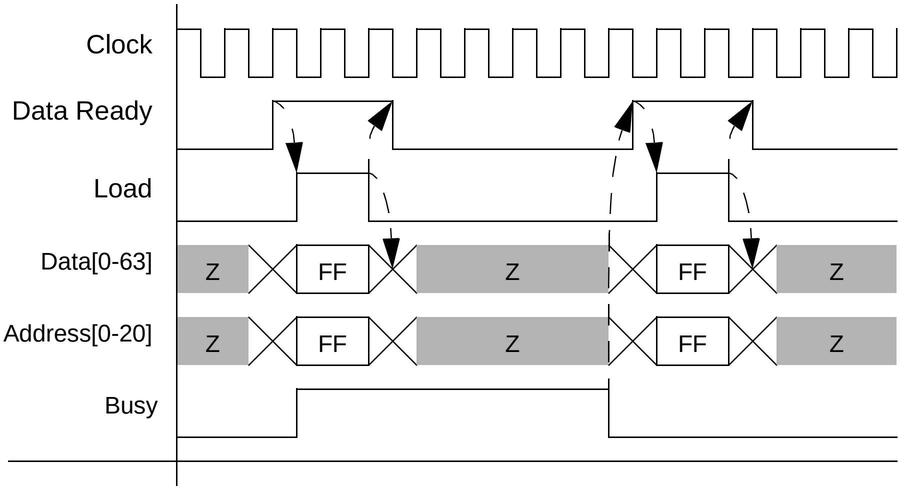
FIGURE 31. FFT Processor - Block Diagram



The following control lines, shown in Figure 32, are provided to the processor interface. The Data Ready line signals the bus that the processor is ready to send data. This line is captured by the Interrupt Request Unit and the data is buffered to prevent deadlock.

Busy is a signal generated externally to instruct the processor that the bus is busy and not to transmit data. Finally, the load signal is provided to inform the processor that valid data is present on the bus to be transferred.

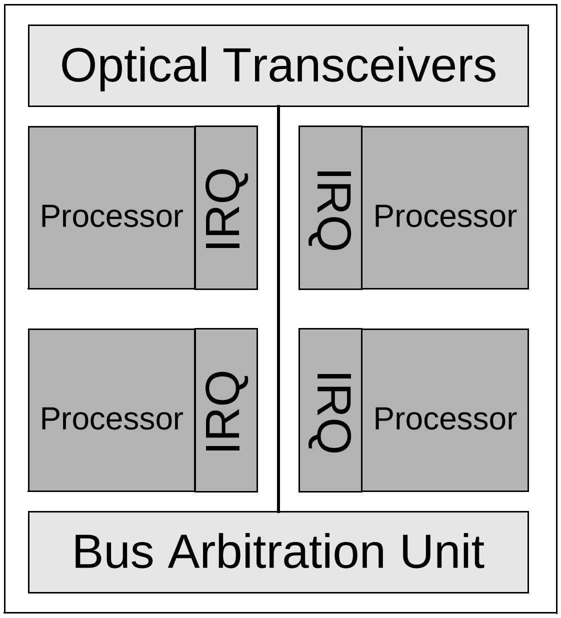
FIGURE 32. Processor Data Transfer Timing Diagram



6.3 Tri-State Bus and Bus Arbitration Unit

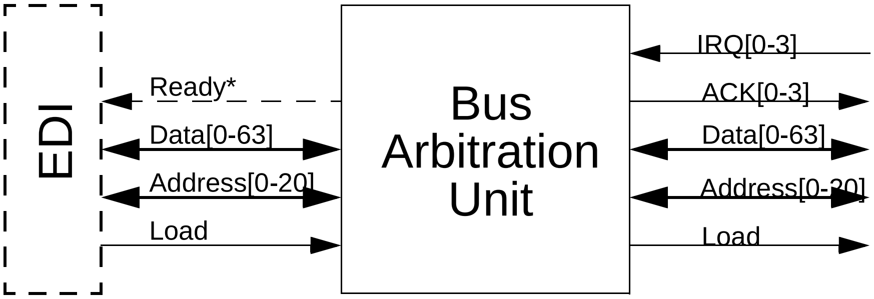
The 3D OESP system uses a centralized tri-state databus for communications through a die as shown in Figure 33. By using this centralized approach, routing area is reduced, saving time and equalizing propagation delay. Since all components connected to the bus operate independently of each other, some mechanism is required for orderly communication. The Bus Arbitration Unit performs this task.

FIGURE 33. Bus Layout



The Bus Arbitration Unit (BAU) shown in Figure 34 is the interface between the processors local to the die, external data interface and the optical interconnects. It is the central authority for communications along the central databus. The BAU receives interrupt signals from interrupt controller indicating a processor attempting to transmit.

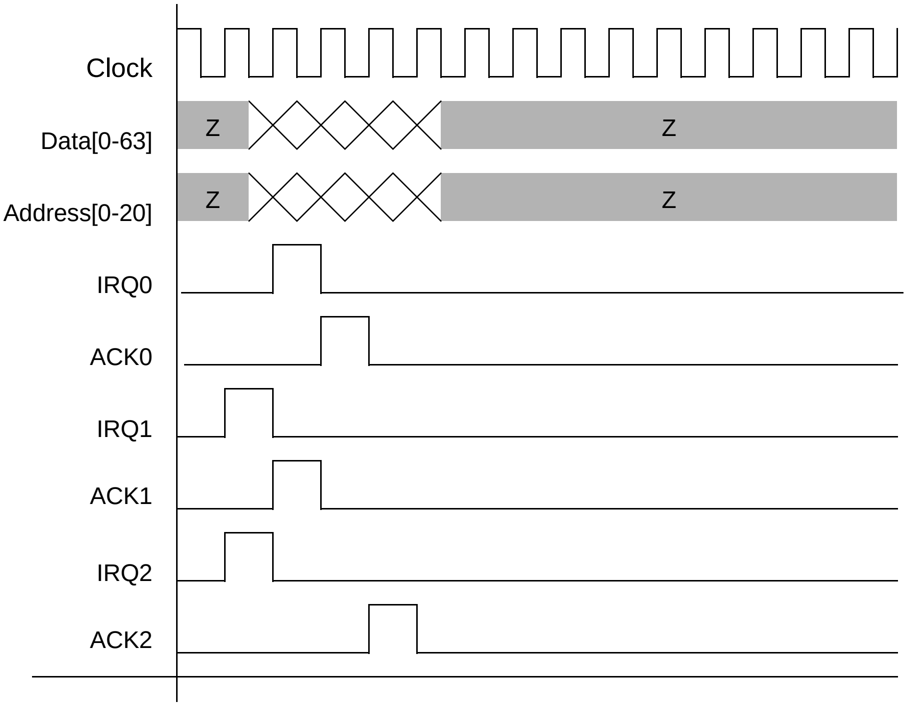
FIGURE 34. Bus Arbitration Unit - Block Diagram



*Open Collector Output

As shown in Figure 35, the priority of a request is determined relative to other requests that may be pending. A higher level interrupt received before the processing of a lower level event will result in the higher level being processed first. This is shown in Figure 35. In the current configuration, a request from the Free Space Optical Transceiver (FSOT) overrides any processor requests. Since the FSOT operates at an average of four times the data rate of any of the processors, it would necessarily be a priority request to provide optimal throughput.

FIGURE 35. Bus Transfer Timing Diagram



Once the interrupt is serviced, the BAU must determine whether the request is an on or off chip destination. This is accomplished by translating the local destination to a physical destination address internally. On chip addresses are completed by simply instructing the appropriate processor to load the data. Off chip addresses are routed to the optical interface, along with the translated destination address.

6.3.1 External Communications

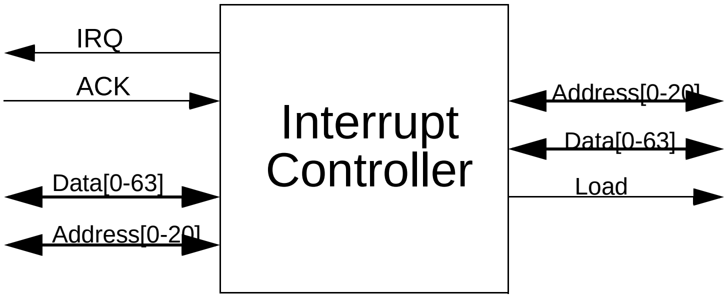
The BAU also acts as the arbiter for the External Data Interface(EDI). Communication with the EDI provides a particular challenge since it is driven by multiple asynchronous drivers. This multiple driver issue is due to the nature of the EDI connections to the processor stack. To resolve the multiple driver issue, all outputs from the BAU are open collector, allowing multiple drivers on a single line without the possibility of short circuits between fighting outputs. However, the problem of collisions still exists.

To address the collision issue, there must be a method for eliminating extraneous drivers that do not have direct communication with each other from the EDI. The problem draws a solution from the fact that each BAU must have a unique address in order for each packet to have a destination. This address is used to isolate unneeded drivers from the EDI. Using this approach, only BAU0 will communicate with the EDI. Other planes will be electrically isolated via the tri-state data and address buses with the exception of the previously mentioned Ready line.

6.4 Interrupt Controller

Since each of the processors is general purpose, it will be the job of the Interrupt Controller provide the required logic to facilitate transfers between the processor, optical interconnects and the data bus. Shown in Figure 36, its job is to queue data from the processor and route it to its respective destination. It provides the special purpose handshake logic that, in conjunction with the BAU, allows orderly sequential access to the data bus.

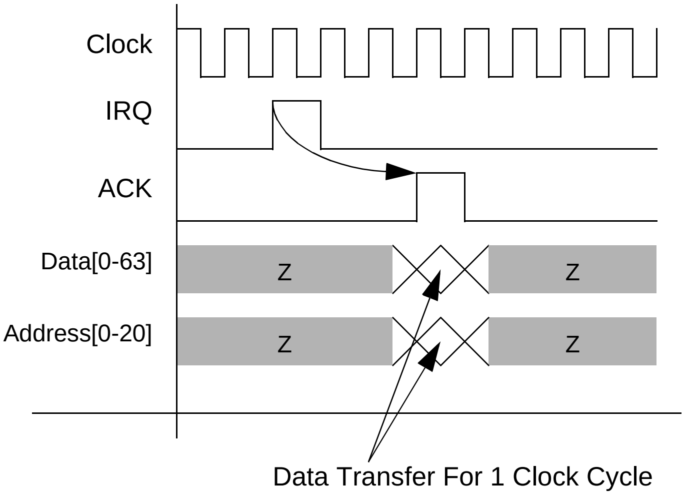
FIGURE 36. Interrupt Controller - Block Diagram



6.4.1 Operation

The Interrupt Controller intercepts the interrupt request line from the processor, as shown in Figure 37. The IC then signals the BAU to request access to the bus. If the bus is ready, then the data is transferred directly to the destination by the BAU. However, this may not always be the case.

FIGURE 37. Interrupt Controller - Timing Diagram



Since the processor may not be capable of bus transfers during the period before a request is received, a potential deadlock situation could occur if the bus was occupied with transmitting to the requesting processor. To prevent this occurrence, the processor is allowed to transmit the data to the IC for queueing. The processor is then signaled to suspend any further transmissions. Once the BAU allows bus access, the queued data is transferred to its destination and the cycle is completed.

6.5 Free Space Optical Transceiver

The Free Space Optical Transceiver operates as a full duplex, high speed, parallel communications link. Transmission is performed over a 16x16 matrix of VCSEL’s. The parallel interface permits high-speed access via traditional tri-state bus logic. However, actual transmission over the interconnects is performed through simultaneous serial transmissions. This approach overcomes two major obstacles. First is the inherent unreliability of parallel transmission over distance. Unequal propagation delay between multiple lines can cause significant delays between data lines. If this delay is of sufficient duration to interfere with the setup and hold times of connected logic, errors result.

Serial transmission does not suffer from this setback. Since each packet of data is transmitted of the same channel, any propagation delay is applied equally to all data, resulting in a reliable, but delayed transmission. However, serial transmission does require additional overhead, in the form of framing headers.

Second is the need for development of a scalable transmission link. Because of the 90° rotation of the stacks relative to each other, a change in the number of planes must be accompanied by an equal change in the number of transmission channels. Expanding the data path with a given data set width results in an incomplete utilization of the communications channel. To overcome this using a parallel transmission scheme, one must add hardware based on the specific circumstances.

The use of a serial protocol however does not have this limitation. Hardware that would already be present to determine available channels can be built scalable through use of high level design languages. A change in the number of planes would require nothing more that a corresponding change in a VHDL variable. It is this approach that is used to guarantee future scalability.

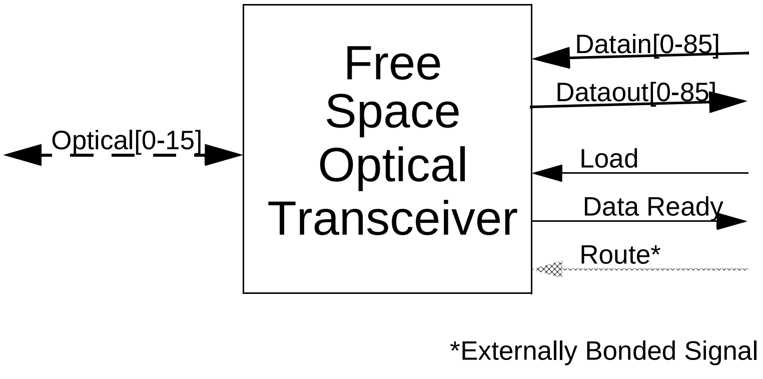
A final requirement for the FSOT is the need for ease of fabrication. Development of specific transmitter/receiver circuits requires more resources than the use of a combined approach. By creating a symmetric architecture that uses the same device on all stacks, fabrication cost is reduced through economies of scale. Die yields would be less of factor since any device can be used as a substitute for another.

6.5.1 Interface

The interface for the FSOT is shown in Figure 38. Optical[0-15] are the free space optical interconnects. The 86-bit parallel input supports the full databus width allowing the processor and associated logic to perform transfers every clock cycle. Data is transferred to and from the FSOT by the DataIn and DataOut signals respectively. DataReady signals that there is valid data on the bus for the FSOT to receive. The Load line transfers data from the FSOT to the processor bus.

Since some of the routing optimization for the transceivers varies depending on which stack it is located on, there is an external signal to designate which mode to operate in. This signal is designated Route. By pulling this line high or low, the routing function is modified. To incorporate this into the design, each of the two stacks would hardware the signal to a different level

FIGURE 38. Free Space Optical Transceiver - Block Diagram



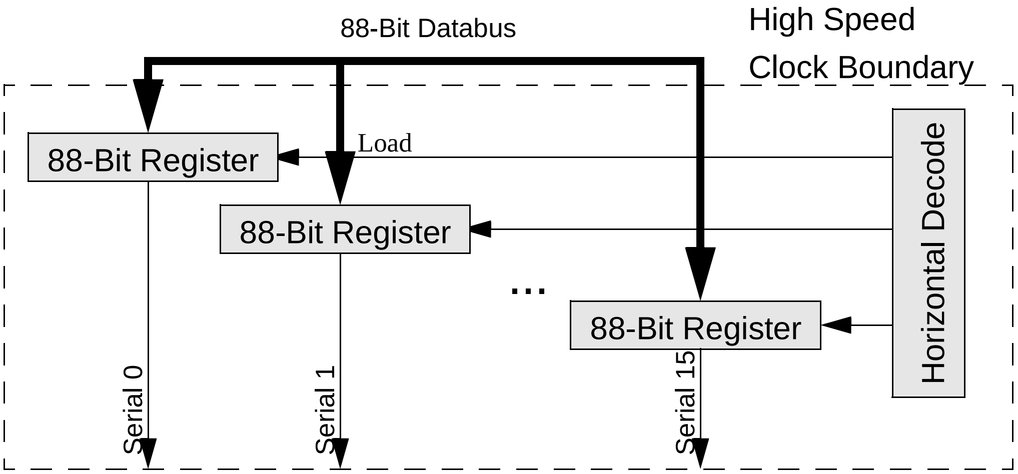
6.5.2 Horizontal Decoding

Channel availability is determined by serial conversion time. Once a packet is placed into a channel register, that register will not be able to accept data for a minimum of 88 FSOT clock cycles. This time period corresponds to 22 processor bus clocks cycles, excluding additional framing information. In the current configuration, there are 16 FSOI channels, so it can be seen that some delays will occur due to limited bandwidth.

The Free Space Optical Transceiver performs the first of a two stage routing operation when Route is asserted. As shown in Figure 39, first stage routing is performed on the horizontal plane. This routing is based on channel availability. It is determined by the Horizontal Decode Logic(HDL). The HDL module keeps track of the state of all serial channels and routes information to the current open channel. By tracking this selection, the FSOT can optimize transmission speed by routing data to currently unused planes on the destination stack.

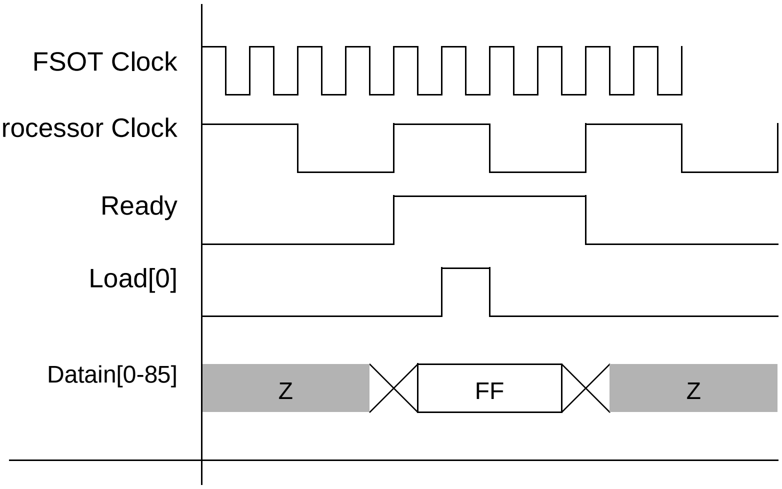
Although horizontal routing makes the maximum use of the channels available, this feature must be disabled for data transmitted back from the crossbar matrix. This is due to the fact that once a channel is selected on the processor side, there is only one possible channel available from the crossbar for the return trip. Allowing the FSOT to change channels because of availability would result in the data being transmitted to the incorrect destination.

FIGURE 39. Horizontal Channel Selection and Serial to Parallel Conversion



The transfer of data from the databus to the PISO registers crosses the boundary from the low speed processor clock to the high speed FSOT clock. The processor and associated logic need to operate at speeds that are accommodated by standard VLSI fabrication technologies. However, the FSOT is capable of higher clock frequencies. This conversion allows both of these technologies to operate at maximum performance. Since both clocks are synchronized relative to the other, problems associated with clock skew are not present assuming that a wait state is inserted in the high speed logic’s load cycle as shown in Figure 40.

