A Two-Dimensional Extensible Bus Technology and Protocol for VLSI Processor Core

A thesis presented in partial fulfilment of the requirements for the degree of

Master of Engineering

In

Computer and Electronic Engineering

at School of Engineering and Advanced Technology,

Massey University, Albany

New Zealand

By

Loke Chun Eng

2011

Main Advisor:

Dr. S.M. Rezaul Hasan

Abstract

Intellectual property (IP) core design modularity and reuse in Very-Large-Scale-Integration (VLSI) silicon have been the key focus areas in design productivity improvement in order to shorten product development lead time as well as minimize design error on new product [11]. The System-On-Chip (SoC) design approach has been adopted in microprocessor design flow with many functional blocks reuse in silicon. SoC has the advantage of cost efficiency and higher fabrication yield. The fundamental building block of SoC is the interconnection of intellectual property (IP) core through a shared bus to establish an on-chip communication. As IP core integration is severely constraint by silicon wafer sizes (cost per die), the right level of integration is never an easy decision. System-in-Package (SiP) addresses this drawback with package level IP core integration. However, SiP has the drawback of lower fabrication yield which results in higher manufacturing cost [6]. In order to address these issues, a new level of integration has been suggested in order to reduce the drawbacks of SiP and SoC approaches. This new integration methodology is also known as System-in-System (SiS) which emulates SoC and SiP at the system level.

The thesis contains a detailed treatment on the processor architecture and SoC used. The design methodologies have been discussed too.

The thesis also contains treatment on the verification methodologies and technologies that are used in design validation.

Research includes the design of two dimensional XBUS system for external IP core integration on SoC. The thesis proposed a system level bus for IP integration through the XBUS. As there are multiple ways of integrating IP core at the system level, the XBUS is limited to two channels (hence two dimensional) in order to simplify implementation complexities.

Based on experimental results, the proposed method can be introduced as a very promising method for the design of SoC and various other high-performance computer systems.

Acknowledgement

First and foremost, I would like to offer my deepest gratitude to the supervisor of this research: Dr. S.M. Rezaul Hasan, who, with his guidance allowed for the completion of this dissertation. Without his help and support throughout the research it would have been impossible to complete.

As usual, the unconditional support of my family and loved ones is something always appreciated; as such, I would like to acknowledge my mother and father; sister and friends. Their support, both direct and indirect, provided a bastion of confidence during times of difficulty.

For those who I have gained knowledge from indirectly, your work has provided a rich source of information that has furthered my own abilities, and I thank you.

Lastly, I would like to thank the staff and lecturers of Massey University's School of Engineering and Advanced Technology at Albany for the interest shown in the project and their freely given advice.

Working towards the Master's Degree in Massey University was the most important, amazing and astonishing experience in my life. This research and training has completely changed the way of my thinking toward problem solving.

Table of contents

List of illustrations

Chapter 1: Introduction

1.1 Problem Description

The introduction of the microprocessor, which was originally constructed for electronic calculator, has inadvertently revolutionized computer technology from embedded processing toward application-rich multi-purpose computing platform. From a humble beginning, continued increase in micro-processor capacity has rendered other forms of computing devices possible and that include the contemporary smart-technology and smart-phone [29]. The outburst of integrated circuit (IC) complexity, as predicted by Moore's Law plus the very exceptional manufacturing advances that bring IC nanotechnology to fruition, are driving the current semiconductor industry to challenge another cutting edge revolution: *System-on-Chip* (SoC) (Figure 1.1) which generally refers to the integration of all components of a computer and peripheral controllers into a single chip to form an entire electronic system. As transistors get smaller they get cheaper, faster and consume less power. The main contribution of this research is the development of external bus system for direct integration of multiple homogeneous or heterogeneous electronic systems.

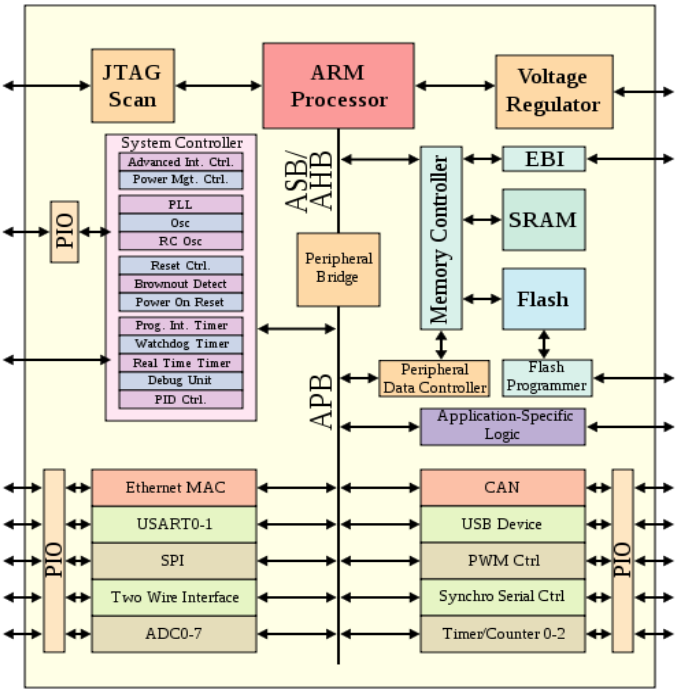

Figure 1.1: A complete System-on-the-chip

1.2 Motivation

"Having different modules on different dice permits a plug-and-play approach to a range of markets. You can do several different RF designs and use the appropriate one for each market segment, without having to change the baseband logic chip, for instance. With an SoC, you are stuck with whatever you chose to put on the die." Pieter Hooijmans, Philips Semiconductor