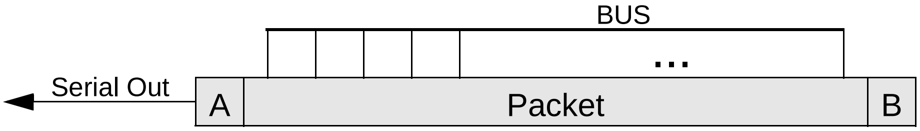
FIGURE 40. Data Transfer Across Clock Boundaries



6.5.3 Serial Encoding

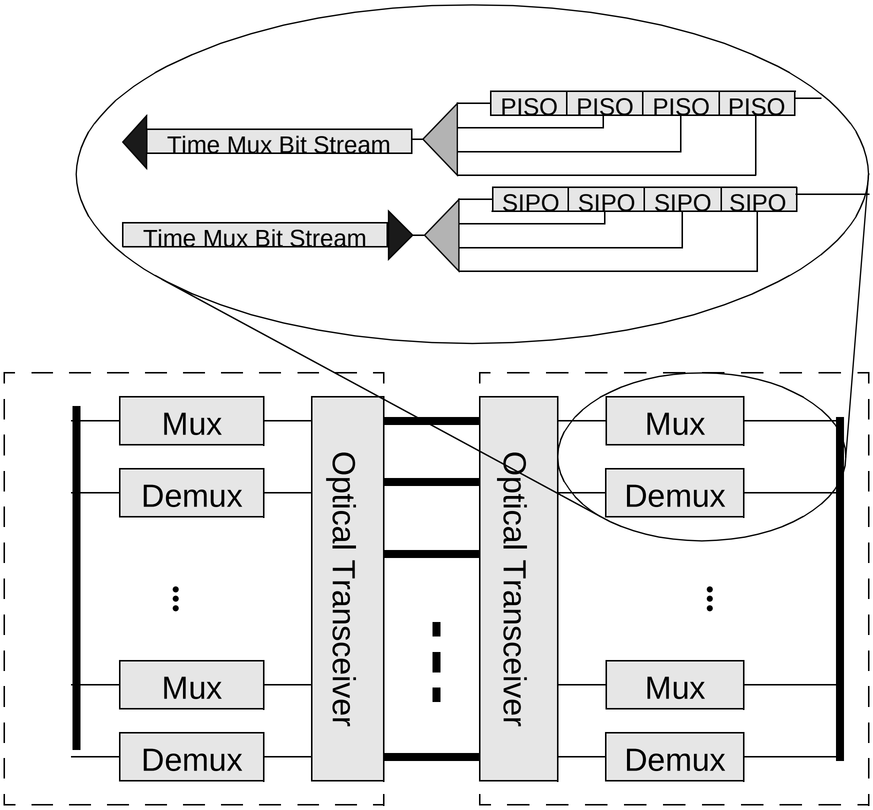
Once the data is in its proper channel, the serial encoding process begins. The actual loading of the data into the register is accompanied by the loading of the framing information as shown in Figure 41. This information is added to the data to allow precise determination of the starting and ending bits.

FIGURE 41. Serial Encoding



Once loaded, the information is shifted out of the serial output resulting in a serial datastream. This datastream is then applied to the input of an optical driver as shown in Figure 42. The driver transmits the data via an optical channel to the receiving system.

FIGURE 42. Time Domain Multiplexed Serial Data Stream



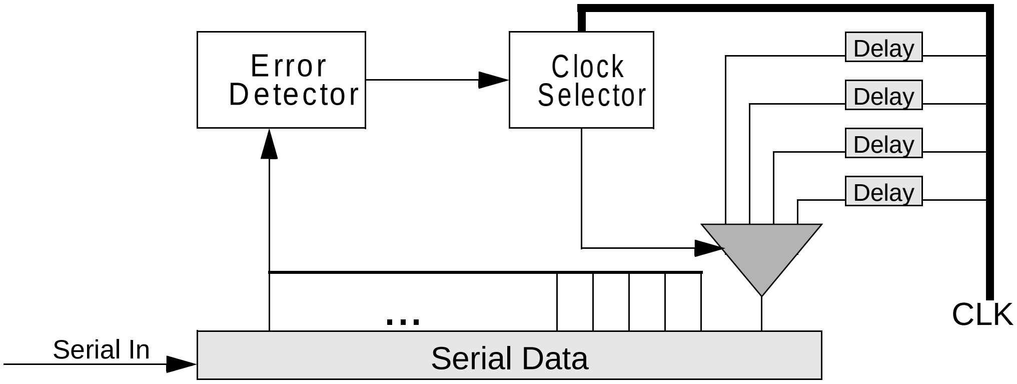
6.5.4 Transmission Clock Synchronization

A key problem that appears in the architecture is providing a clock signal on the receiver that is synchronized with the data being transmitted. Clock recovery is usually accomplished through the use of a Digital Phase Locked Loop (DPLL) or an oversampling technique. However, to achieve the data rates specified in the architecture, a different approach was necessary.

The Clock Synchronization Unit, shown in Figure 43, operates by receiving a known sequence through the FSOI channels multiple times using different clocks, each with slight delay relative to the others. Once a reliable delay is found, that clock is used to drive the receiver.

The training process is controlled by a finite state machine which operates on a clock separate from the trainer receiver. Spikes in the receiver clock during the delay selection are resolved by use of full coverage combinational logic design in the multiplexor and a latch system. Delays are provided by the use of precision 100ps delay elements available in the VLSI process.

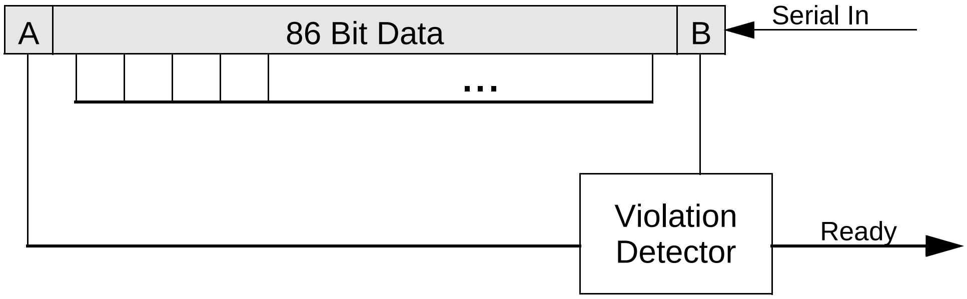
FIGURE 43. Clock Synchronization Unit (Trainer)



6.5.5 Serial Decoding

Once data is received at the other stack, the Serial Decoding conversion is begun. As shown in Figure 44, this is accomplished with a finite state machine that continually monitors the data stream looking for framing information. This information is then used to determine the validity of the data being received.

FIGURE 44. Serial Decoding



In the current architecture, the following algorithm is used. The frame detector monitors the link until a violation is received. It makes the assumption that this is the start of a header and that data will directly follow. It then resets an internal counter and begins to count the number of bits that are received. Bits inserted into the data stream to achieve data transparency are discarded as they are received.   This continues until it has received a total of 86 plus sufficient bits to be evaluated for a violation. It then tests to see if the tail of the frame is in violation. If it is, a READY signal is asserted to inform the receiving system that a valid frame had been received and has been converted. If an error is detected, then the circuit asserts an ERROR signal and begins the process again. This is summarized in Figure 45. Assuming a reasonably noise free link, the framing detection circuity can resync if a framing error occurs. However, during this realignment process, some data loss is inevitable.

Finally, the data is transmitted to the low speed bus. This transfer is initiated by the receiving bus on reception of the data ready signal from the FSOT. Synchronization from a high speed bus to the low speed bus is not an issue as long as the data is present for at least 8 FSOT clock cycles. Under the current architecture this constraint will not present a problem.

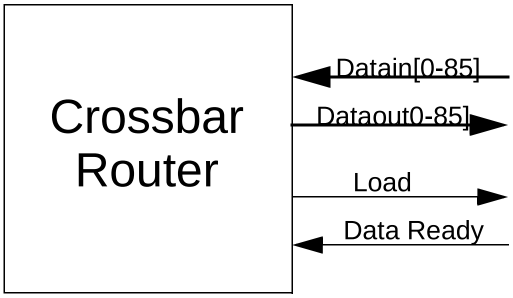
FIGURE 45. Frame Detection Algorithm

```
while(1) {
      while(! B) { //Do nothing };

      // B is true Wait for data and check header
      // WAIT 87 CLOCK CYCLES
      int loop = 0;
      while(loop++ < 86){ //Do nothing}

      if(A && B)
          assert READY;
      else
          assert ERROR;
}
```

6.6 Crossbar Router

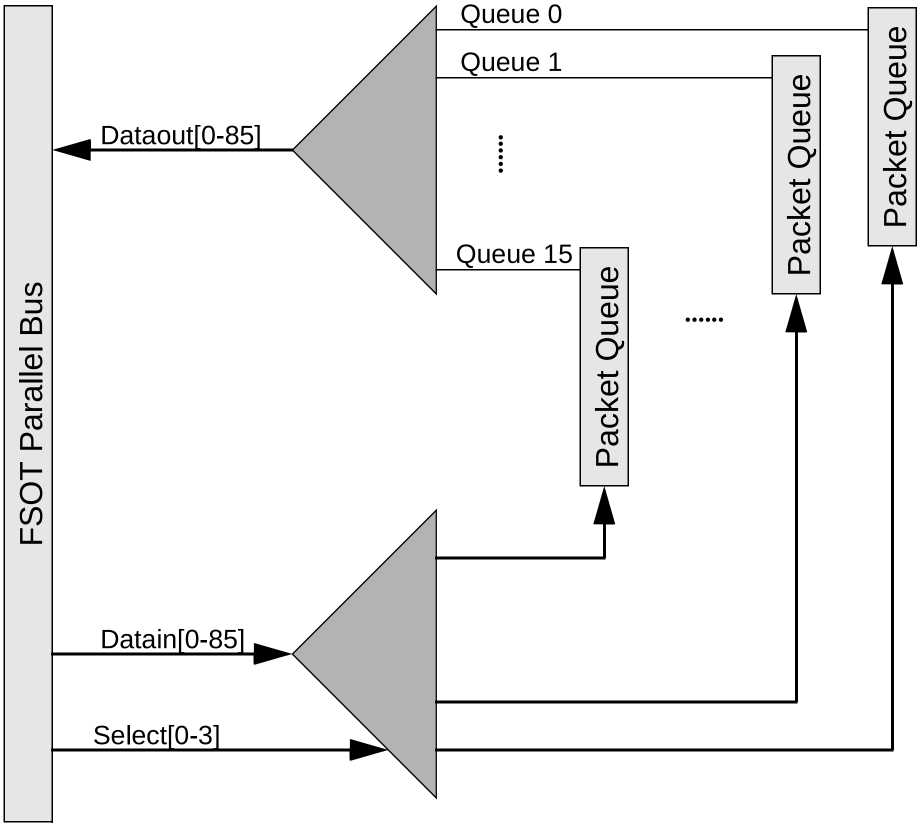
The actual routing of data from plane to plane is performed by the crossbar router which is shown in Figure 46. This logic resides in a stack rotated 90° relative to the orientation of the processor. Because of this relationship, each router can receive and transmit to each of the processor planes.

FIGURE 46. Crossbar Router - Block Diagram



The operation of the crossbar is shown in Figure 47. Once data arrives from the FSOT, the 6 bit destination address is examined to determine the desired output for the switch. If the selected port is open, then the data is placed directly back onto the FSOT and is recorded as busy for the next 22 clock cycles. However, if the destination channel is already busy, the crossbar places the data in the associated queue for that channel and continues processing. When the destination port becomes available, then the data is placed on the output bus to the FSOT.

FIGURE 47. Crossbar Router (One Plane Shown)



In the event of a packet simultaneously arriving at the same time that a previously queued packet's channel become available, the queued packet takes priority. The incoming packet is placed in the queue and the outgoing packet is directed to the FSOT. This approach guarantees that packet will arrive in with the least amount of reordering.

6.6.1 Address Resolution

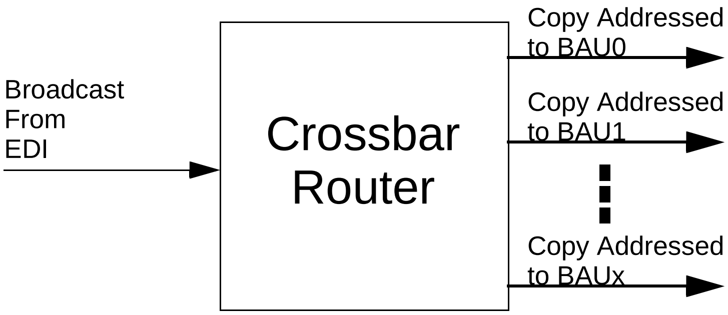
Another function of the crossbar is that of resolving transmission addresses for the planes during initialization. Since the crossbar maintains a physical link to each die, it is in the unique position of being able to discriminate between them at start up. This knowledge must be provided to the BAU on each die in order for them to process on/off chip packets properly.

To accomplish this task, immediately after reset the crossbar transmits a control packet to each of the planes. This packet contains no data, just the destination address of the die being transmitted to. It is then received by the BAU and the plane address information is saved for later use. Once all BAU's in the system have received this information, the system is ready to begin processing information.

6.6.2 Broadcast Resolution

One final function of the crossbar is resolving final addresses for broadcast packets. On reception of a packet addressed to itself, the crossbar creates multiple copies and retransmits them to each of the BAU's as shown in Figure 48. Each BAU would repeat the process, transmitting a packet to each of the processors in its node. Using this approach, only one initial packet is necessary to communicate with all the components within the system.

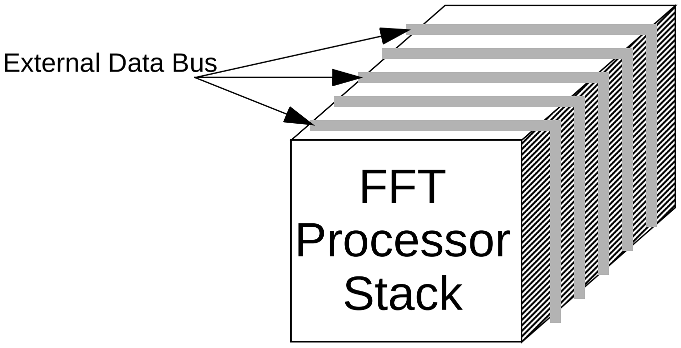
FIGURE 48. Resolution of Broadcast Packets



6.7 External Data Interface

The transmission of data external to the system is handled by the External Data Interface. This interface exists as a number of electrical connections that surround each of the stacks as shown in Figure 49. These common connections injects data to the BAU. This information is then routed to the appropriate target based on the address information stored in the packet. This connector exists in the system for testing and verification purposes. In a fully closed system, the external data connector would not be used. Data would be supplied by the optical interconnects.

FIGURE 49. External Data Interface



The following signals shown in Table 2 are routed externally. The CLOCK signal provides the necessary synchronization for all sequential devices within the system. LOAD indicates that an external device has placed data on the EDI bus. It expects that the data will be captured on the next clock cycle. POWERUP is an internally generated signal connected to the BAU's informing the outside that initialization has been performed and the system is ready to receive data.

TABLE 2. External Data Interface

Pins	Name	Description
1	CLOCK	External Clock
1	LOAD	EDI Data on Bus
1	POWERUP	System is ready for data
1	READY	BAU Data on Bus
86	DATA[0-85]	Bidirectional Data bus
4	TEST[0-3]	External Test Pins
X	PWR/GND	Power/Ground as needed

READY is an indication that the system has data on the bus destined for the external system. DATA is the bidirectional bus that all data is transferred on. TEST is a series of pins present to support runtime testing such as the implementation of JTAG boundary scan hardware. Finally, the number of PWR and GND pins are currently unspecified and will be dictated by the power requirements.

CHAPTER 7 - RESULTS

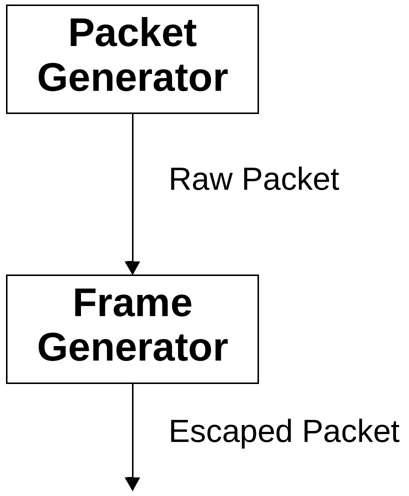
7.1 Simulation Results

Upon completion of the architectural specification, VHDL and C++ models were created for each of the major functional components within the system. The VHDL models were created under Synopsys, using both behavioral and structural code to describe the components. Simulations on this code were performed at the Link and Network OSI layers to explore the feasibility of the 3D-OESP design.

7.1.1 Link Level Simulation

This lowest level simulation performed was created in C++ to examine the efficiency of the FSOI link for different packet payload lengths. Figure 50 shows a block diagram of the simulation model. The packet generator used the C++ rand() library routine to create a string of pseudorandom data in user specified lengths. Randomly generated addresses and control information where added to fully model the packet previously shown in Figure 18. The data was then passed to a bit stuffing algorithm that created the necessary escape sequences to properly frame the data. At this point, the finalized packet possessed the encoding that would occur in the FSOI serial datastream.

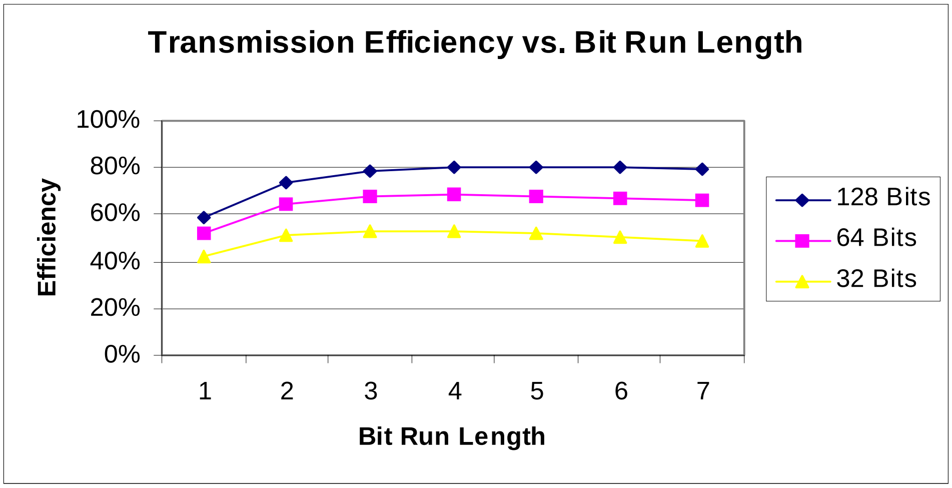
FIGURE 50. Framing Simulation Block Diagram



On completing the framing process, the encoded packet’s size was compared to its original raw form. The result of the comparison was averaged over a large number of simulation cycles. The simulation was repeated with varying packet and bit run lengths.

Figure 51 shows the results of 50 MB of data transmitted through the system. In this graph a theoretical maximum of 100% efficiency would occur with a frame packet equal in length to its original raw size (0% overhead). Simulations showed a trend of increased efficiency for larger packet sizes. Peak efficiency was shown at bit run lengths of 3 to 4, depending on packet size. From this result, it was determined that a 32-bit payload would result in an unacceptable level of throughput. 128 and 64 bit payloads showed far better results with 69%-80% framing efficiency. This translates to an effective channel rate of 690-800 Mb/s assuming 1 Gb/s raw throughput and packet sizes of either 64 or 128 bits. The trade-off between 128 bit payloads relative to VLSI real estate has not been determined at this time and must be examined to fully weigh this decision.

FIGURE 51. Framing Efficiency



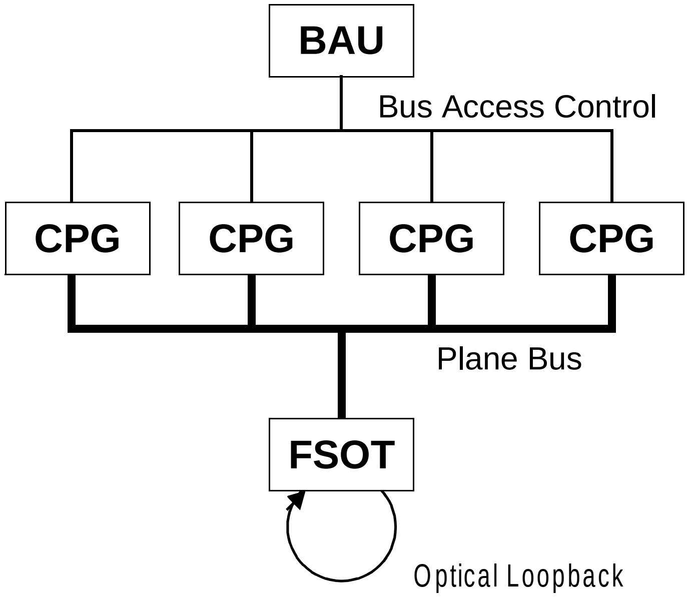
7.1.2 Network Level Flow

This next level of simulation was performed at the network level under VHDL. Its purpose was to examine the flow of traffic through the channels at the packet level. Of particular interest was the efficiency of the bus arbitration technique. This simulation utilized pseudo-random data created by four CPU Packet Generators(CPG), a BAU and one Free-Space Optical Transceiver(FSOT) shown in Figure 52.

Packets were transmitted in a pseudorandom fashion through the use of a Linear Feedback Shift Register driving a counter. This created a staggered time interval for each source and allowed the CPG's to generate packets in a manner similar to random traffic. The BAU functioned to allow orderly access to the bus and to direct packets with off chip addresses to the FSOT.

The FSOT was minimal implementation, operating basically as a loopback connection. It received data from the parallel bus, converted it to serial format and transmitted back to itself. To resolve destinations for off chip transmissions, all addresses where stripped of their high order bits (modulo 4). When the packet was received, it would be routed to a CPG that corresponded to the modified address.

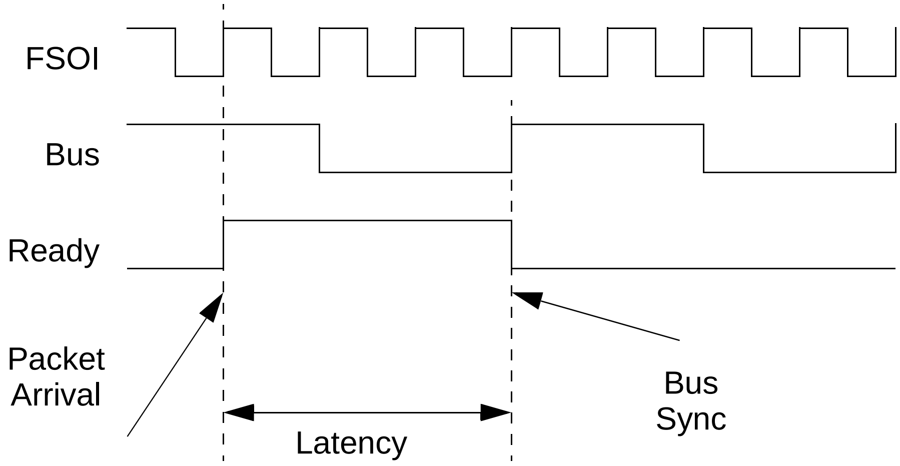
FIGURE 52. Network Simulation Model



The simulation showed that the system performed with approximately 5% throughput loss due to clock synchronization latency. Delays associated with the bus arbitration protocol were not present due to the pipelined request technique used by the BAU.

Figure 53 shows an example of the clock latency within the system. To take advantage of its higher bandwidth, the free space optical channel operates at a higher clock frequency than the local bus. Data that travels through the optical channel arrives at the local bus relative to the FSOI clock. Since there is a clock differential, the bus clock can take up to four FSOI clock cycles before it synchronizes and removes the data from the optical buffer.

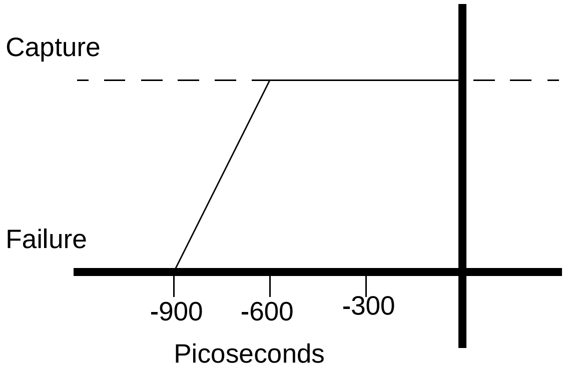
FIGURE 53. Bus Synchronization Latency



7.1.3 Trainer Simulation

Simulation of the clock synchronization trainer showed expected results. As shown in Figure 54, within minor variations of the clock, the receiver was capable of correctly identifying the training dataset and selecting an acceptable clock skew. The falloff curve is approximate because of the limitations of behavioral simulations. The actual curve will vary based on the device parameters of the target VLSI process being used. Timing analysis showed a potential to operate in excess of 300 MHz.

FIGURE 54. Clock Capture at 300MHz



CHAPTER 8 - CONCLUSIONS

Table 3 shows the performance numbers for the current system architecture. The performance of the FSOI as a communications conduit appears to be close to optimal. Issues relating to the latency and synchronization do not appear to have sufficient impact to degrade its performance.

TABLE 3. Performance Parameters for 3D-OESP Architecture

Parameter	256 Mhz Bus, 1 Ghz FSOI	256 Mhz Bus and FSOI
Bisection Bandwidth	256 Gb/s	64 Gb/s
Packet Transmission Delay over FSOI (Assuming 20% Framing Overhead)	105 ns	422 ns
Point to Point Packet Transmission Time	230 ns	864 ns
Calculation Time 1024x1024, 32 Bit Dataset	1.32 us	5.28 us
Time to Transpose 1024x1024, 32 Bit Dataset	108 us	430 us
Retraining Time	50 ns (Estimated)	200 ns
Data Load Time	2 ms	2 ms

Table 4 shows a comparison between a number of commerically availble devices for FFT computation. In this table the 3D-OESP system out performs each of these devices by several orders of magnitude while maintaining a realtively small footprint. In addition the system has the advantage of providing additional functionality beyond the FFT.

TABLE 4. Performance Comparison for FFT Processors

Device	Precision (bits)	1K x 1K Speed (ms)
AD 21060	32 (Float)	920
TI C80	32 (Float)	326
Plessy	16 (Fixed)	192
Sharp	24 (Fixed)	174

Device	**Precision (bits)**	**1K x 1K Speed (ms)**
Projected	24 (Fixed)	16
Actual	32 (Fixed)	1.4

To implement the current clock synchronization design, the system clock needed to be reduced to 256 Mhz due to the limitation of the current CMOS process. Further design optimizations could allow speeds in excess of 500 Mhz, however, it appears that this approach is limited for future applications if the full capabilities of the FSOI are to be explored.

The obvious bottleneck in the architecture is the EDI. The ability to transfer data to and from the system is the major factor in its speed. Since the EDI is merely a test interface, this should not impact the evaluation of the design as a whole. Commercial applications of the system would use another form of input such as a charge coupled device that would provide data through the optical channels.

A further increase in speed could be achieved through use of 128 bit packets over FSOI. By expanding the packet size, the control data overhead in the bit stuffing algorithm would be reduced. This would increase transfer efficiency by approximately 10%. However the increased chip area taken by the wider bus might make this an unappealing option.

REFERENCES

1. G. Betzos, P. Mitkas, “Performance Evaluation of 3D Optoelectronic Computer Architectures Based on the FFT and Sorting Benchmarks”, Proceedings of the 2nd International Conference on Parallel Processing Using Optical Interconnects”, San Antonio, TX, Oct 23-27, 1995

2. P. Cheo, Handbook of Solid State Lasers, Marcel Dekker(1989), p. 2.

3. M. Inguscio, R. Wallenstein, Solid State Lasers: New Developments and Applications, New York, Plenum Press, p. 83.

4. A. Mainwaring, S. Schleimer, “System Area Network Mapping”, 9th Annual ACM Symposium on Parallel Algorithms and Architectures, Newport, RI, June 22-25, 1997

5. B. Anglis, H. Hinton, “A Dynamically reconfigurable Token-based Optical Backplane”, IEEE/LOES Summer Topicals: Smart Pixels, July 1998.

6. J. Wu, C. Kuznia, C. Chen, B. Hoanca, A. Sawchuk, “Network with Free Space Optical Data Packet Using Carrier-Sense Multi-Access with Collision Detection (CSMA/CD) Protocol”, IEEE/LOES Summer Topicals: Smart Pixels, July 1998.

7. J. Rorie, P. Marchand, P. Chandramani, J. Ekman, F. Kiamilev, F. Zane, V. Ozguz, S. Esener, “A System Architecture for use with Free Space Optical Interconnects in a 3D Stacked Processor Environment”, IEEE/LOES Summer Topicals: Smart Pixels, July 1998.

8. S. Esener, P. Marchand, “3D Opto-electronic Stacked Processors: Design and Analysis” OC Computing '98, Bruges, Belgium, June 1998.

9. F. Halsall, “Data Communications, Computer Networks and Open Systems”, Fourth Edition, Addison Wesley, Harlow, England, 1996

10. J. Hammand, “Performance Analysis of Local Computer Networks”, Addison-Wesely, Reading, MA

11. B. Stuck, “Calculating the Maximum Throughput Rate in Local Area Networks”, IEEE Computer

12. S. Leong, B. Dewar, “The Effect of Traffic Locality on Network Architecture Performance”, Proceedings of the IASTED International Conference, Orlando, FL January 8-10, 1996

13. Johnson, Barry W., “Design and Analysis of Fault Tolerant Digital Systems”, Reading: Addison Wesley, 1989. 95-112.

14. N. McArdle, M. Naruse, T. Komuro, H. Sakaida, M. Ishikawa, Y. Kobayashi and H. Toyoda, “A Smart-Pixel Parallel Optoelectronic Computing System with Free-Space Dynamic Interconnections”, Proceedings of the International Conference on Massively Parallel Processing

15. T. Szymanski, H. Scott Hinton, “Design of a Terabit Free-Space Photonic Backplane for Parallel Computing”, Proceedings of the Second International Conference on Massively Parallel Processing Using Optical Interconnections, 1995

16. J. Wu, C. Kuznia, C. Chen, B. Hoanca, A. Sawchuk, “Network with Free Space Optical Packet Using Carrier-Sence Multiple-Access with Collision Detection (CSMA/CD) Protocol, IEEE/LOES Summer Topicals: Smart Pixels, July 1998.

17. A. Louri, S. Furlonge, C. Neocleous, “Experimental demonstration of the optical multi-mesh hypercube: scalable interconnection network for multiprocessors and multicomputers”, Applied Optics, December 10, 1996, pg 6906-6919

18. A. Louri, S. Furlonge, “Feasibility study of scalable optical interconnection network for massively parallel processing systems”, Applied Optics, March 10, 1996, pg 1296-1307

19. T. Pinkston, U. Efron, M. Cambell, “Applying Optical Interconnects to the 3-D Computer: A Performance Evaluation”, Journal of Parallel and Distributed Computing, October, 1994

20. O.Sjölund, D. A. Louderback, E. R. Hegblom, J. Ko, and L. A. Coldren, “Individually optimized bottom-emitting vertical-cavity lasers and bottom-illuminated resonant photodetectors sharing the same epitaxial structure”, Optics in Computing, Bruges, Belgium, June 1998.

21. Irvine Sensors Corporation, Costa Mesa, CA, 92626

22. M. Hibbs-Brenner, Y. Liu, R.Morgan, J. Lehman, “VCSEL/MSM Detector Smart Pixel Arrays”, IEEE/LOES Summer Topicals: Smart Pixels, July 1998.

23. J. Neff, C. Chen, T. McLaren, A. Mao, “VCSEL/CMOS Smart Pixel Arrays for Free-Space Optical Interconnects”, Proceeding of the International Conference on Massively Parallel Processing Using Optical Interconnects