Independently from the shift in silicon revolution, processor architecture has evolved dramatically in the last decade. Modern computer system achieves high performance through a combination of advances in computer architecture and improvements in manufacturing technology. One consequence of these advancements is the evolution of Field-Programmable Gate Array (FPGA) which carries enough resources to implement complex embedded system on a single device or multiple devices. FPGA comprises configurable interconnects, large memory and hardwired arithmetic blocks and an array of configurable Look-up table (LUT) [17]. Further refinement in FPGA technology has led to the integration of analogue intellectual property (IP) cores and RISC processors such as the Micro-Blaze and Power-PC. The major advantage of FPGA over custom IC is that it relieves the designer from addressing the increasingly complicated IC physical design flow (Figure 1.2). Inherent re-configurability is another added plus for FPGA. This device has some drawbacks in that they carry extra overheads versus cost and lower overall system performance. FPGA consumes more power compare with custom IC. However, considering the NRE (Non-Return Engineering) cost for IC fabrication, the FPGAs can be applied to a number of applications as prototypes or as part of the final product.

The following section discusses in detail about the commonly found and off-the-shelf VLSI processor architectures.

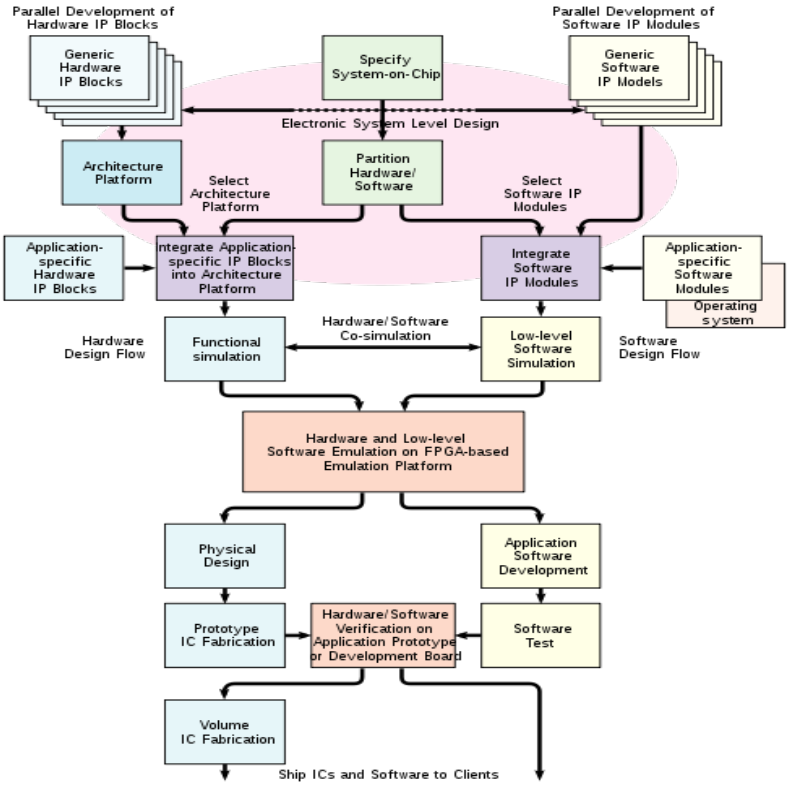

Figure 1.2: Typical IC design flow

i) RISC versus CISC

General purpose processors are finite-state automations that execute instruction held in a memory (hence the stored-program model); every instruction defines a particular way the total state should change and it also defines the next instruction to be executed. These devices are further categorized by their processor architectures, i.e.: Reduce Instruction Set Computers (RISC) and Complex Instruction Set Computers (CISC). RISC type processor executes small instructions (hence a small instruction set) tailored for specific tasks and generally performs faster compared to CISC type processors for the same task [13]. CISC instructions tend to be large and perform more functions. The instruction set for CISC type processors assimilates high-level language thence requires less machine code for the same task [14]. However, these functions are rarely used by the compiler and results in a poorer performance compare to RISC type processors.

ii) Harvard versus Princeton

The program stored in the memory feeds the CPU (Central Processing Unit) with instruction in order to execute a function. The Princeton architecture (Von Neumann) machine stores the control program, variables and other data structures in a common memory space. This results in simpler interface to the memory space. The Princeton architecture's memory interface unit is responsible for arbitrating access to the memory space between reading instructions and

passing data back and forth to the processor. This architecture adds a pre-fetch stage in the instruction pipeline to reduce bottleneck [16]. In contrast, the Harvard architecture machine uses separate memory banks for program storage, processor stack and variable RAM [15]. And this results in less instruction cycles as the pre-fetch stage is not needed. However, it lacks the flexibility to process large amounts of memory from different sources (compared to Von Neumann) and have to access this small amount of memory very quickly.

The section below elaborates in detail the goal of this research.

1.3 Extensible bus (XBUS)

The fundamental building blocks of a system-on-chip (SoC) are its intellectual property (IP) cores which are reusable hardware blocks designed to perform a particular task of a given component. Different IP cores are interconnected on SoC by a communication structure such as a shared bus or *network-on-chip* (NoC) in order to establish communication amongst them. This model is used as a ground for extensible bus (XBUS) design which provides a fabric for communication at the system level with the internal components (hence external processing), which is the aim of this research. Bus network is a rapidly growing division of communication industry in which high quality information (or data) can be transferred at high speed between devices located anywhere in the world. Broadly speaking, networks can be divided into three main categories, i.e.: Circuit-switched, Message-switched and Packet-switched. In the circuit-switched network, the two communicating data terminal equipment (DTEs) establish a continuous physical link for the entire duration of the communication sessions. Circuit switching is inefficient for variable bit rate transmission or high-bandwidth data serving since the circuit must always support the highest data rate expected [18]. Message-switched network does not require a continuous physical path to exist between the DTEs. Data from a DTE is formatted as message of reasonable length and stored/forwarded at each data network node. Physical connections between the node pairs are made only for the duration of the message transfer between these node pairs and are broken as soon as the message transfer is complete [19]. Packet-switched is in many ways similar to message-switched except that the message is further divided into many standard packets which are then routed individually through the network. Each packet is stored and forwarded at each network node. Messages are reassembled from their constituent packets at the receiving DTE [20].

Serving high bandwidth data transfer workloads would require high data processing throughput. Throughput computing is a technique that takes advantage of the thread-level

parallelism (TLP) and hence concurrent threads. This approach has the advantage that memory stall time of one strand can often be overlapped with execution of other strands on the same processor.

This report discusses the implementation and verification of the DTP-XBUS-2 as a high-bandwidth data transfer protocol. As manufacturing cost is one of the many important factors in the industry. Therefore, a decision is made to use open-source and standard tool-chains to implement this system. Modern industry is also rapidly shifting towards lower cost open-source solutions. The performance, area and power are of significant concerns while implementing the subsystem. The goal of the project is to implement a low-cost DTP system with a satisfactory performance and a comprehensive verification of its protocol. This report has been structured in chapters for the simplicity and easiness. Brief information about the contents of chapters is given as below.

Chapter 2 reviews the literature relevant to the objectives and contributions of this thesis.

Chapter 3 outlines the environment and processor architecture relevant to DTP-XBUS-2. Furthermore, this chapter describes the basic operations of the DTP-XBUS-2 backbone devices. The main emphasis of this chapter is on the development of DTP-XBUS-2.

Chapter 4 provides a short introduction about the basics of verification concepts. It discusses the different types of verification and evaluates the possible alternatives to verify the DTP-XBUS-2.

Chapter 5 discusses the verification setup based on chapter 3. This chapter also describes the framework and development of the test bench used for structural and functional verification of the DTP-XBUS-2 core.

Chapter 6 discusses the results obtained from Power-On Test. This chapter focuses on the performance analysis of DTP-XBUS-2.

Chapter 7 concludes the thesis and highlights the future work.

1.4 The thesis contribution

The major contribution of this thesis lies in design and development of a direct off-chip communication protocol for seamless integration of external IP cores which bypasses inherent pipelining latency of the microprocessor. As discussed in the abstract section about the inherent limitations of SoC and SiP, the objective of this thesis aims to resolve the silicon constraints imposed on SoC and the reliability issues and manufacturing costs associated with SiP. The communication protocol developed or the DTP-XBUS-2 enables IP core integration at the system-level. Unlike current prevailing system level bus such as the Peripheral Component Inter-connect (PCI) and VMEBus, DTP-XBUS-2 bypasses the ISO-OSI protocol stack above layer 3 and has a more specific focus on IP cores communication (Figure 1.4) in contrast to other buses which is more application focused (Figure 1.3).