APPENDIX A - VHDL SOURCE FILES

Included on the following pages are the source files for the simulations used in this document. The files are written in VHDL for use in both Epoch and Synopsys. Also included are the test benches for the associated components.

```
--
--                          Program:     3DOESP.vhd
--                          Author:      Jim Rorie
--                          Purpose:     Top Level Entity.
--
--
LIBRARY ieee;
USE ieee.std_logic_1164.all;


package LocalDefines is
constant        DataBusWidth : integer := 64;
constant  ProcessorAddressWidth : integer := 2;
constant  PlaneAddressWidth : integer := 4;

constant        AddressBusWidth : integer := ProcessorAddressWidth + PlaneAddressWidth + 1;

constant        EDIIndex : integer := 0;
constant        FSOIIndex : integer := 1;

constant        RunLength : integer := 5;
constant        FrameLength : integer := DataBusWidth + RunLength;
----------------------------------------------
-- Derived Constants
----------------------------------------------
constant        Planes : integer := 2 ** PlaneAddressWidth;
constant        Processors : integer := 2 ** ProcessorAddressWidth;
constant        VCSELChannels : integer := Planes;
CONSTANT DataWidth : Integer := DataBusWidth + 2;

TYPE VCSELArray IS Array (0 to Planes - 1) of STD_LOGIC_VECTOR(VCSELChannels -1 downto 0);

TYPE SerialRegisters IS ARRAY (Planes - 1 downto 0) of STD_LOGIC_VECTOR (DataWidth - 1 downto 0);

TYPE MemoryWord IS ARRAY ((Processors * Planes ) -1 downto 0) of STD_LOGIC_VECTOR (DataBusWidth - 1 downto 0);


end LocalDefines;
```

```
--
--                              Program:      ADD_TEST.vhd
--                              Author:       Jim Rorie
--                              Purpose:      Testbench for address decoder.
--
--
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE work.LocalDefines.all;
USE ieee.numeric_std.all;


ENTITY test_system IS
END test_system;


ARCHITECTURE testbench OF test_system IS


component AddressDecoder
PORT (
   AddressBus : IN STD_LOGIC_VECTOR(AddressBusWidth - 1 downto 0);
   Valid : OUT STD_LOGIC_VECTOR(Processors - 1 downto 0);
   CLK, Reset : IN STD_LOGIC;
   Forward, Control : OUT STD_LOGIC;
   PlaneAddress : IN STD_LOGIC_VECTOR(PlaneAddressWidth - 1 downto 0)
);
END COMPONENT;

-- Declaration of the component under test


SIGNAL  CLK : STD_LOGIC:=’0’;
SIGNAL  Reset, Forward, Control : STD_LOGIC:=’0’;

SIGNAL      Address : STD_LOGIC_VECTOR(AddressBusWidth - 1 downto 0);
SIGNAL      Valid : STD_LOGIC_VECTOR(Processors - 1 downto 0);
SIGNAL      PlaneAddress : STD_LOGIC_VECTOR(PlaneAddressWidth - 1 downto
0);

for dut : AddressDecoder use entity work.AddressDecoder(behavior);

BEGIN

CLK <= not (CLK) after 35 ns;
```

```
process begin
  wait for 10 ns;
  Reset <= ‘0’;
  Address <= (others => ‘0’);
                        PlaneAddress <= “1111”;

wait until clk’event and clk = ‘1’;

  Reset <= ‘1’;

wait until clk’event and clk = ‘1’;
 Address <= “0111100”;

wait until clk’event and clk = ‘1’;
  Address <= “0111111”;

wait until clk’event and clk = ‘1’;
                        Address <= “0110011”;

wait until clk’event and clk = ‘1’;
  Address <= “0111100”;

wait until clk’event and clk = ‘1’;
  Address <= “0111101”;
wait until clk’event and clk = ‘1’;
  Address <= “0111110”;
wait until clk’event and clk = ‘1’;
  Address <= “1111111”;
wait;

end process;



dut : AddressDecoder PORT MAP (Addressbus => Address, Valid => Valid, CLK => CLK, Reset => Reset, Forward => Forward, Control => Control, PlaneAddress => PlaneAddress);



END testbench;
```

```
--
--                Program:    ADD_DEC.vhd
--                Author:     Jim Rorie
--                Purpose:    address decoder.
--
--
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_unsigned.all;
USE work.LocalDefines.all;
-- USE ieee.numeric_std.all;

ENTITY AddressDecoder IS
PORT (
  AddressBus : IN STD_LOGIC_VECTOR(AddressBusWidth - 1 downto 0);
  Valid : OUT STD_LOGIC_VECTOR(Processors - 1 downto 0);
  CLK, Reset : IN STD_LOGIC;
  Forward, Control : OUT STD_LOGIC;
  PlaneAddress : IN STD_LOGIC_VECTOR(PlaneAddressWidth - 1 downto 0)
);
END AddressDecoder;

ARCHITECTURE Behavior OF AddressDecoder IS

BEGIN

        Decode : process(CLK, RESET, AddressBus, PlaneAddress)

        BEGIN

            if(Reset = ‘0’) then

                Valid <= (Others => ‘Z’);
                Forward <= ‘0’;

            else


                if(AddressBus(AddressBusWidth - 2 downto ((AddressBus-
Width - 2) - PlaneAddressWidth) + 1) = PlaneAddress) then
                    -- Same Plane address, figure local address
                    Forward <= ‘0’;

                    -- if control not bit set
                    if(AddressBus(AddressBusWidth - 1) = ‘0’) then
                        Valid <= (Others => ‘0’);
                        Valid(conv_integer(AddressBus(Processor-
AddressWidth - 1 downto 0))) <= ‘1’;
                        Control <= ‘0’;
                    else
                        Valid <= (others => ‘0’);
                        Control <= ‘1’;
                    end if;

                else
                    Forward <= ‘1’;
                    Valid <= (others => ‘0’);
                    Control <= ‘0’;
                end if;

            end if;


        END Process;


END behavior;
```

```
--
--                      Program:    BAU.vhd
--                      Author:     Jim Rorie
--                      Purpose:    Bus Arbitration Unit.
--
--
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_unsigned.all;
USE work.LocalDefines.all;
-- USE ieee.numeric_std.all;

ENTITY BAU IS
PORT (
                        DataBus, EDIBus : INOUT STD_LOGIC_VECTOR(DataBusWidth - 1 downto 0);
                        PlaneAddress : OUT STD_LOGIC_VECTOR(PlaneAddressWidth - 1 downto 0);
                        ACK : OUT STD_LOGIC_VECTOR(Processors + 1 downto 0);
                        CLK, Reset, Control, DataReady, EDIReady   : IN STD_LOGIC;
                        BAU0Ready, Online: OUT std_logic;
                        IRQ : IN STD_LOGIC_VECTOR(Processors + 1 downto 0)
);
END BAU;

ARCHITECTURE Behavior OF BAU IS
            Signal InitState : STD_LOGIC;

            Signal LocalAddress : Integer;


BEGIN

            HandleIRQ : process(CLK, RESET, IRQ)
                  Variable CurrentIRQ : Integer;
            BEGIN

                  if(Reset = ‘0’) then

                        ACK <= (Others => ‘0’);
                        CurrentIRQ := 0;

                  elsif(CLK’Event AND CLK = ‘1’) then
```

```
        ACK <= (Others => ‘0’);

        if(IRQ(EDIIndex) = ‘1’) then  -- EDI is maximum priority, but only for short periods
            -- If I have received my address and it is 0
            if(InitState = ‘0’ AND LocalAddress = 0) then
                ACK(EDIIndex) <= ‘1’;
            end if;

        elsif(IRQ(FSOIIndex) = ‘1’) then -- IRQ0 is the highest runtime priority
                ACK(FSOIIndex) <= ‘1’; -- Tell the source to put the data on the bus
        else
            -- CurrentIRQ exists to use the irqs in a round robin approach
            for I in 2 to Processors + 1 loop
                CurrentIRQ := CurrentIRQ + 1;
                if(CurrentIRQ > Processors + 1) then
                    CurrentIRQ := 2;
                end if;

                if(IRQ(CurrentIRQ) = ‘1’) then
                    ACK(CurrentIRQ) <= ‘1’; -- Tell the source to put the data on the bus
                    exit;  -- Process Nothing else
                end if;
            END LOOP;
        END IF;

    END IF;

END Process;

HandleData : process(CLK, RESET, DataReady)
BEGIN
if(Reset = ‘0’) then
        InitState <= ‘1’;
        BAU0Ready <= ‘0’;
        Online          <= ‘0’;
        DataBus <= (Others => ‘Z’);

        PlaneAddress <= (others => ‘0’);
        LocalAddress <= 0;

elsif(clk’event AND clk = ‘1’) then
```

```
            for I in 0 to Processors loop

                -- Get initialization data first
                if(InitState = ‘1’ AND DataReady = ‘1’ AND Control = ‘1’)
then

                    PlaneAddress <= DataBus(PlaneAddressWidth - 1
downto 0);
                    LocalAddress <= conv_integer(DataBus(PlaneAd-
dressWidth - 1 downto 0));
                    InitState <= ‘0’;

                    if(LocalAddress = 0) then
                        Online <= ‘1’;
                    end if;
                    exit;

                end if;

            end loop;
            if(EDIReady = ‘1’) then
                DataBus <= EDIBus;
            else
                DataBus <= (others => ‘Z’);
            end if;

        end if;



        END Process;

END behavior;
```

```
--
--                          Program:     Bau_TEST.vhd
--                          Author:      Jim Rorie
--                          Purpose:     Testbench for bau.
--
--
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE work.LocalDefines.all;
--USE ieee.numeric_std.all;


ENTITY test_system IS
END test_system;


ARCHITECTURE testbench OF test_system IS


component BAU
PORT (
                          DataBus, EDIBus : INOUT STD_LOGIC_VECTOR(DataBusWidth - 1 downto 0);
                          PlaneAddress : OUT STD_LOGIC_VECTOR(PlaneAddressWidth - 1 downto 0);
                          ACK : OUT STD_LOGIC_VECTOR(Processors + 1 downto 0);
                          CLK, Reset, Control, DataReady, EDIReady  : IN STD_LOGIC;
                          BAU0Ready, Online: OUT std_logic;
                          IRQ : IN STD_LOGIC_VECTOR(Processors + 1 downto 0)
);
END COMPONENT;

-- Declaration of the component under test


SIGNAL      CLK : STD_LOGIC:=’0’;
SIGNAL      Reset, Control, DataReady, BAU0Ready, DataPresent : STD_LOGIC:=’0’;
SIGNAL      DataBus : STD_LOGIC_vector(DatabusWidth - 1 downto 0);
SIGNAL      PlaneAddress : STD_LOGIC_VECTOR(PlaneAddressWidth - 1 downto 0);
SIGNAL      ACK, IRQ : STD_LOGIC_VECTOR(Processors downto 0);

for dut : BAU use entity work.bau(behavior);
```

```
BEGIN

CLK <= not (CLK) after 35 ns;

process begin
  wait for 10 ns;
  Reset <= ‘0’;
  Databus <= (others => ‘Z’);

   IRQ <= (others => ‘0’);

wait until clk’event and clk = ‘1’;

  Reset <= ‘1’;

wait until clk’event and clk = ‘1’;
                        IRQ(0) <= ‘1’;

wait until clk’event and clk = ‘1’;
                        IRQ(1) <= ‘1’;
                        IRQ(0) <= ‘0’;


wait until clk’event and clk = ‘1’;
                        IRQ(1) <= ‘0’;
wait until clk’event and clk = ‘1’;
wait until clk’event and clk = ‘1’;
wait;

end process;

Process (reset, ack, DataReady)
                        Variable OutputData : Integer;

begin

if(reset = ‘0’) then
                        OutputData := 0;
  DataReady <= ‘0’;
else
--elsif(clk’event and clk = ‘1’) then

                                    if(DataReady = ‘1’) then
                                                DataReady <= ‘0’;
                                                OutputData := 0;
                                    end if;
                        for i in 1 to Processors loop
                                if(ack(i) = ‘1’) then
                                        DataReady <= ‘1’;
                                        OutputData := 1;
                                end if;
                        end loop;

                        if(OutputData = 1) then
                                DataBus                                                 <=
“1000000000000000000000000000000000000000000000000000000000000000”;
                        else
                                DataBus <= (others => ‘Z’);
                        end if;

                end if;

                Control <= DataBus(DataBusWidth -1);

end process;



dut : BAU PORT MAP (Databus => Databus, PlaneAddress => PlaneAddress, ACK =>
ACK, CLK => CLK, Reset => Reset, Control => Control, DataReady => DataReady, IRQ
=> IRQ, BAU0Ready => BAU0Ready);



END testbench;
```

```
--
--                              Program:        EDI.vhd
--                              Author:         Jim Rorie
--                              Purpose:        External Data Interface Stub.
--
--
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE work.LocalDefines.all;

ENTITY EDI IS
PORT (
    CLK, Reset  : IN STD_LOGIC;
    DataBus : INOUT STD_LOGIC_VECTOR(DataBusWidth-1 downto 0);
    ACK, BAU0Ready, Online  : IN STD_LOGIC;
    IRQ, EDIReady : OUT STD_LOGIC
);
END EDI;

ARCHITECTURE Behavior OF EDI IS


BEGIN

Process(clk, reset)
begin
                if(Reset = ‘0’) then
                        DataBus <= (others => ‘Z’);
                        IRQ <= ‘0’;
                        EDIReady <= ‘0’;
                else
                end if;

end process;



END behavior;
```

```
--
--                              Program:        Frame_Recv.vhd
--                              Author:         Jim Rorie
--                              Purpose:        Serial Receiver.
--
--
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE work.LocalDefines.all;
-- USE ieee.numeric_std.all;


ENTITY FrameRecv IS

PORT (
                        SyncClk, Reset  : IN STD_LOGIC;
   PacketBus : OUT STD_LOGIC_VECTOR(DataBusWidth - 1 downto 0);
                        FrameReady : OUT STD_LOGIC;
                        SerialInput : IN STD_LOGIC;
);
END FrameRecv;

ARCHITECTURE Behavior OF FrameRecv IS

-- Input register must be large enough to hold data plus the trailing frame marker
SIGNAL FrameData : STD_LOGIC_VECTOR((FrameLength+ RunLength) - 1 downto
0);

BEGIN

-------------------------------------------------------------------------------
--
--                Find the frame markers and signal when a packet has been received.
--  Bit stuffing is used as the method of frame termination
--
-------------------------------------------------------------------------------


GetData : Process (SyncClk, Reset, SerialInput)

                        Variable CurrentRunLength : Integer range 0 to RunLength;
                        signal InFrame: Std_Logic;
                        Variable BitCount : Integer range 0 to FrameLength;

BEGIN
```

```
 if(Reset = ‘0’) then
            CurrentRunLength := 0;
            FrameData <= (Others => ‘0’);
            InFrame <= ‘0’;
            FrameReady <= ‘0’;
            BitCount := 0;



        elsif(SyncClk’event and SyncClk = ‘1’) then

            if(FrameReady = ‘1’) then
                FrameReady <= ‘0’;
            end if;


            -- If the runlength is near, check the next bit
            if(CurrentRunLength = RunLength) then

                if(SerialInput = ‘1’) then -- if a violation occurs, must be a frame marker
                    if(InFrame = ‘1’) then
                        if(BitCount < FrameCount + RunLength) else -- Too Short, must be out of sync
                            InFrame <= ‘1’;
                            BitCount := 0;
                        else
                            FrameReady <= ‘1’;
                            InFrame <= ‘0’;
                    else
                        InFrame <= ‘1’;
                        BitCount := 0;
                    end if;
                end if; --  Do nothing if a ‘0’ is received

            else

                if(SerialInput = ‘1’) then
                    CurrentRunLength := CurrentRunLength + 1;
                else
                    CurrentRunLength := 0;
                end if;

                FrameData(FrameLength - 2 downto 0) <= FrameData(FrameLength - 1 downto 1);
                FrameData(FrameLength - 1) <= SerialInput)
```

```
                    BitCount := BitCount + 1;

          end if;

End Process;

END Behavior;
```

```
--
--                              Program:        Frame_Xmit.vhd
--                              Author:         Jim Rorie
--                              Purpose:        Serial Transmitter.
--
--
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE work.LocalDefines.all;
-- USE ieee.numeric_std.all;


ENTITY FrameXmit IS

PORT (
                        SyncClk, Reset, Load  : IN STD_LOGIC;
   PacketBus : IN STD_LOGIC_VECTOR(DataBusWidth - 1 downto 0);
                        Idle : OUT STD_LOGIC;
                        SerialOutput : OUT STD_LOGIC
);
END FrameXmit;

ARCHITECTURE Behavior OF FrameXmit IS

-- Input register must be large enough to hold data plus the trailing frame marker
SIGNAL FrameData : STD_LOGIC_VECTOR((FrameLength+ RunLength) - 1 downto
0);
type StateType is (NoData, StartFrame, Data, EndFrame);
                Signal CurrentState, NextState : StateType;

BEGIN

---------------------------------------------------------------------------
--
--              Generate frame markers and signal when a packet has been transmitted.
--  Bit stuffing is used as the method of frame termination
--
---------------------------------------------------------------------------


SendData : Process (SyncClk, Reset)

                        Variable CurrentRunLength : Integer range 0 to RunLength;
                        Variable BitCount : Integer range 0 to FrameLength;

BEGIN
```

```
 if(Reset = ‘0’) then
                CurrentRunLength := 0;
                FrameData <= (Others => ‘0’);
                NextState <= NoData;
                BitCount := 0;


        else

        case CurrentState is
        when NoData =>

                if(Load = ‘1’) then
                        FrameData((FrameLength - RunLength) - 1 downto 0) <= PacketBus;
                        NextState <= StartFrame;
                        Idle <= ‘0’;

                        BitCount := 0;
                        CurrentRunLength := 0;
                else
                        NextState <= NoData;
                        Idle <= ‘1’;
                end if;

        when Data =>
                -- If the runlength is near, check the next bit
                if(BitCount = FrameLength) then -- Data Sent, Send Frame Now
                        BitCount := 0;
                        SerialOutput <= ‘0’;
                        NextState <= EndFrame;
                else
                        SerialOutput <= FrameData(FrameLength - 1);
                        FrameData(FrameLength-1 downto 1) <= FrameData(FrameLength - 2 downto 0);
                        BitCount := BitCount + 1;
                end if;
        when EndFrame =>
                if(CurrentRunLength = RunLength) then
                        SerialOutput <= ‘0’;
                        NextState <= NoData;
                else
                        BitCount := 0;
                end if;
        when StartFrame =>
```

```
            if(CurrentRunLength = RunLength) then
                SerialOutput <= ‘0’;
                NextState <= Data;
            else
                BitCount := 0;
            end if;

        end case;

        end if;




End Process;


Sequential: Process(SyncClk, Reset, NextState)
BEGIN

        if(Reset = ‘0’) then
            CurrentState <= NoData;
        elsif(SyncClk’Event AND SyncClk = ‘1’) then
            CurrentState <= NextState;
        end if;

End Process;

END Behavior;
```

```
--
--                          Program:     FSOI.vhd
--                          Author:      Jim Rorie
--                          Purpose:     FSOI Comm. Network.
--
--

LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE work.LocalDefines.all;
-- USE ieee.numeric_std.all;


ENTITY FSOI IS

PORT (
                    CLK, Reset  : IN STD_LOGIC;
  DataBus : INOUT STD_LOGIC_VECTOR(DataBusWidth - 1 downto 0);
                    DataPresent, ACK, AddressEnable : IN STD_LOGIC;
                    DataReady, IRQ : OUT STD_LOGIC;
                    Output : OUT STD_LOGIC_VECTOR(Planes - 1 downto 0);
                    Input : IN STD_LOGIC_VECTOR(Planes - 1 downto 0)
);
END FSOI;

ARCHITECTURE Behavior OF FSOI IS

SIGNAL PISOLoad, PISOBusy, SIPOBusy : STD_LOGIC_VECTOR(Planes - 1 downto
0);

SIGNAL PISO, SIPO : SerialRegisters;
signal DataReadyOut : Std_logic;

BEGIN

-------------------------------------------------------------------------------
--
--             Accepts data from the Parallel bus and shifts it out the serial channels
--
-------------------------------------------------------------------------------

             HandleLoad : process(CLK, RESET, DataPresent)

             BEGIN

                    if(Reset = ‘0’) then
```

```
                PISOLoad <= (Others => ‘0’);
                PISOBusy <= (Others => ‘0’);
                for i in 0 to Planes - 1 loop
                    PISO(i) <= (Others => ‘0’);
                end loop;
                Output <= (others => ‘0’);


            elsif(CLK’Event AND CLK = ‘1’) then

                -- if data is on the bus and I am selected

                if(DataPresent = ‘1’ AND AddressEnable = ‘1’) then

                    -- This loop finds an open channel to put databus into
                    for I in 0 to Planes - 1 LOOP

                        if(PISOBusy(I) = ‘0’) then
                            PISOBusy(I) <= ‘1’;
                            PISO(I)(DataWidth - 2 downto 1) <=
DataBus;

                            PISO(I)(DataWidth -1)  <= ‘0’;
                            PISO(I)(0) <= ‘1’;
                            exit;
                        end if;
                    end loop;
                end if;

                -- This loop shifts all busy PISO’s

                for I in 0 to Planes - 1 LOOP
                    if(PISOBusy(I) = ‘1’) then

                        for J in 0 to DataWidth-2 LOOP
                            PISO(I) (J) <= PISO(I) (J+1);
                        end loop;
                        Output(I) <= PISO(I)(0);
                    end if;
                end loop;

            end if;

        END Process;

-------------------------------------------------------------------------
```

```
--
--
--
-------------------------------------------------------------------------------


        HandleShift : process(CLK, RESET, ACK)

            Variable ChannelLocked, Index : Integer range 0 to Planes -1;
            Variable OutputChannel : Integer range 0 to Planes -1;

        BEGIN

            if(Reset = ‘0’) then

                DataBus <= (Others => ‘Z’);
                SIPOBusy <= (Others => ‘0’);
                OutputChannel := 0;
                ChannelLocked := 0;

                IRQ <= ‘0’;
                DataReadyOut <= ‘0’;
                Index := 0;

            elsif(CLK’Event AND CLK = ‘1’) then

                if(DataReadyOut = ‘1’) then
                    DataReadyOut <= ‘0’;

                end if;


                for I in 0 to Planes - 1 loop

                    -- if a valid frame is received, stop shifting that chan-
nel and IRQ
                    if(SIPO(I)(0) = ‘1’ AND SIPO(I)(DataWidth -1) =
‘0’) then

                        SIPOBusy(I) <= ‘1’;

                    -- else shift
                    elsif(SIPOBusy(I) = ‘0’) then
                        FOR J IN 0 TO  DataWidth - 2 LOOP
                            SIPO(I) (J) <= SIPO(I) (J+1);
                            SIPO(I) (DataWidth -1) <= Input(I);
                        END LOOP;

                    end if;

                end loop;

                if(ACK = ‘1’)  then
                    DataBus  <=  SIPO(OutputChannel)(DataWidth  -  2
downto 1);
                    DataReadyOut <= ‘1’;
                    SIPOBusy(OutputChannel) <= ‘0’;
                    SIPO(OutputChannel) <= (Others => ‘0’);
                    ChannelLocked := 0;
                    IRQ <= ‘0’;
                else
                    DataBus <= (others => ‘Z’);
                end if;

                if(ChannelLocked = 0) then
                    Index := OutputChannel;
                    for I in 0 to Planes - 1 loop
                        if(SIPOBusy(Index)  =  ‘1’  AND  Index  /=
OutputChannel) then
                            OutputChannel := Index;
                            ChannelLocked := 1;
                            IRQ <= ‘1’;
                            exit; -- stop once I found it
                        end if;
                        Index := (Index + 1) mod (Planes - 1);
                    end loop;
                    if(ChannelLocked = 0) then
                        OutputChannel := (OutputChannel + 1) mod
(Planes - 1);
                    end if;
                end if;

            end if;

        END Process;

        DataReady <= DataReadyOut;


END behavior;
```

```
--
--                              Program:        FSOI_Test.vhd
--                              Author:         Jim Rorie
--                              Purpose:        Testbench for FOSI.
--
--
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_arith.all;
USE work.LocalDefines.all;
--USE ieee.numeric_std.all;



ENTITY test_system IS
END test_system;

ARCHITECTURE testbench OF test_system IS
                COMPONENT FSOI
                PORT (
                        CLK, Reset  : IN STD_LOGIC;
  DataBus : INOUT STD_LOGIC_VECTOR(DataBusWidth - 1 downto 0);
                        DataPresent, ACK : IN STD_LOGIC;
                        DataReady, IRQ : OUT STD_LOGIC;
                        Output : OUT STD_LOGIC_VECTOR(Planes -1 downto 0);
                        Input : IN STD_LOGIC_VECTOR(Planes - 1 downto 0)
                );

                END COMPONENT;

                -- Declaration of the component under test


                SIGNAL          CLK : STD_LOGIC:=’0’;
                SIGNAL          Reset : STD_LOGIC:=’0’;
                SIGNAL          IDataBus, ODataBus : STD_LOGIC_vector(DatabusWidth
- 1 downto 0);
                SIGNAL          DUH, ACK, IRQ, IDataPresent, ODataPresent, IDataReady,
ODataReady : STD_LOGIC;
                SIGNAL          Input, Output : STD_LOGIC_vector(Planes - 1 downto 0);

                for dut1 : FSOI use entity work.FSOI(behavior);
                for dut2 : FSOI use entity work.FSOI(behavior);

                function OutputData(c:integer) return Std_Logic_vector is
                        variable Index : Integer;
```

```
            variable Data : Std_Logic_vector(DataBusWidth -1 downto 0);
        begin
            Index := 0;
            while (Index < DataBusWidth) loop
                Data(Index) := ‘0’;
                Index := Index + 2;

            end loop;

            Index := 1;
            while (Index < DataBusWidth) loop
                Data(Index) := ‘1’;
                Index := Index + 2;

            end loop;

            return Data;
        end OutputData;


BEGIN

        CLK <= not (CLK) after 35 ns;

        Process begin
            wait for 10 ns;
        Reset <= ‘0’;
        IDatabus <= (others => ‘Z’);
            ODatabus <= (others => ‘Z’);

        IDataPresent <= ‘0’;
        ODataPresent <= ‘0’;

            wait until clk’event and clk = ‘1’;
        Reset <= ‘1’;

            wait until clk’event and clk = ‘1’;
            IDataBus <= CONV_STD_LOGIC_VECTOR(122, DataBus-
Width);
            IDataPresent <= ‘1’;

            wait until clk’event and clk = ‘1’;
            IDataPresent <= ‘0’;
            IDataBus <= (Others => ‘Z’);

            wait until clk’event and clk = ‘1’;
            wait until clk’event and clk = ‘1’;
            wait until clk’event and clk = ‘1’;

            IDataBus <= CONV_STD_LOGIC_VECTOR(1817, DataBusWidth);
            IDataPresent <= ‘1’;

            wait until clk’event and clk = ‘1’;
            IDataBus <= CONV_STD_LOGIC_VECTOR(543556, DataBusWidth);
            IDataPresent <= ‘1’;

            wait until clk’event and clk = ‘1’;
            IDataBus <= CONV_STD_LOGIC_VECTOR(23452, DataBusWidth);
            IDataPresent <= ‘1’;

            wait until clk’event and clk = ‘1’;

            IDataBus <= CONV_STD_LOGIC_VECTOR(6745656, DataBusWidth);
            IDataPresent <= ‘1’;
            wait until clk’event and clk = ‘1’;
            IDataBus <= CONV_STD_LOGIC_VECTOR(836465, DataBusWidth);
            IDataPresent <= ‘1’;

            wait until clk’event and clk = ‘1’;
            IDataPresent <= ‘0’;
            IDataBus <= (Others => ‘Z’);

            wait until clk’event and clk = ‘1’;
            wait until clk’event and clk = ‘1’;

            wait until clk’event and clk = ‘1’;

            for I in 0 to DataBusWidth + 2 loop
                wait until clk’event and clk = ‘1’;
            end loop;

            wait;

end process;
```

```
dut1 : FSOI PORT MAP (Databus => IDatabus, DataPresent => IDataPresent,  CLK =>
CLK, ACK => DUH, IRQ => DUH, Reset => Reset, DataReady => IDataReady, Input =>
Input, Output => Output);

dut2 : FSOI PORT MAP (Databus => ODatabus, DataPresent => ODataPresent,  CLK =>
CLK, ACK => ACK, IRQ => IRQ, Reset => Reset, DataReady => ODataReady, Output
=> Input, Input => Output);


ACK <= IRQ;

END testbench;
```

```
--
--                          Program:     Node.vhd
--                          Author:      Jim Rorie
--                          Purpose:     Node Stub.
--
--
LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY Node IS
PORT (
      MainDataBus, LowDataBus, HighDataBus   : INOUT STD_LOGIC_VECTOR(63
downto 0);
   DestinationAddress : INOUT STD_LOGIC_VECTOR(9 downto 0);
   EDIReady, CLK, Reset  : IN STD_LOGIC;
   BAU0Ready : OUT STD_LOGIC
);
END Node;

ARCHITECTURE behavior OF Node IS

COMPONENT ProcessorStack
PORT (
   DataBus : INOUT STD_LOGIC_VECTOR(63 downto 0);
   SourceAddress : INOUT STD_LOGIC_VECTOR(9 downto 0);
   DestinationAddress : INOUT STD_LOGIC_VECTOR(9 downto 0);
   EDIReady, CLK, Reset  : IN STD_LOGIC;
   BAU0Ready : OUT STD_LOGIC
);
END COMPONENT;

COMPONENT FSOI
GENERIC (
               DataBusWidth : integer := 64;
               AddressBusWidth : integer := 10;
               Planes : integer := 16
);

COMPONENT XBarPlane IS
PORT (
      LocalDataBus, LowDataBus, HighDataBus  : INOUT STD_LOGIC_VECTOR(63
downto 0);

   CLK, Reset, BusReady  : IN STD_LOGIC;
               XBarReady: OUT STD_LOGIC
);
END COMPONENT;



BEGIN

ProcStack :  ProcessorStack  PORT MAP (
    DataBus => DataBus,
                   SourceAddress => SourceAddress,
                   DestinationAddress => DestinationAddress,
    EDIReady => EDIReady,
                   CLK => CLK,
    BAU0Ready => BAU0Ready,
    Reset => Reset
);

                   FSOI0 : FSOI PORT MAP (DataBus, SourceAddress, DestinationAddress,
Halt(0), BusReady(0), CLK, Reset, CPUReady(0), IRQ(0), FSOIOutput, FSOIInput);



END Behavior;
```

```
--
--                         Program:     Plane_Test.vhd
--                         Author:      Jim Rorie
--                         Purpose:     Testbench for Plane.
--
--
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE work.LocalDefines.all;
--USE ieee.numeric_std.all;


ENTITY test_system IS
END test_system;


ARCHITECTURE testbench OF test_system IS


component ProcessorPlane
PORT (
    CLK, Reset  : IN STD_LOGIC;
    DataBus, EDIBus : INOUT STD_LOGIC_VECTOR(DataBusWidth - 1 downto 0);
    EDIReady, EIRQ  : IN STD_LOGIC;
    BAU0Ready, Online, EACK : OUT STD_LOGIC;
    FSOIOutput : OUT STD_LOGIC_VECTOR(Planes-1 downto 0);
    FSOIInput : IN STD_LOGIC_VECTOR(Planes-1 downto 0)
);
END COMPONENT;

-- Declaration of the component under test


SIGNAL        CLK : STD_LOGIC:=’0’;
SIGNAL        Reset, EIRQ, EDIReady, EACK, Control, DataReady, BAU0Ready, DataPresent : STD_LOGIC:=’0’;
SIGNAL        DataBus, EDIBus : STD_LOGIC_vector(DatabusWidth - 1 downto 0);
SIGNAL        PlaneAddress : STD_LOGIC_VECTOR(PlaneAddressWidth - 1 downto 0);
SIGNAL        ACK, IRQ : STD_LOGIC_VECTOR(Processors downto 0);
signal  FSOIInput, FSOIOutput : STD_LOGIC_VECTOR(Planes-1 downto 0);


for dut : ProcessorPlane use entity work.ProcessorPlane(behavior);

BEGIN
```

```
CLK <= not (CLK) after 35 ns;

process begin
  wait for 10 ns;
  Reset <= ‘0’;
  Databus <= (others => ‘Z’);

   IRQ <= (others => ‘0’);

wait until clk’event and clk = ‘1’;

  Reset <= ‘1’;

wait until clk’event and clk = ‘1’;
                        IRQ(0) <= ‘1’;

wait until clk’event and clk = ‘1’;
                        IRQ(1) <= ‘1’;
                        IRQ(0) <= ‘0’;


wait until clk’event and clk = ‘1’;
wait until clk’event and clk = ‘1’;

                        IRQ(0) <= ‘0’;
wait until clk’event and clk = ‘1’;
wait;

end process;

Process (reset, ack, DataReady)
                        Variable OutputData : Integer;

begin

if(reset = ‘0’) then
                        OutputData := 0;
  DataReady <= ‘0’;
else
--elsif(clk’event and clk = ‘1’) then

                                    if(DataReady = ‘1’) then
                                                DataReady <= ‘0’;
                                                OutputData := 0;
                                    end if;
```

```
            for i in 1 to Processors loop
                if(ack(i) = ‘1’) then
                    DataReady <= ‘1’;
                    OutputData := 1;
                end if;
            end loop;

            if(OutputData = 1) then
                DataBus                                                  <=
“1000000000000000000000000000000000000000000000000000000000000000”;
            else
                DataBus <= (others => ‘Z’);
            end if;

        end if;

        Control <= DataBus(DataBusWidth -1);

end process;



dut : ProcessorPlane PORT MAP (CLK => CLK, Reset => Reset, Databus => Databus,
EDIBus => EDIBus, EDIReady => EDIReady, EIRQ => EIRQ, BAU0Ready =>
BAU0Ready, FSOIOutput => FSOIOutput, FSOIInput => FSOIInput);



END testbench;
```

```
--
--                              Program:       Processor.vhd
--                              Author:        Jim Rorie
--                              Purpose:       Processor Stub.
--
--

LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_unsigned.all;
USE ieee.std_logic_arith.all;
USE work.LocalDefines.all;
--USE ieee.numeric_std.all;




ENTITY Processor IS
PORT (
   CLK, Reset  : IN STD_LOGIC;
   DataBus : INOUT STD_LOGIC_VECTOR(DataBusWidth - 1 downto 0);
   ACK, DataPresent, AddressEnable : IN STD_LOGIC;
   DataReady, IRQ : OUT STD_LOGIC
);

END Processor;


ARCHITECTURE behavior OF Processor IS
Signal Counter : integer range 0 to 64;

BEGIN

                GenData: Process(Clk, Reset, ACK)
                Begin

                if(Reset = ‘0’) then
                        Counter <= 0;
                elsif(clk’event and clk = ‘1’) then

                        if(Ack = ‘0’) then
                                IRQ <= ‘1’;
                        else
                                Counter <= counter + 1;
                        end if;
```

```
        end if;

        end Process;


        Xmit: Process(Clk, Reset, ACK)
        Begin

        if(Reset = ‘0’) then
                Databus <= (Others => ‘Z’);
                DataReady <= ‘0’;
        elsif(clk’event and clk = ‘1’) then

            if(Ack = ‘1’) then

                DataBus <= CONV_STD_LOGIC_VECTOR(Counter, DataBusWidth);
                DataReady <= ‘1’;

            else

                Databus <= (Others => ‘Z’);
                DataReady <= ‘0’;

            end if;

        end if;

        end Process;

END behavior;
```

```
--
--                    Program:    Processor_Plane.vhd
--                    Author:     Jim Rorie
--                    Purpose:    One VLSI PLane.
--
--

LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE work.LocalDefines.all;

ENTITY ProcessorPlane IS
PORT (
   CLK, Reset  : IN STD_LOGIC;
   DataBus, EDIBus : INOUT STD_LOGIC_VECTOR(DataBusWidth - 1 downto 0);
   EDIReady, EIRQ  : IN STD_LOGIC;
   BAU0Ready, Online, EACK : OUT STD_LOGIC;
   FSOIOutput : OUT STD_LOGIC_VECTOR(Planes-1 downto 0);
   FSOIInput : IN STD_LOGIC_VECTOR(Planes-1 downto 0)
);
END ProcessorPlane;

ARCHITECTURE behavior OF ProcessorPlane IS


COMPONENT BAU
PORT (
                    DataBus, EDIBus : INOUT STD_LOGIC_VECTOR(DataBusWidth - 1 downto 0);
                    PlaneAddress : OUT STD_LOGIC_VECTOR(PlaneAddressWidth - 1 downto 0);
                    ACK : OUT STD_LOGIC_VECTOR(Processors + 1 downto 0);
                    CLK, Reset, Control, DataReady, EDIReady  : IN STD_LOGIC;
                    BAU0Ready, Online: OUT std_logic;
                    IRQ : IN STD_LOGIC_VECTOR(Processors + 1 downto 0)
);
END COMPONENT;


COMPONENT Processor
PORT (
   CLK, Reset  : IN STD_LOGIC;
   DataBus : INOUT STD_LOGIC_VECTOR(DataBusWidth - 1 downto 0);
   ACK, DataPresent, AddressEnable : IN STD_LOGIC;
```

```
    DataReady, IRQ : OUT STD_LOGIC
);
END COMPONENT;


COMPONENT FSOI
PORT (
                              CLK, Reset  : IN STD_LOGIC;
   DataBus : INOUT STD_LOGIC_VECTOR(DataBusWidth - 1 downto 0);
                              DataPresent, ACK, AddressEnable : IN STD_LOGIC;
                              DataReady, IRQ : OUT STD_LOGIC;
                              Output : OUT STD_LOGIC_VECTOR(Planes - 1 downto 0);
                              Input : IN STD_LOGIC_VECTOR(Planes - 1 downto 0)
);
END COMPONENT;

component AddressDecoder
PORT (
    AddressBus : IN STD_LOGIC_VECTOR(AddressBusWidth - 1 downto 0);
    Valid : OUT STD_LOGIC_VECTOR(Processors - 1 downto 0);
    CLK, Reset : IN STD_LOGIC;
    Forward, Control : OUT STD_LOGIC;
    PlaneAddress : IN STD_LOGIC_VECTOR(PlaneAddressWidth - 1 downto 0)
);
END component;

SIGNAL DataReady, ACK, IRQ : STD_LOGIC_VECTOR(Processors + 1 downto 0);
signal Control, DataPresent, Forward : std_logic;
signal Valid : std_logic_vector(Processors - 1 downto 0);
signal PlaneAddress : std_logic_vector(PlaneAddressWidth-1 downto 0);


BEGIN



Decoder : AddressDecoder port map (AddressBus => DataBus((DatabusWidth - 1) downto
(((DatabusWidth - 1) - AddressBusWidth) + 1)), Valid => Valid, clk => clk, reset => reset,
Forward => Forward, Control => Control, PlaneAddress => PlaneAddress);

                    -- Expand this to 4 processors
Procs: FOR X in 1 to Processors GENERATE

                              CPU : Processor PORT MAP (DataBus => DataBus, Ack=>
Ack(X),
                                        DataPresent => DataPresent, AddressEnable => Valid(X),
CLK => CLK,
                                        Reset => Reset, DataReady => DataReady(X), IRQ =>
IRQ(X));

end generate;

                    BAU0 : BAU PORT MAP (Databus => DataBus, EDIBus => EDIBus, PlaneAddress => PlaneAddress, ACK => Ack, CLK => CLK, Reset => Reset, Control =>
Control,        DataReady => DataPresent, EDIReady => EDIReady, BAU0Ready =>
BAU0Ready, Online => Online, IRQ => IRQ);

                    VCSELS: FSOI port map (clk => clk, reset => reset, DataBus => Databus,
DataPresent => DataPresent, ACK => Ack(FSOIIndex), AddressEnable => Forward, DataReady => DataReady(FSOIIndex), Input => FSOIInput, Output => FSOIOutput);

DataReady(EDIIndex) <= EDIReady;

process(DataReady)
begin
for I in 0 to Processors - 1 loop
                    DataPresent <= DataPresent OR DataReady(i);
end loop;


end process;


END behavior;
```

```
--
--                          Program:      Processor_Stack.vhd
--                          Author:       Jim Rorie
--                          Purpose:      Stakc of Planes.
--
--

LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE work.LocalDefines.all;

ENTITY ProcessorStack IS
PORT (
    DataBus : INOUT STD_LOGIC_VECTOR(DataBusWidth - 1 downto 0);
    EDIReady, CLK, Reset  : IN STD_LOGIC;
    BAU0Ready : OUT STD_LOGIC;
    FSOIOutput : OUT VCSELArray;
    FSOIInput : IN VCSELArray

);
END ProcessorStack;

ARCHITECTURE behavior OF ProcessorStack IS

COMPONENT ProcessorPlane
PORT (
    CLK, Reset  : IN STD_LOGIC;
    DataBus : INOUT STD_LOGIC_VECTOR(DataBusWidth - 1 downto 0);
    EDIReady  : IN STD_LOGIC;
    BAU0Ready : OUT STD_LOGIC;
    FSOIOutput : OUT STD_LOGIC_VECTOR(15 downto 0);
    FSOIInput : IN STD_LOGIC_VECTOR(15 downto 0)
);
END COMPONENT;




BEGIN

-- Because ORCAD doesn't have a $&$%^%*&#^$ Generate Statement
Planes: FOR i in 0 to 15 GENERATE

Slice : ProcessorPlane PORT MAP (CLK => CLK, Reset => Reset, DataBus => DataBus,
EDIReady  =>  EDIReady,  BAU0Ready  =>  BAU0Ready,  FSOIInput  =>  FSOIInput(i),
FSOIOutput => FSOIOutput(i));

END GENERATE;

END behavior;
```

```
--
--                              Program:        Trainer.vhd
--                              Author:         Jim Rorie
--                              Purpose:        Synchronization Trainer.
--
--
LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY Trainer IS
PORT (

   DataIn, Train, CLK, Reset  : IN STD_LOGIC;
   SyncClk: OUT STD_LOGIC
);
END Trainer;

ARCHITECTURE behavior OF Trainer IS

COMPONENT TrainerFSM
PORT (

   Train, CLK, Reset  : IN STD_LOGIC;
                        CurrentPass : OUT Integer range 0 to 3;
   Store, TrainClk, Compare: OUT STD_LOGIC
);
END COMPONENT;

               Signal DataReg : Std_Logic_Vector(7 downto 0);
               Signal CurrentClk, Equal, Store, TrainClk, Compare : Std_logic;
               Signal DelayCLK : Std_Logic_Vector(3 downto 0);
               Signal GoodCLK : Std_Logic_Vector(3 downto 0);
               Signal CurrentPass : Integer range 0 to 3;


BEGIN
FSM : TrainerFSM port map (CLK => CLK, Train => Train, Reset => Reset, Store =>
Store, TrainClk => TrainClk, CurrentPass => CurrentPass, Compare => Compare);

-------------------------------------------------------------------------------
--
--             This shifts the data in from the serial input
--
-------------------------------------------------------------------------------
```

```
Shifter : Process(CurrentClk, Reset, DataIn)
Begin
                if(Reset = ‘0’) then
                        DataReg <= (Others => ‘0’);

                elsif(CurrentClk’event AND CurrentClk = ‘1’) then
                        DataReg(7 downto 1) <= DataReg(6 downto 0);
                        DataReg(0) <= DataIn;
                end if;

end Process;

----------------------------------------------------------------------------
--
--              Creates a number of delayed clocks from a single clock source
--
----------------------------------------------------------------------------


ClockGen : Process(Clk, Reset)
Begin

                DelayClk(0) <= Clk;
                DelayClk(1) <= clk After 100 ps;
                DelayClk(2) <= clk After 200 ps;
                DelayClk(3) <= clk After 300 ps;

end Process;

----------------------------------------------------------------------------
--
--  Check the data coming in to determine if the clock is good
--  enough to use
--
--              This process operate asynchronously, thus no clock inference.
--
----------------------------------------------------------------------------




ClockSelect : Process(Reset, DelayClk, CurrentClk, TrainClk)
Begin

                if(Reset = ‘0’) then
```

```
            GoodClk <= “0000”;
        else
            if(Compare = ‘1’) then
                if(Equal = ‘1’) then
                    GoodClk(CurrentPass) <= ‘1’;
                else
                    GoodClk(CurrentPass) <= ‘0’;
                end if;
            end if;

            if(TrainClk = ‘1’) then
                CurrentClk <= DelayClk(CurrentPass);
            else
                for i in 3 downto 0 loop
                    if(GoodClk(i) = ‘1’) then
                        CurrentClk <= GoodClk(i); -- This should
make the lowest good one active
                    end if;
                end loop;
            end if;

        SyncClk <= CurrentClk;
        end if;


End Process;

-----------------------------------------------------------------------------
--
--      Compare the data to a know value and generated a Equal signal
--  It is important that the data is not symetric with time so that
--  false equals will not be generated
--
-----------------------------------------------------------------------------


Comparator : Process(DataReg)
Begin
        if(DataReg = “11111010”) then
            Equal <= ‘1’;
        else
            Equal <= ‘0’;
        end if;

end Process;
```

```
END Behavior;
```

```
--
--                         Program:     TrainerFSM.vhd
--                         Author:      Jim Rorie
--                         Purpose:     FSM for Trainer.
--
--

LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY TrainerFSM IS
PORT (

   Train, CLK, Reset  : IN STD_LOGIC;
                      CurrentPass : OUT Integer range 0 to 3;
   Store, TrainClk, Compare: OUT STD_LOGIC

);
END TrainerFSM;

ARCHITECTURE behavior OF TrainerFSM IS
            type StateType is (Idle, GetFrame);

            Signal CurrentState, NextState : StateType;
            Signal BitCount, BitReset, IncrementPass, PassCount, PassReset :
Std_logic;

BEGIN
Compare <= BitCount;

------------------------------------------------------------------------------
--
--          Combinational next state logic
--
------------------------------------------------------------------------------



Combinational: Process(CurrentState, PassCount, BitCount, Reset, Train)
BEGIN
--  Reset: Start from known state

if(reset = ‘0’) then
            IncrementPass <= ‘0’;
            BitReset <= ‘0’;
            PassReset <= ‘0’;
```

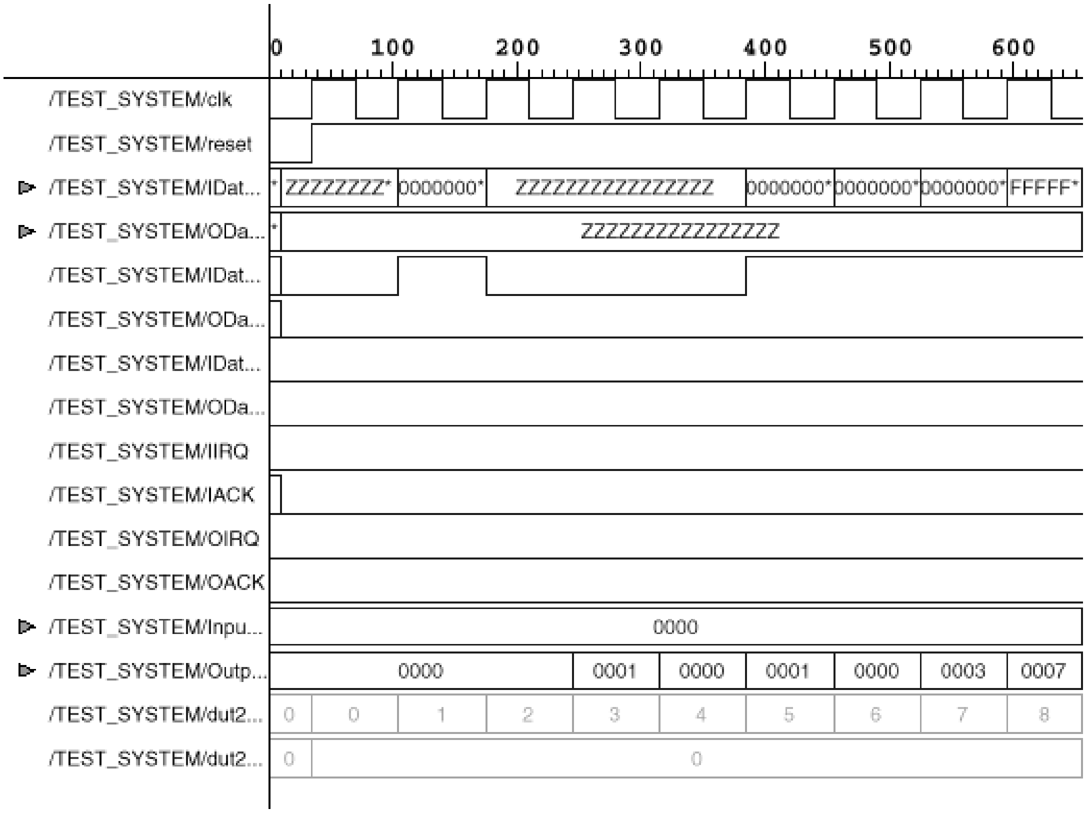
```
                    TrainClk <= ‘0’;
else

case CurrentState is
-- Train is asserted, begin counters
                    when IDLE =>
                              TrainClk <= ‘0’;
                              if(Train = ‘1’) then
                                        NextState <= GetFrame;
                                        BitReset <= ‘1’;
                              end if;

-- If Terminal Count, goto Idle, else increment counters

                    when GetFrame =>
                              TrainClk <= ‘1’;
                              BitReset <= ‘0’;
                              PassReset <= ‘0’;
                              IncrementPass <= ‘0’;
                              if(PassCount = ‘1’ AND BitCount = ‘1’) then
                                        PassReset <= ‘1’;
                                        BitReset <= ‘1’;
                                        NextState <= Idle;
                              elsif(BitCount = ‘1’) then
                                        BitReset <= ‘1’;
                                        IncrementPass <= ‘1’;
                              end if;

end case;
end if;

End Process;

------------------------------------------------------------------------------
--
--  Count to eight bits and reset when ordered
--
------------------------------------------------------------------------------


BitCounter : Process(Clk, Reset, BitReset)
Variable CurrentBit : Integer range 0 to 7;

Begin

if(Reset = ‘0’) then
```