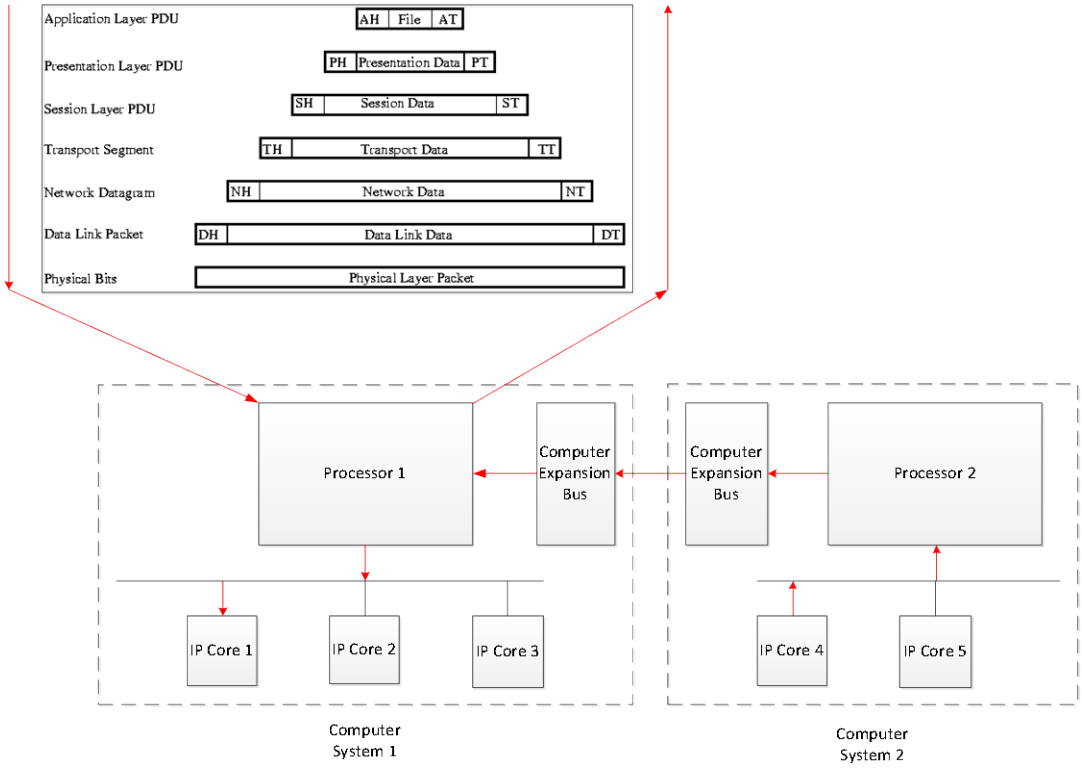

Figure 1.3: Conventional System Level Bus.

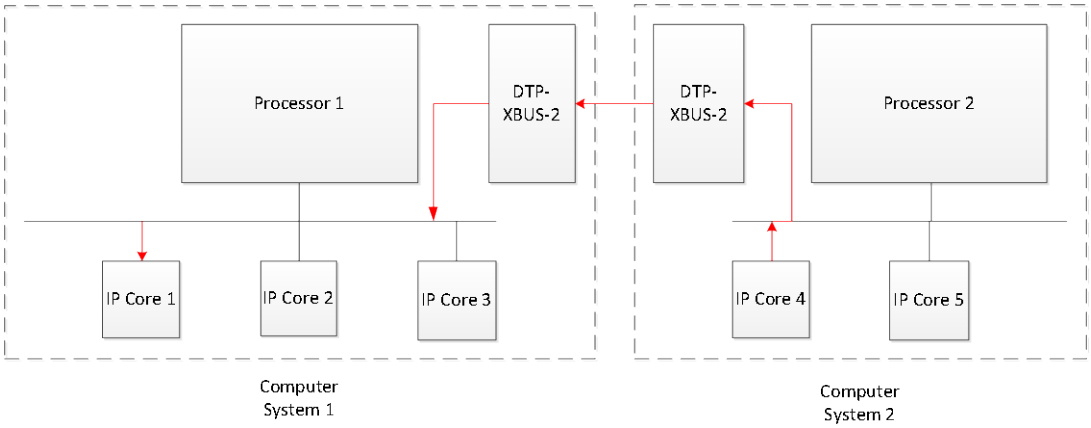

Figure 1.4: DTP-XBUS-2

The experimental results show that DTP-XBUS-2 is reliable and could be implemented with SoC for IP cores integration. The gain in system reliability (low bit error rate) compensates for the performance in SiP.

Chapter 2: Literature Review

This chapter reviews the literature about some existing bus communication protocol suitable for both SoC and SiP.

In general, the performance of a system is dependent on the bus communication efficiency [8]. Thus, efficient bus architecture with optimal arbitration, where contention is reduced, plays an important role in maximizing the performance for all on-chip communications. There are five types of bus architectures for on-chip communication: Global Bus I Architecture (GBIA), Global Bus II Architecture (GBIIA), Bi-Fifo Bus Architecture (BFBA), Crossbar Switch Bus Architecture (CSBA), and CoreConnect Bus Architecture (CCBA). For off-chip communication, VMEBus and Peripheral Component Interconnect (PCI) are the common prevailing bus at the system level. However, these buses are more application focused as discussed in section 1.4 and may not be suitable for IP cores integration. The reviews for on-chip buses are given as below:

2.1 Global Bus I Architecture

GBIA uses two registers DONE_OP and DONE_RV to establish communication between two computing nodes. A flag is set by each node in these registers after the data processing or the data receipt from the corresponding node. Bus bridges are constructed to allow different processors on the same substrate to access data memory. The details of GBIA are illustrated below (Figure 2.1).

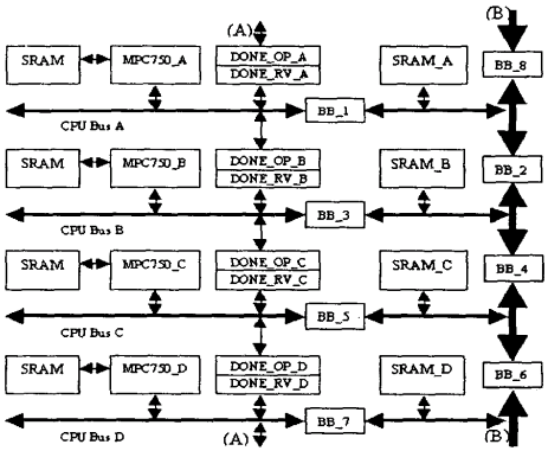

Figure 2.1: Global Bus I Architecture

For example, if MPC 750_A writes to SRAM_A, the address decoder of that processor makes a connection through BB_I to the memory, and BB_2 and BB_8 block the access from any other processors. MPC 750_B then reads from SRAM_A and while the MPC 750_B address decoder attempts to disconnect BB_I from CPU Bus A, BB_2 and BB_3 are re-connected to CPU Bus B by the control of the address decoder. For the handshake operation between two computing node, MPC750_A begins with setting DONE_OP_B register at the completion of its operation. MPC750_B then resets the DONE_OP_B and reads SRAM_A. After MPC750_B completes the read operation, it then sets DONE_RV_B register to "1". MPC750_A terminates the handshake by resetting DONE_RV_B to "0" for subsequent packet transmission.

2.2 Global Bus II Architecture

In this architecture (Figure 2.2), all processing nodes share a common global bus [8] which requires an arbiter to resolve bus contention when two or more computing nodes try to initiate data transmission. The arbiter grants the bus in a similar fashion to First-in-First-out (FiFo) architecture.

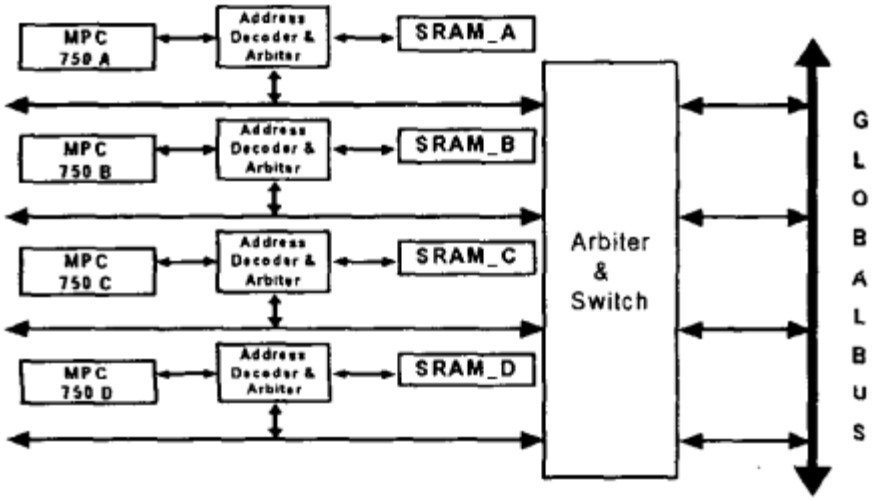

Figure 2.2: Global Bus II Architecture

2.3 Bi-FiFo Bus Architecture

For this architecture (Figure 2.3), the data output by each computing node are exchanged through the Bi-directional FiFo located between the nodes. Each node has two ports reserved

as interfaces to the Bi-FiFo: the upper port, ZZ and lower port, XX. When a node pushes a data to the Bi-FiFo, this data is also made available to the adjacent node.

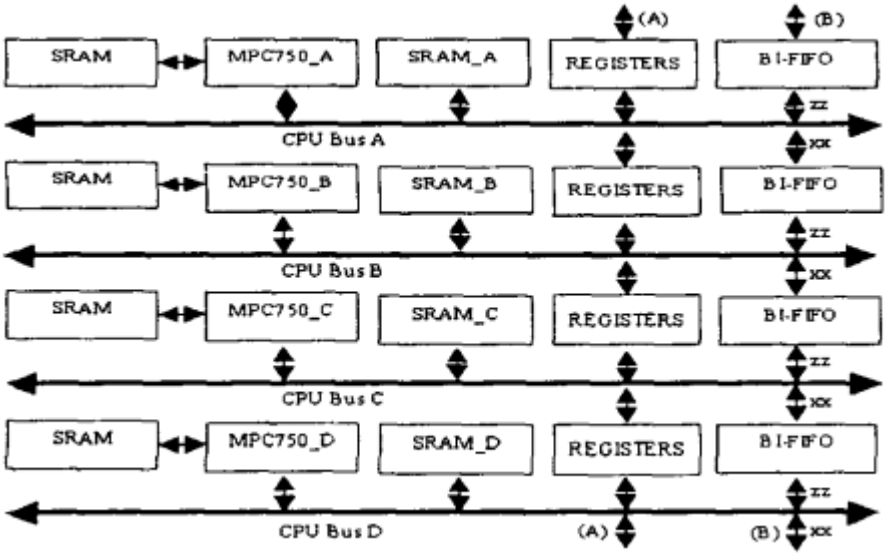

Figure 2.3: Bi-FiFo Bus Architecture.