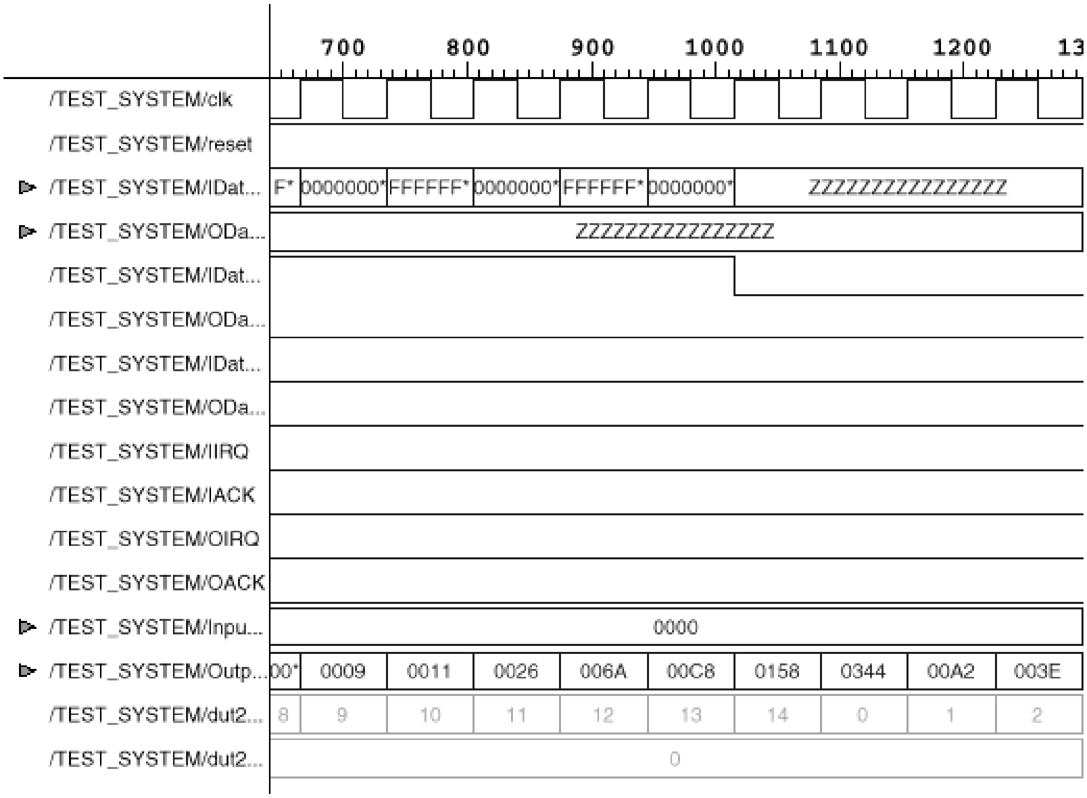
```
        CurrentBit := 0;
        BitCount <= ‘0’;
elsif(Clk’event and Clk=’1’) then
        if(BitReset = ‘1’) then
            CurrentBit := 0;
            BitCount <= ‘0’;
        else
            CurrentBit := CurrentBit + 1;
            if(CurrentBit = 7) then
                BitCount <= ‘1’;
            else
                Bitcount <= ‘0’;
            end if;
        end if;

end if;

end Process;

------------------------------------------------------------------------------
--
--  Count for 4 passes and reset when ordered
--
------------------------------------------------------------------------------



PassCounter : Process(Clk, Reset, IncrementPass, PassReset)
Variable LocalPass : integer range 0 to 3;

Begin

if(Reset = ‘0’) then
        LocalPass := 0;
        PassCount <= ‘0’;

elsif(Clk’event and Clk=’1’) then
        if(PassReset = ‘1’) then
            LocalPass := 0;
            PassCount <= ‘0’;
        elsif(IncrementPass = ‘1’) then
                LocalPass := LocalPass + 1;
                if(LocalPass = 3) then
                    PassCount <= ‘1’;
                else
                    Passcount <= ‘0’;
```

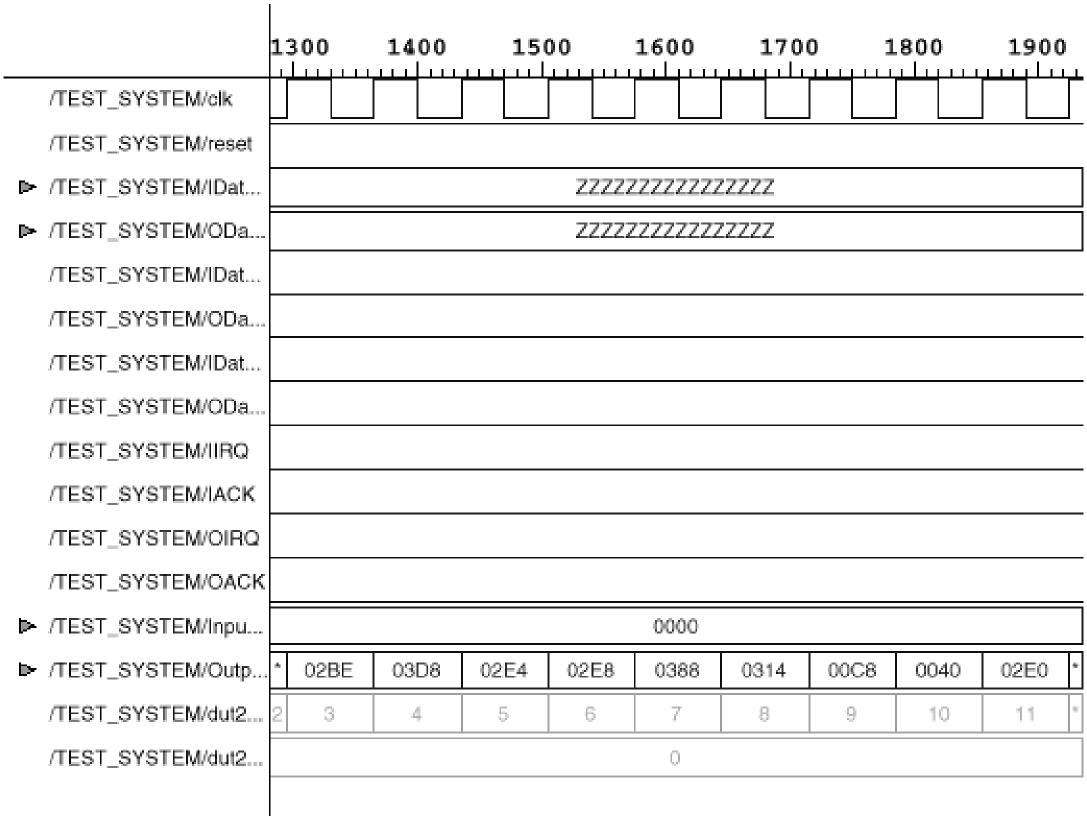
```
                                        end if;

                    end if;
CurrentPass <= LocalPass;
end if;


end Process;

-----------------------------------------------------------------------------
--
--  Sequential Logic
--
-----------------------------------------------------------------------------



Sequential: Process(Clk, Reset, NextState)
BEGIN

                    if(Reset = ‘0’) then
                              CurrentState <= IDLE;
                    elsif(CLK’Event AND CLK = ‘1’) then
                              CurrentState <= NextState;
                    end if;

End Process;

END Behavior;
```

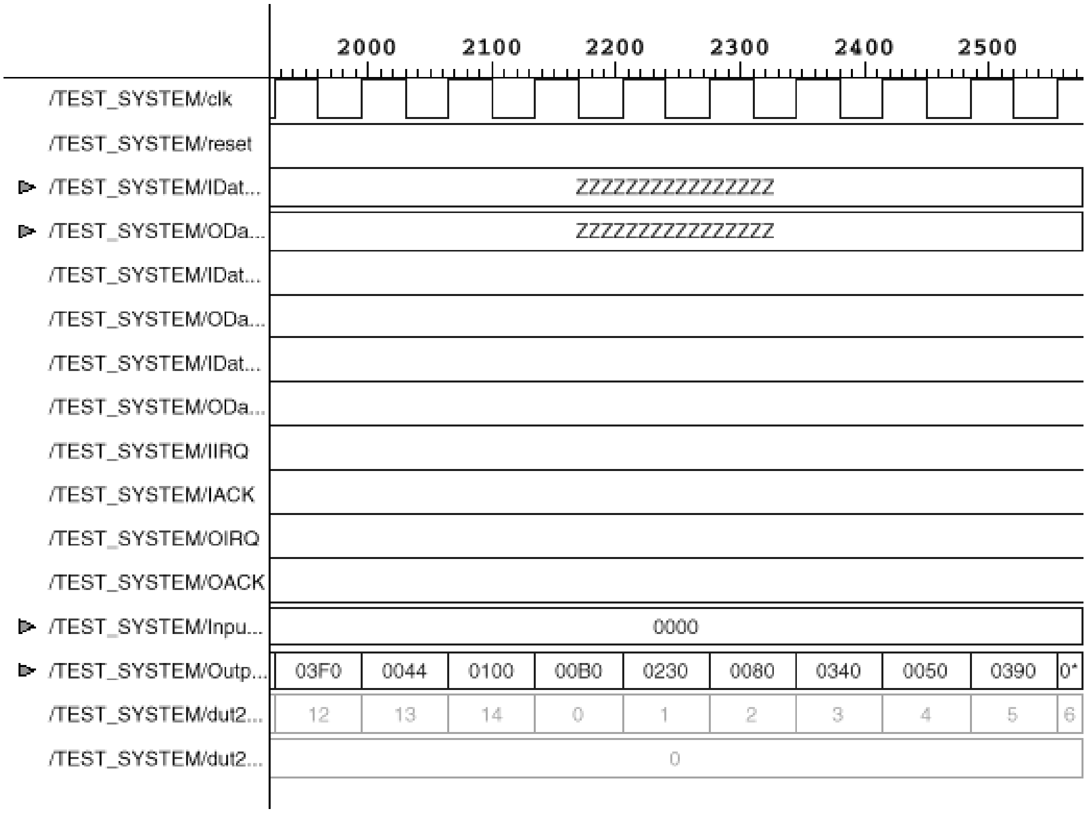
APPENDIX B - FSOI TIMING DIAGRAMS

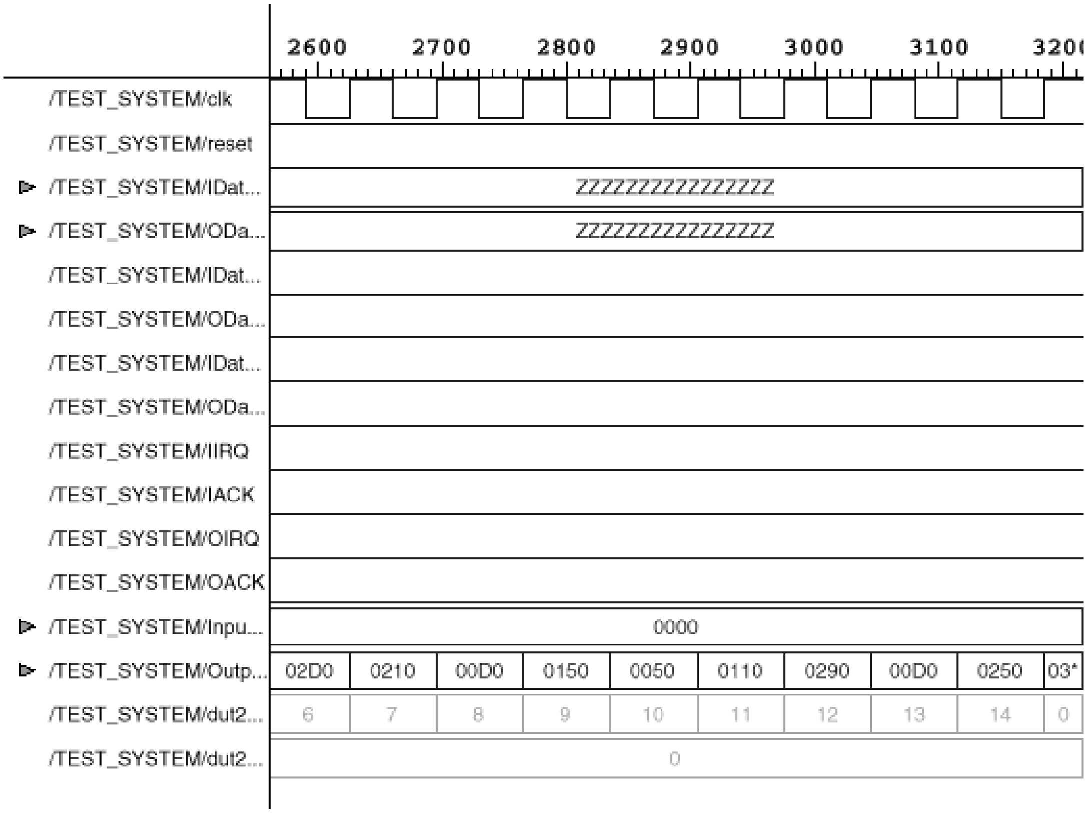
Included on the following pages are the simulation results from Synopsys for the FSOI link.

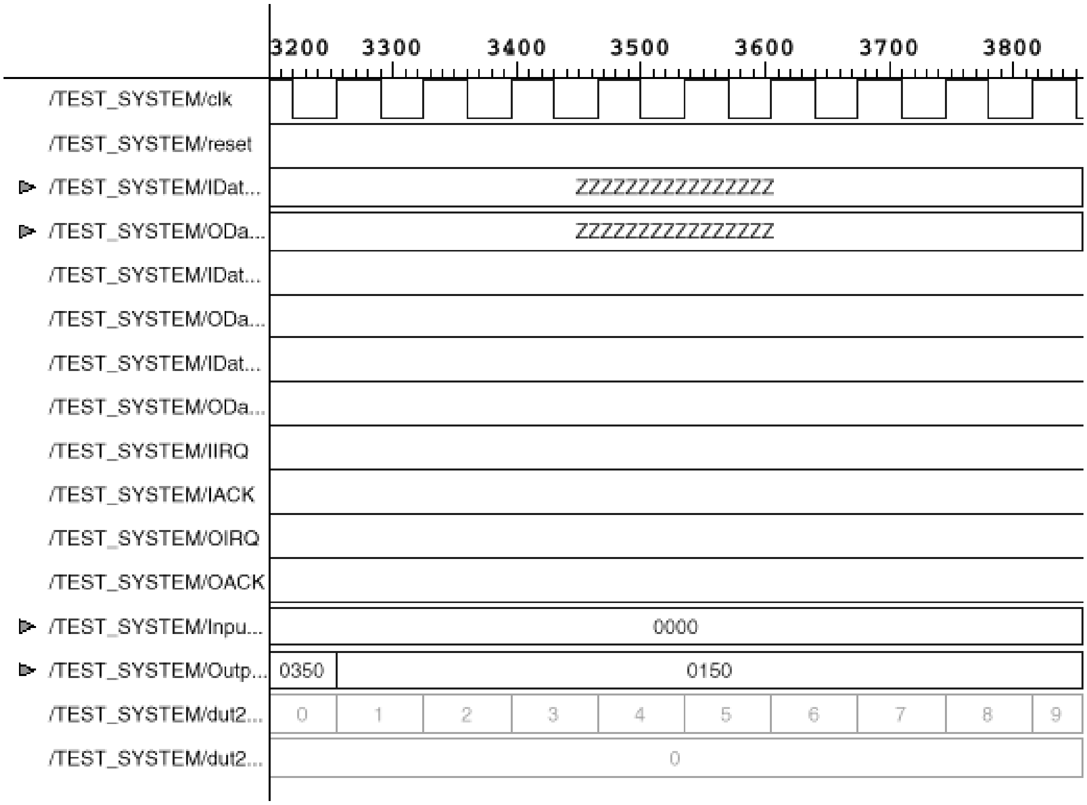

0
100
200
300
400
500
600
/TEST_SYSTEM/clk
/TEST_SYSTEM/reset
/TEST_SYSTEM/IDat...
* ZZZZZZZZ* 0000000* ZZZZZZZZZZZZZZZZ 0000000* 0000000* 0000000* FFFFF*
/TEST_SYSTEM/ODa...
* ZZZZZZZZZZZZZZZZ
/TEST_SYSTEM/IDat...
/TEST_SYSTEM/ODa...
/TEST_SYSTEM/IDat...
/TEST_SYSTEM/ODa...
/TEST_SYSTEM/IIRQ
/TEST_SYSTEM/IACK
/TEST_SYSTEM/OIRQ
/TEST_SYSTEM/OACK
/TEST_SYSTEM/Inpu...
0000
/TEST_SYSTEM/Outp...
0000 0001 0000 0001 0000 0003 0007
/TEST_SYSTEM/dut2...
0 0 1 2 3 4 5 6 7 8
/TEST_SYSTEM/dut2...
0 0