Both high and low threshold values are defined to indicate that the status of Bi-FiFo: Full and empty. An interrupt signal is used as an indication for the adjacent computing node when the data in a Bi-FiFo reaches the high threshold. The interrupted node continuously read data from the Bi-FiFo until it reaches the low threshold. Communication synchronisation is performed with the interrupt function and two flag registers, TX_DONE and RV_DONE, for handshaking. These registers and the threshold registers are contained in the "REGISTERS" block of figure 2.3.

2.4 Crossbar Switch Bus Architecture

This architecture (Figure 2.4) is derived from GBIIA with the introduction of an array of transmission gates that provide paths between all computing nodes and shared SRAMS as shown in figure 2.4

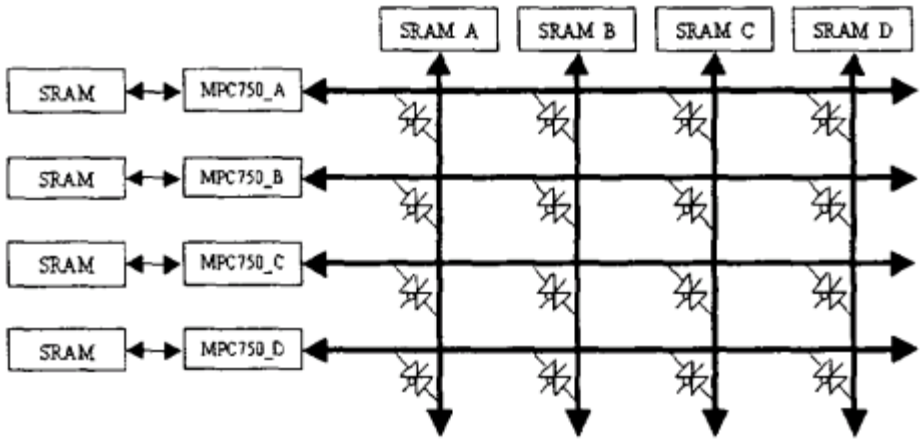

Figure 2.4: Crossbar Switch Bus Architecture

Each computing node accesses any shared SRAM A, B, C, or D at the same time if there is no competition for accessing the same SRAM block. When a contention occurs, an arbiter is used to resolve this situation in a similar fashion to FiFo architecture.

2.5 IBM CoreConnect Bus Architecture

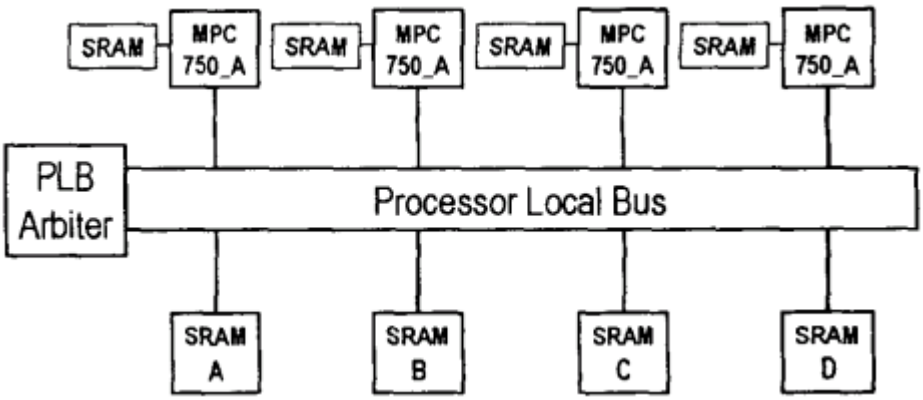

Figure 2.5: IBM CoreConnect Bus

This architecture (Figure 2.5) is similar to the GBIIA bus. An arbiter is used to grant full control of the bus to the computing node according to the priority order in contrast to FIFO fashion as in GBIIA. The memory unit can be designed as a separate slave entity providing simpler interfaces to other processing unit [22].

As noted in [1], high performance computing does solely rely on bus architecture. As operating frequency of microprocessor continues to scale in the GHz range, computer

systems with more efficient communication protocol have demonstrated with higher return of investment in terms of system performance [2][3][4][5]. With the microprocessor clock rate continues to scale as Moore's Law predicted, simultaneous switching noise (SSN) or so-called power/ground bounce noise (GBN) becomes one of the critical issues [30]. The presence of noise in high speed computer system increases the bit error rate which lowers the signal-to-noise ratio or fault tolerance and this significantly impacts the overall system performance [2]. Shunt through currents that may excite the resonance modes of power distribution networks (PDN) are created when high speed digital circuits transit between power and ground planes (Figure 2.6). In return, the resonating PDN causes undesired electromagnetic energy propagation that leads to Inter-Symbol Interference (ISI). ISI continues to be the biggest challenge in SiP design (Figure 2.8) as operating frequency continues to scale.

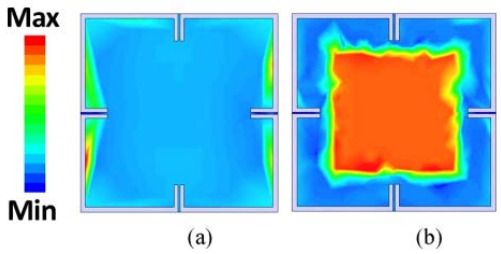

Figure 2.6: Electric field distribution of second order mode in SiP. (a) Long Period Coplanar Electromagnetic Bandgap Power Planes (LPC-EBG) (b) LPC-EBG with multi via ground surface perturbation lattice (MV-GSPL)

SoC architecture attempts to integrate multiple functions, both analogue and digital into a monolithic device as a solution to address Electromagnetic Interference (EMI) and Electromagnetic Compatibility (EMC) issues. However, many integrated functional blocks cannot be optimized due to the inherent limitation of the semiconductor substrate used [31]. As defect density scales with area, the integration of large scale functions such as memory and switch fabrics with small scale functions (Figure 2.7) results in compounded yield impacts.

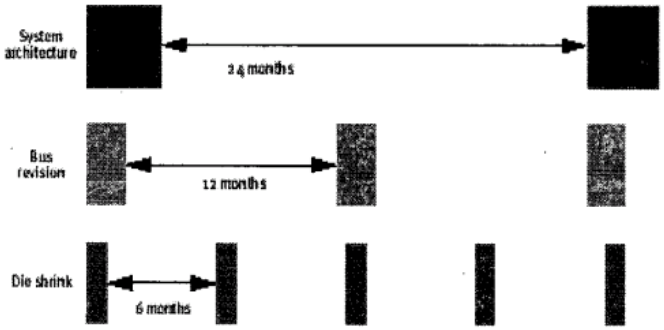

Figure 2.7: Differential rates of system IC upgrades.

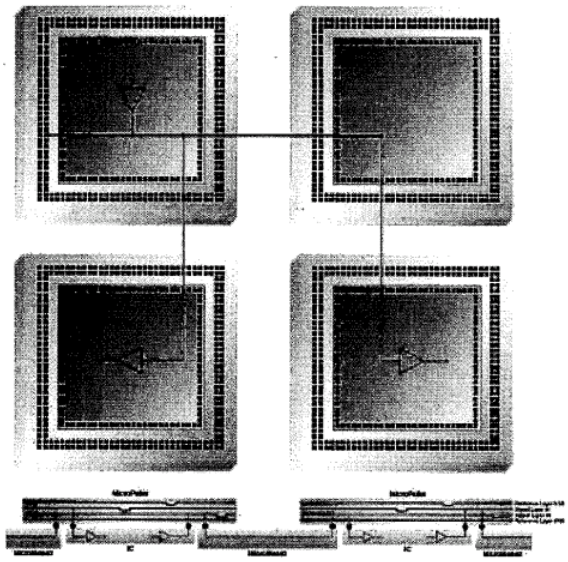

Figure 2.8: SiP system interconnect routing architecture

2.6 The development of DTP-XBUS-2 as SoC-SiP Hybrid

The bus architectures for IP core integration as mentioned before are located on the same substrate. In other words, IP core integration can only be performed for ICs on the same die or package. However, these methodologies contain inherent limitations as discussed previously and are yet to be solved [6][12][30][31]. With this background, DTP-XBUS-2 has been developed as a system level bus dedicated for IP cores integration. As demonstrated in [32], at the system level, EMI/EMC is more predictable and could be minimized under proper configurations (Figure 2.9, Figure 2.10, Figure 2.11, Figure 2.12).

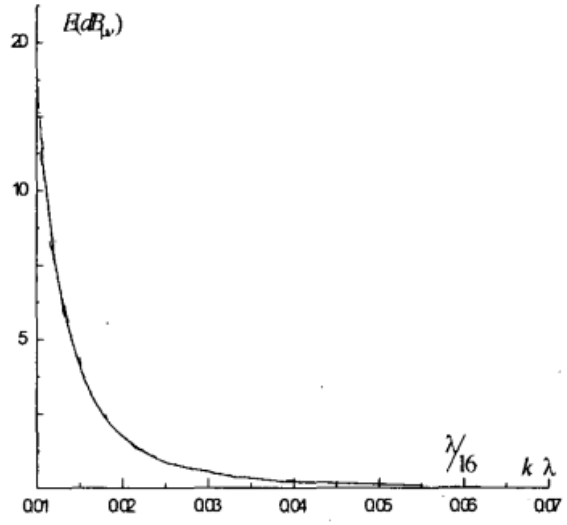

Figure 2.9: Radiative electric field of common-mode current varying with the distance arranged strips, clock frequency f=500MHz.