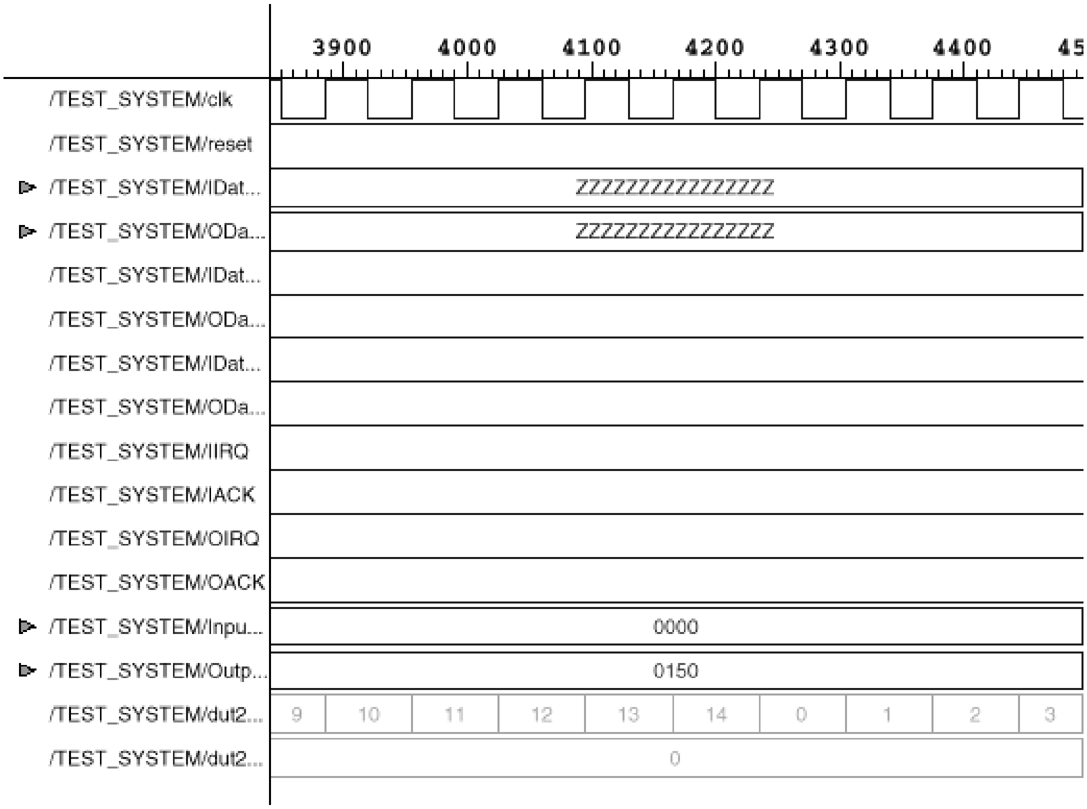

700
800
900
1000
1100
1200
13
/TEST_SYSTEM/clk
/TEST_SYSTEM/reset
/TEST_SYSTEM/IDat...
F* 0000000* FFFFFF* 0000000* FFFFFF* 0000000* ZZZZZZZZZZZZZZZZ
/TEST_SYSTEM/ODa...
ZZZZZZZZZZZZZZZZ
/TEST_SYSTEM/IDat...
/TEST_SYSTEM/ODa...
/TEST_SYSTEM/IDat...
/TEST_SYSTEM/ODa...
/TEST_SYSTEM/IIRQ
/TEST_SYSTEM/IACK
/TEST_SYSTEM/OIRQ
/TEST_SYSTEM/OACK
/TEST_SYSTEM/Inpu...
0000
/TEST_SYSTEM/Outp...
00* 0009 0011 0026 006A 00C8 0158 0344 00A2 003E
/TEST_SYSTEM/dut2...
8 9 10 11 12 13 14 0 1 2
/TEST_SYSTEM/dut2...
0

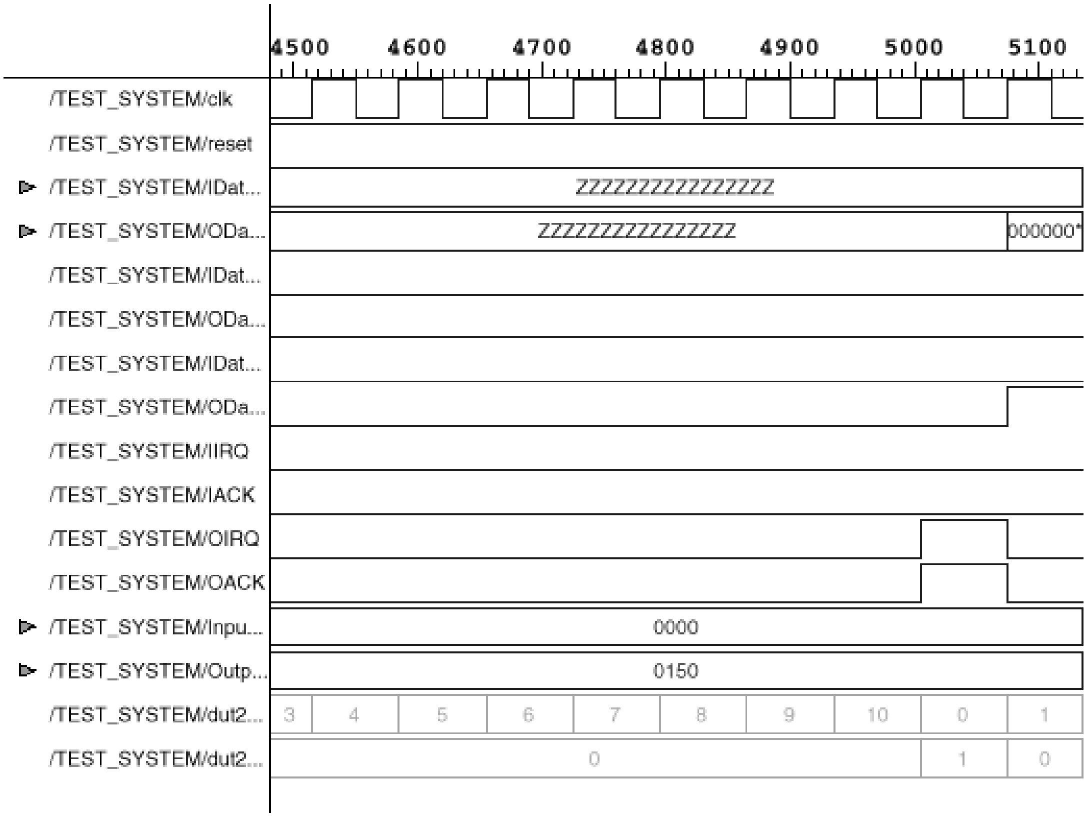

1300
1400
1500
1600
1700
1800
1900
/TEST_SYSTEM/clk
/TEST_SYSTEM/reset
/TEST_SYSTEM/IDat...
ZZZZZZZZZZZZZZZZ
/TEST_SYSTEM/ODa...
ZZZZZZZZZZZZZZZZ
/TEST_SYSTEM/IDat...
/TEST_SYSTEM/ODa...
/TEST_SYSTEM/IDat...
/TEST_SYSTEM/ODa...
/TEST_SYSTEM/IIRQ
/TEST_SYSTEM/IACK
/TEST_SYSTEM/OIRQ
/TEST_SYSTEM/OACK
/TEST_SYSTEM/Inpu...
0000
/TEST_SYSTEM/Outp...
*
02BE
03D8
02E4
02E8
0388
0314
00C8
0040
02E0
*
/TEST_SYSTEM/dut2...
2
3
4
5
6
7
8
9
10
11
*
/TEST_SYSTEM/dut2...
0

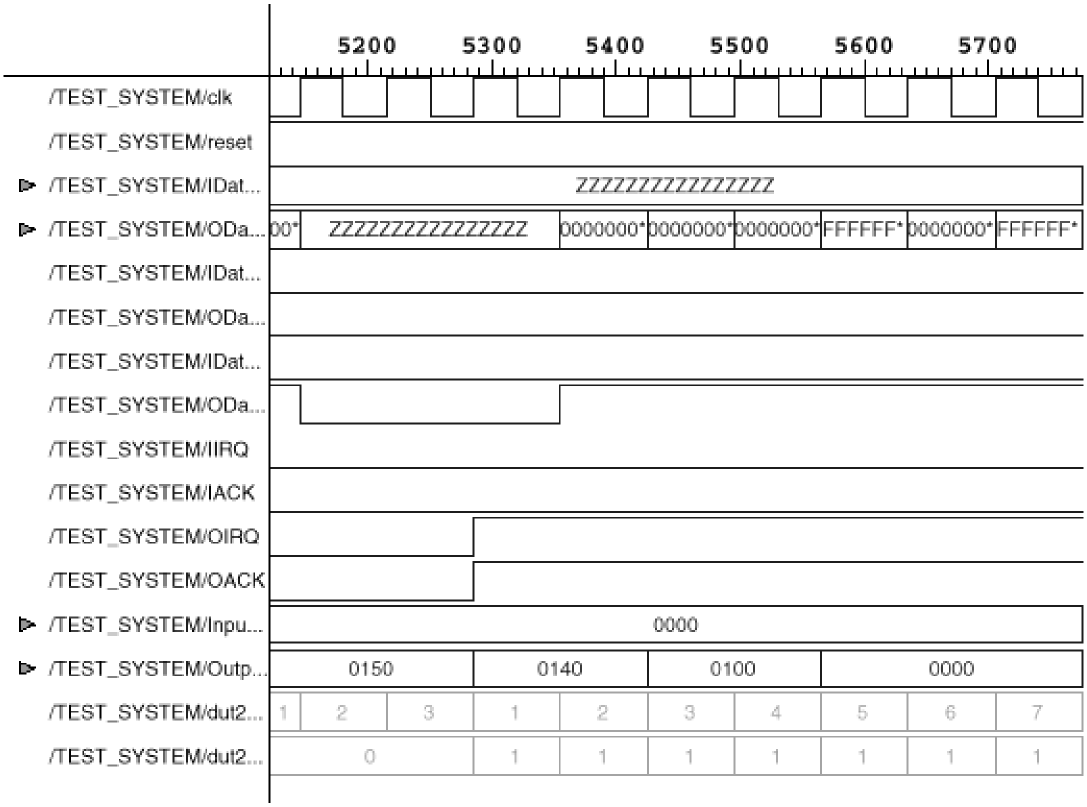

2000
2100
2200
2300
2400
2500
/TEST_SYSTEM/clk
/TEST_SYSTEM/reset
/TEST_SYSTEM/IDat...
ZZZZZZZZZZZZZZZZ
/TEST_SYSTEM/ODa...
ZZZZZZZZZZZZZZZZ
/TEST_SYSTEM/IDat...
/TEST_SYSTEM/ODa...
/TEST_SYSTEM/IDat...
/TEST_SYSTEM/ODa...
/TEST_SYSTEM/IIRQ
/TEST_SYSTEM/IACK
/TEST_SYSTEM/OIRQ
/TEST_SYSTEM/OACK
/TEST_SYSTEM/Inpu...
0000
/TEST_SYSTEM/Outp...
03F0
0044
0100
00B0
0230
0080
0340
0050
0390
0*
/TEST_SYSTEM/dut2...
12
13
14
0
1
2
3
4
5
6
/TEST_SYSTEM/dut2...
0

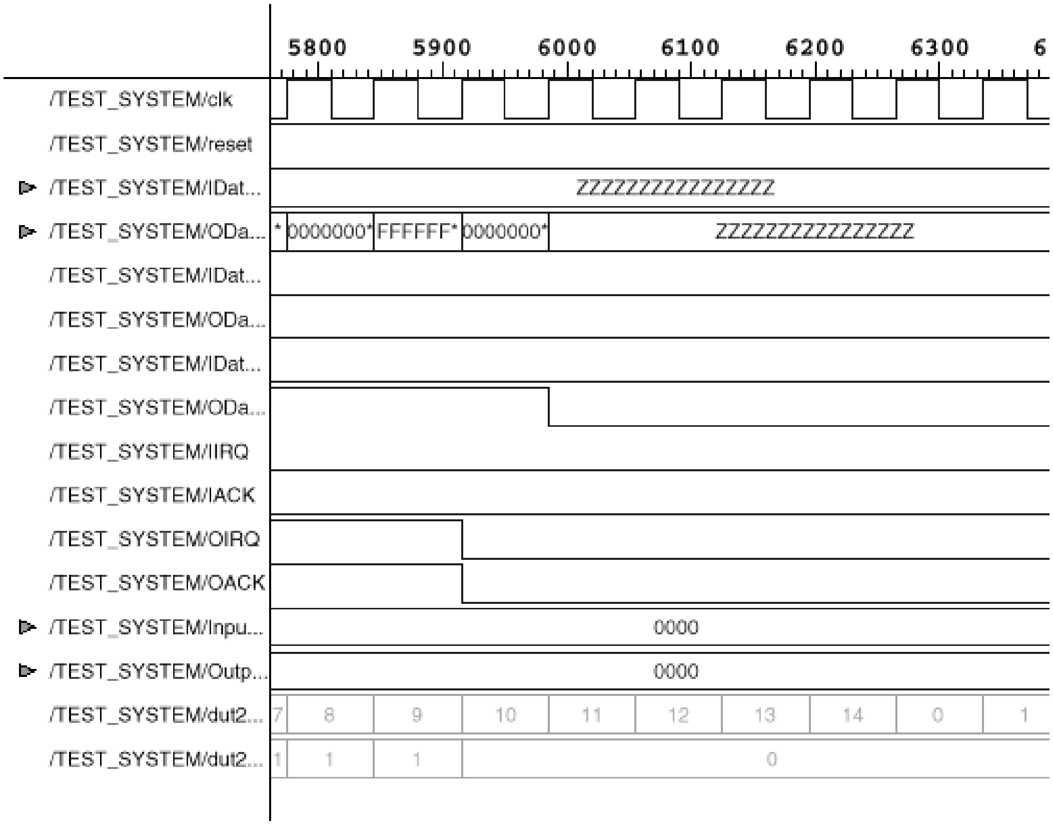

2600
2700
2800
2900
3000
3100
3200
/TEST_SYSTEM/clk
/TEST_SYSTEM/reset
/TEST_SYSTEM/IDat...
/TEST_SYSTEM/ODa...
/TEST_SYSTEM/IDat...
/TEST_SYSTEM/ODa...
/TEST_SYSTEM/IDat...
/TEST_SYSTEM/ODa...
/TEST_SYSTEM/IIRQ
/TEST_SYSTEM/IACK
/TEST_SYSTEM/OIRQ
/TEST_SYSTEM/OACK
/TEST_SYSTEM/Inpu...
/TEST_SYSTEM/Outp...
/TEST_SYSTEM/dut2...
/TEST_SYSTEM/dut2...
ZZZZZZZZZZZZZZZZ
ZZZZZZZZZZZZZZZZ
0000
02D0
0210
00D0
0150
0050
0110
0290
00D0
0250
03*
6
7
8
9
10
11
12
13
14
0
0


3200
3300
3400
3500
3600
3700
3800
/TEST_SYSTEM/clk
/TEST_SYSTEM/reset
/TEST_SYSTEM/IDat...
/TEST_SYSTEM/ODa...
/TEST_SYSTEM/IDat...
/TEST_SYSTEM/ODa...
/TEST_SYSTEM/IDat...
/TEST_SYSTEM/ODa...
/TEST_SYSTEM/IIRQ
/TEST_SYSTEM/IACK
/TEST_SYSTEM/OIRQ
/TEST_SYSTEM/OACK
/TEST_SYSTEM/Inpu...
/TEST_SYSTEM/Outp...
/TEST_SYSTEM/dut2...
/TEST_SYSTEM/dut2...
ZZZZZZZZZZZZZZZZ
ZZZZZZZZZZZZZZZZ
0000
0350
0150
0
1
2
3
4
5
6
7
8
9
0


3900
4000
4100
4200
4300
4400
45
/TEST_SYSTEM/clk
/TEST_SYSTEM/reset
/TEST_SYSTEM/IDat...
/TEST_SYSTEM/ODa...
/TEST_SYSTEM/IDat...
/TEST_SYSTEM/ODa...
/TEST_SYSTEM/IDat...
/TEST_SYSTEM/ODa...
/TEST_SYSTEM/IIRQ
/TEST_SYSTEM/IACK
/TEST_SYSTEM/OIRQ
/TEST_SYSTEM/OACK
/TEST_SYSTEM/Inpu...
/TEST_SYSTEM/Outp...
/TEST_SYSTEM/dut2...
/TEST_SYSTEM/dut2...
ZZZZZZZZZZZZZZZZ
ZZZZZZZZZZZZZZZZ
0000
0150
9
10
11
12
13
14
0
1
2
3
0

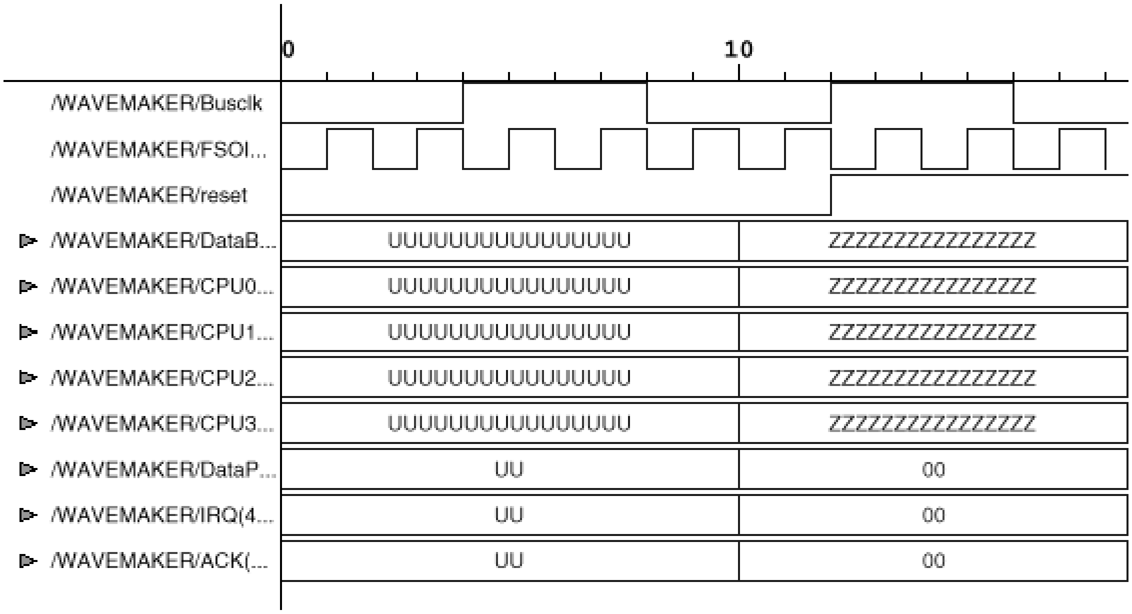

4500
4600
4700
4800
4900
5000
5100
/TEST_SYSTEM/clk
/TEST_SYSTEM/reset
/TEST_SYSTEM/IDat...
/TEST_SYSTEM/ODa...
/TEST_SYSTEM/IDat...
/TEST_SYSTEM/ODa...
/TEST_SYSTEM/IDat...
/TEST_SYSTEM/ODa...
/TEST_SYSTEM/IIRQ
/TEST_SYSTEM/IACK
/TEST_SYSTEM/OIRQ
/TEST_SYSTEM/OACK
/TEST_SYSTEM/Inpu...
/TEST_SYSTEM/Outp...
/TEST_SYSTEM/dut2...
/TEST_SYSTEM/dut2...
ZZZZZZZZZZZZZZZZ
ZZZZZZZZZZZZZZZZ
000000*
0000
0150
3
4
5
6
7
8
9
10
0
1
0
1
0

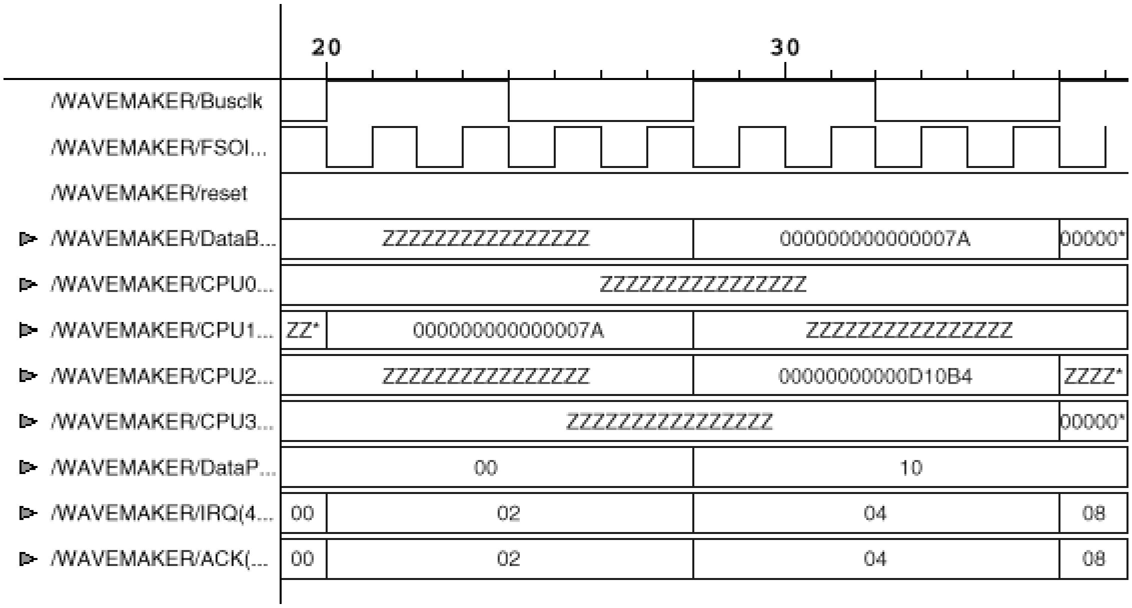

5200
5300
5400
5500
5600
5700
/TEST_SYSTEM/clk
/TEST_SYSTEM/reset
/TEST_SYSTEM/IDat...
ZZZZZZZZZZZZZZZZ
/TEST_SYSTEM/ODa...
00*
ZZZZZZZZZZZZZZZZ
0000000*
0000000*
0000000*
FFFFFF*
0000000*
FFFFFF*
/TEST_SYSTEM/IDat...
/TEST_SYSTEM/ODa...
/TEST_SYSTEM/IDat...
/TEST_SYSTEM/ODa...
/TEST_SYSTEM/IIRQ
/TEST_SYSTEM/IACK
/TEST_SYSTEM/OIRQ
/TEST_SYSTEM/OACK
/TEST_SYSTEM/Inpu...
0000
/TEST_SYSTEM/Outp...
0150
0140
0100
0000
/TEST_SYSTEM/dut2...
1
2
3
1
2
3
4
5
6
7
/TEST_SYSTEM/dut2...
0
1
1
1
1
1
1
1

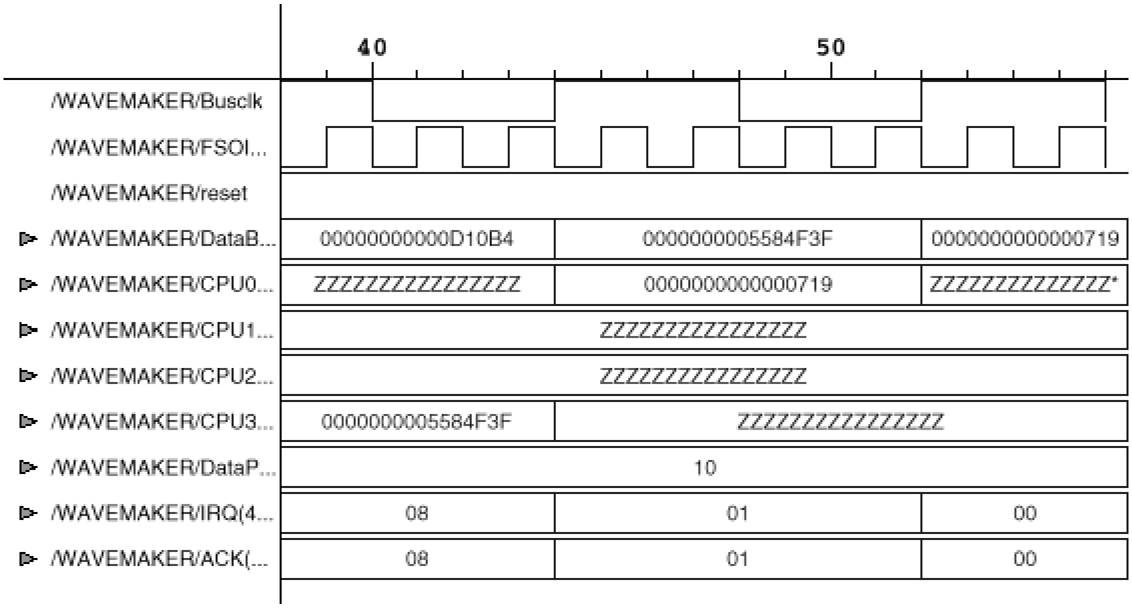

5800
5900
6000
6100
6200
6300
6
/TEST_SYSTEM/clk
/TEST_SYSTEM/reset
/TEST_SYSTEM/IDat...
ZZZZZZZZZZZZZZZZ
/TEST_SYSTEM/ODa...
*
0000000*
FFFFFF*
0000000*
ZZZZZZZZZZZZZZZZ
/TEST_SYSTEM/IDat...
/TEST_SYSTEM/ODa...
/TEST_SYSTEM/IDat...
/TEST_SYSTEM/ODa...
/TEST_SYSTEM/IIRQ
/TEST_SYSTEM/IACK
/TEST_SYSTEM/OIRQ
/TEST_SYSTEM/OACK
/TEST_SYSTEM/Inpu...
0000
/TEST_SYSTEM/Outp...
0000
/TEST_SYSTEM/dut2...
7
8
9
10
11
12
13
14
0
1
/TEST_SYSTEM/dut2...
1
1
1
0

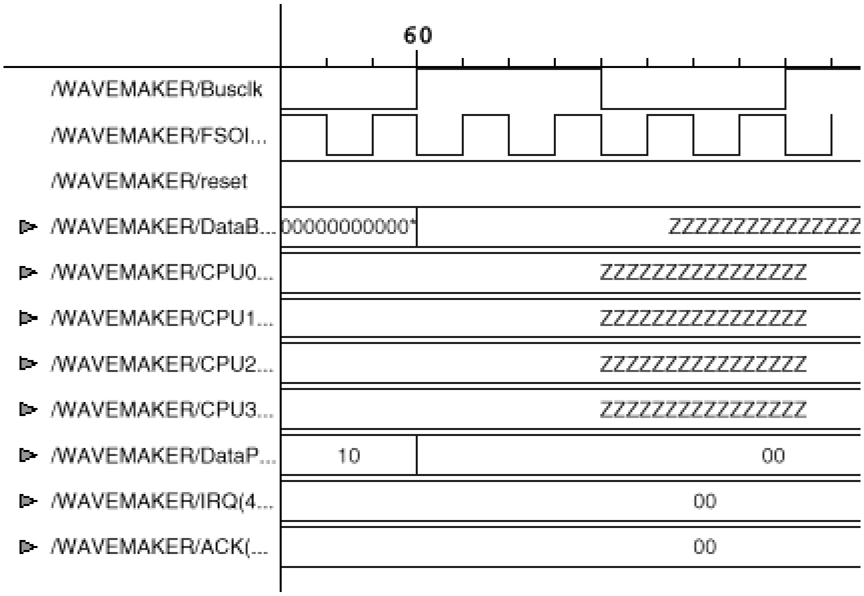
APPENDIX C - DIE TIMING DIAGRAMS

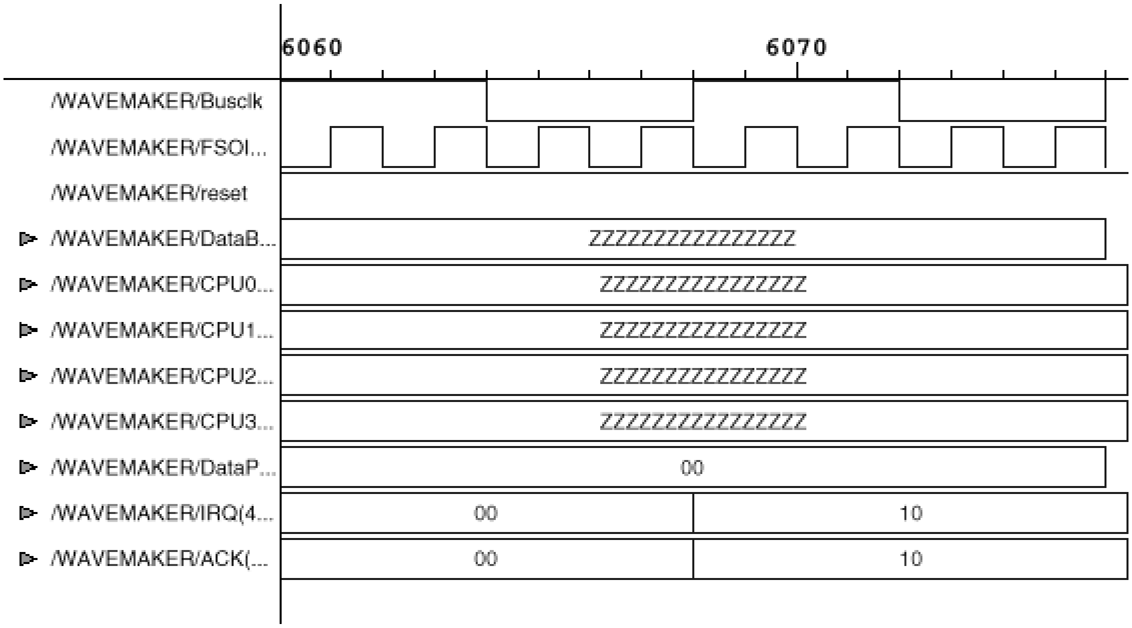
Included on the following pages are the simulation results from Synopsys for the local bus operation.

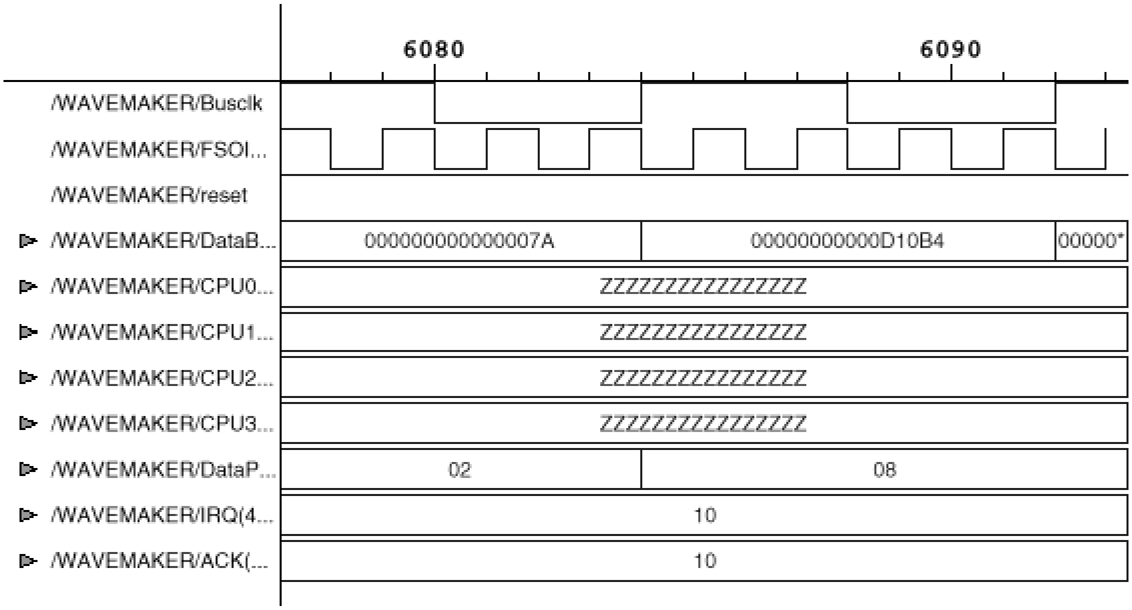

0
10
/WAVEMAKER/Busclk
/WAVEMAKER/FSOI...
/WAVEMAKER/reset
/WAVEMAKER/DataB...
/WAVEMAKER/CPU0...
/WAVEMAKER/CPU1...
/WAVEMAKER/CPU2...
/WAVEMAKER/CPU3...
/WAVEMAKER/DataP...
/WAVEMAKER/IRQ(4...
/WAVEMAKER/ACK(...
UUUUUUUUUUUUUUUU
ZZZZZZZZZZZZZZZZ
UUUUUUUUUUUUUUUU
ZZZZZZZZZZZZZZZZ
UUUUUUUUUUUUUUUU
ZZZZZZZZZZZZZZZZ
UUUUUUUUUUUUUUUU
ZZZZZZZZZZZZZZZZ
UUUUUUUUUUUUUUUU
ZZZZZZZZZZZZZZZZ
UU
00
UU
00
UU
00

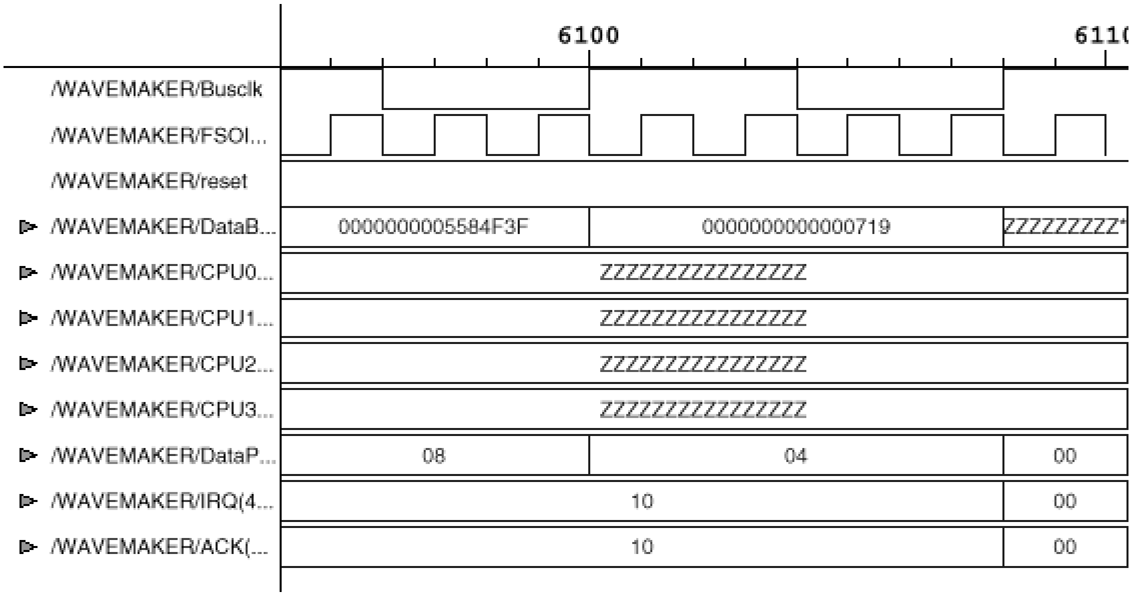

20
30
/WAVEMAKER/Busclk
/WAVEMAKER/FSOI...
/WAVEMAKER/reset
/WAVEMAKER/DataB...
/WAVEMAKER/CPU0...
/WAVEMAKER/CPU1...
/WAVEMAKER/CPU2...
/WAVEMAKER/CPU3...
/WAVEMAKER/DataP...
/WAVEMAKER/IRQ(4...
/WAVEMAKER/ACK(...
ZZZZZZZZZZZZZZZZ
000000000000007A
00000*
ZZZZZZZZZZZZZZZZ
ZZ*
000000000000007A
ZZZZZZZZZZZZZZZZ
ZZZZZZZZZZZZZZZZ
00000000000D10B4
ZZZZ*
ZZZZZZZZZZZZZZZZ
00000*
00
10
00
02
04
08
00
02
04
08


40
50
/WAVEMAKER/Busclk
/WAVEMAKER/FSOI...
/WAVEMAKER/reset
/WAVEMAKER/DataB...
00000000000D10B4
0000000005584F3F
0000000000000719
/WAVEMAKER/CPU0...
ZZZZZZZZZZZZZZZZ
0000000000000719
ZZZZZZZZZZZZZZZ*
/WAVEMAKER/CPU1...
ZZZZZZZZZZZZZZZZ
/WAVEMAKER/CPU2...
ZZZZZZZZZZZZZZZZ
/WAVEMAKER/CPU3...
0000000005584F3F
ZZZZZZZZZZZZZZZZ
/WAVEMAKER/DataP...
10
/WAVEMAKER/IRQ(4...
08
01
00
/WAVEMAKER/ACK(...
08
01
00


60
/WAVEMAKER/Busclk
/WAVEMAKER/FSOI...
/WAVEMAKER/reset
/WAVEMAKER/DataB...
00000000000*
ZZZZZZZZZZZZZZZZ
/WAVEMAKER/CPU0...
ZZZZZZZZZZZZZZZZ
/WAVEMAKER/CPU1...
ZZZZZZZZZZZZZZZZ
/WAVEMAKER/CPU2...
ZZZZZZZZZZZZZZZZ
/WAVEMAKER/CPU3...
ZZZZZZZZZZZZZZZZ
/WAVEMAKER/DataP...
10
00
/WAVEMAKER/IRQ(4...
00
/WAVEMAKER/ACK(...
00


6060
6070
/WAVEMAKER/Busclk
/WAVEMAKER/FSOI...
/WAVEMAKER/reset
/WAVEMAKER/DataB...
/WAVEMAKER/CPU0...
/WAVEMAKER/CPU1...
/WAVEMAKER/CPU2...
/WAVEMAKER/CPU3...
/WAVEMAKER/DataP...
/WAVEMAKER/IRQ(4...
/WAVEMAKER/ACK(...
ZZZZZZZZZZZZZZZZ
ZZZZZZZZZZZZZZZZ
ZZZZZZZZZZZZZZZZ
ZZZZZZZZZZZZZZZZ
ZZZZZZZZZZZZZZZZ
00
00
10
00
10


6080
6090
/WAVEMAKER/Busclk
/WAVEMAKER/FSOI...
/WAVEMAKER/reset
/WAVEMAKER/DataB...
/WAVEMAKER/CPU0...
/WAVEMAKER/CPU1...
/WAVEMAKER/CPU2...
/WAVEMAKER/CPU3...
/WAVEMAKER/DataP...
/WAVEMAKER/IRQ(4...
/WAVEMAKER/ACK(...
000000000000007A
00000000000D10B4
00000*
ZZZZZZZZZZZZZZZZ
ZZZZZZZZZZZZZZZZ
ZZZZZZZZZZZZZZZZ
ZZZZZZZZZZZZZZZZ
02
08
10
10


6100
611(
/WAVEMAKER/Busclk
/WAVEMAKER/FSOI...
/WAVEMAKER/reset
/WAVEMAKER/DataB...
/WAVEMAKER/CPU0...
/WAVEMAKER/CPU1...
/WAVEMAKER/CPU2...
/WAVEMAKER/CPU3...
/WAVEMAKER/DataP...
/WAVEMAKER/IRQ(4...
/WAVEMAKER/ACK(...
0000000005584F3F
0000000000000719
ZZZZZZZZZ*
ZZZZZZZZZZZZZZZZ
ZZZZZZZZZZZZZZZZ
ZZZZZZZZZZZZZZZZ
ZZZZZZZZZZZZZZZZ
08
04
00
10
00
10
00