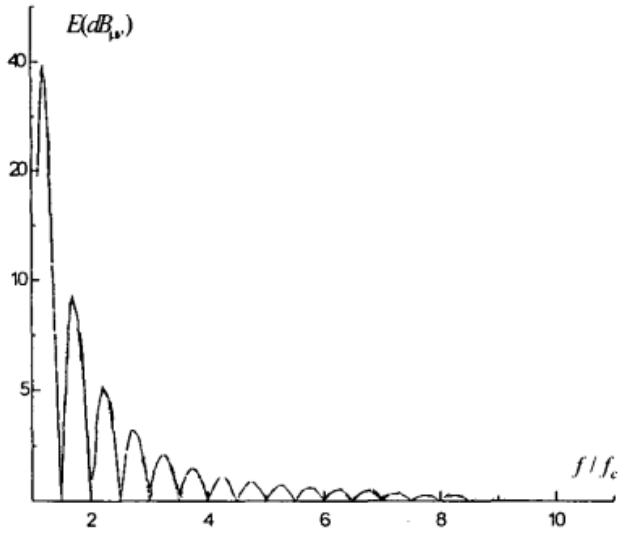

Figure 2.10: Spectral density of radiative electric field of common-mode current varying from fc to 10fc, fc=100MHz. The distance from a clock strip to other strip is λ/16.

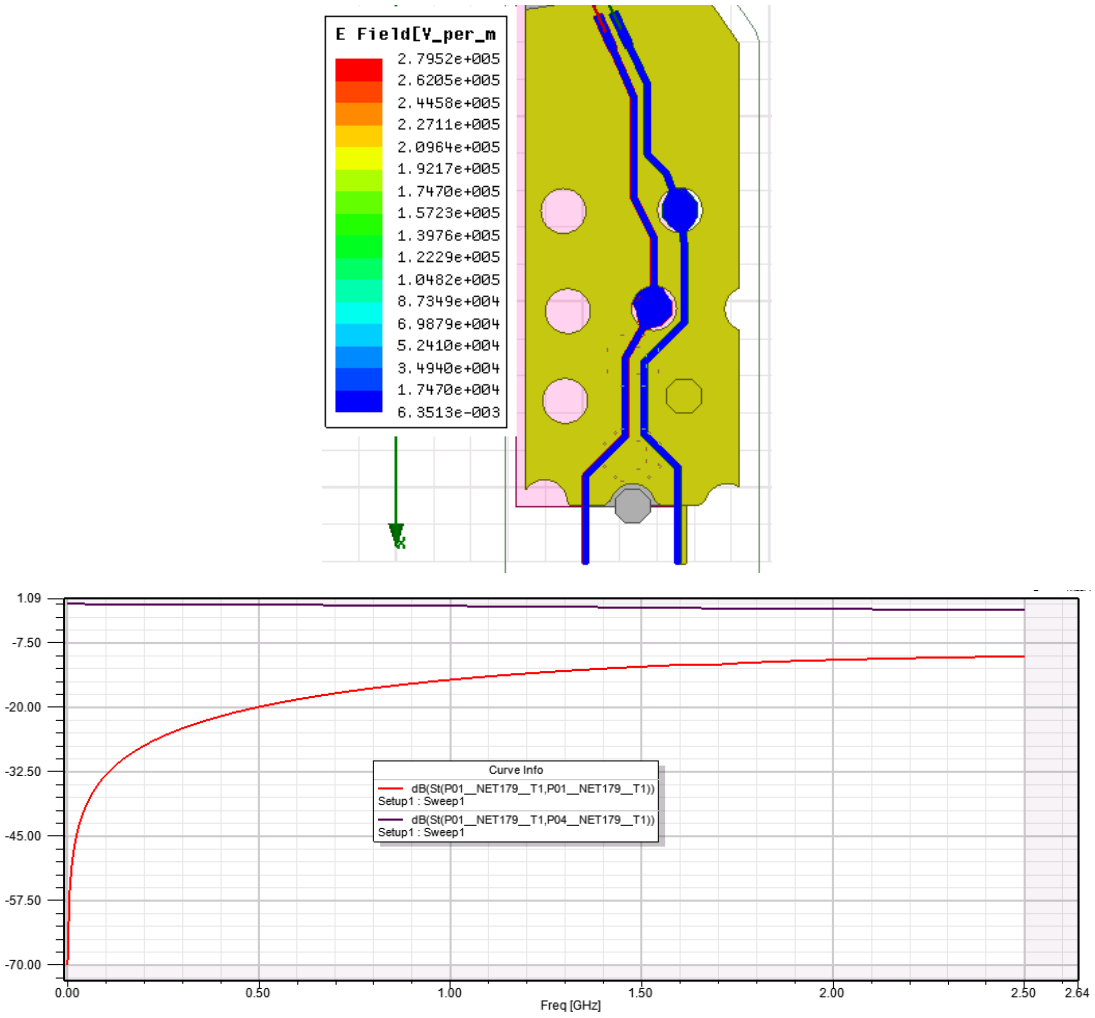

Figure 2.11: Clock strip analysis and S-Parameters. Refer Appendix C

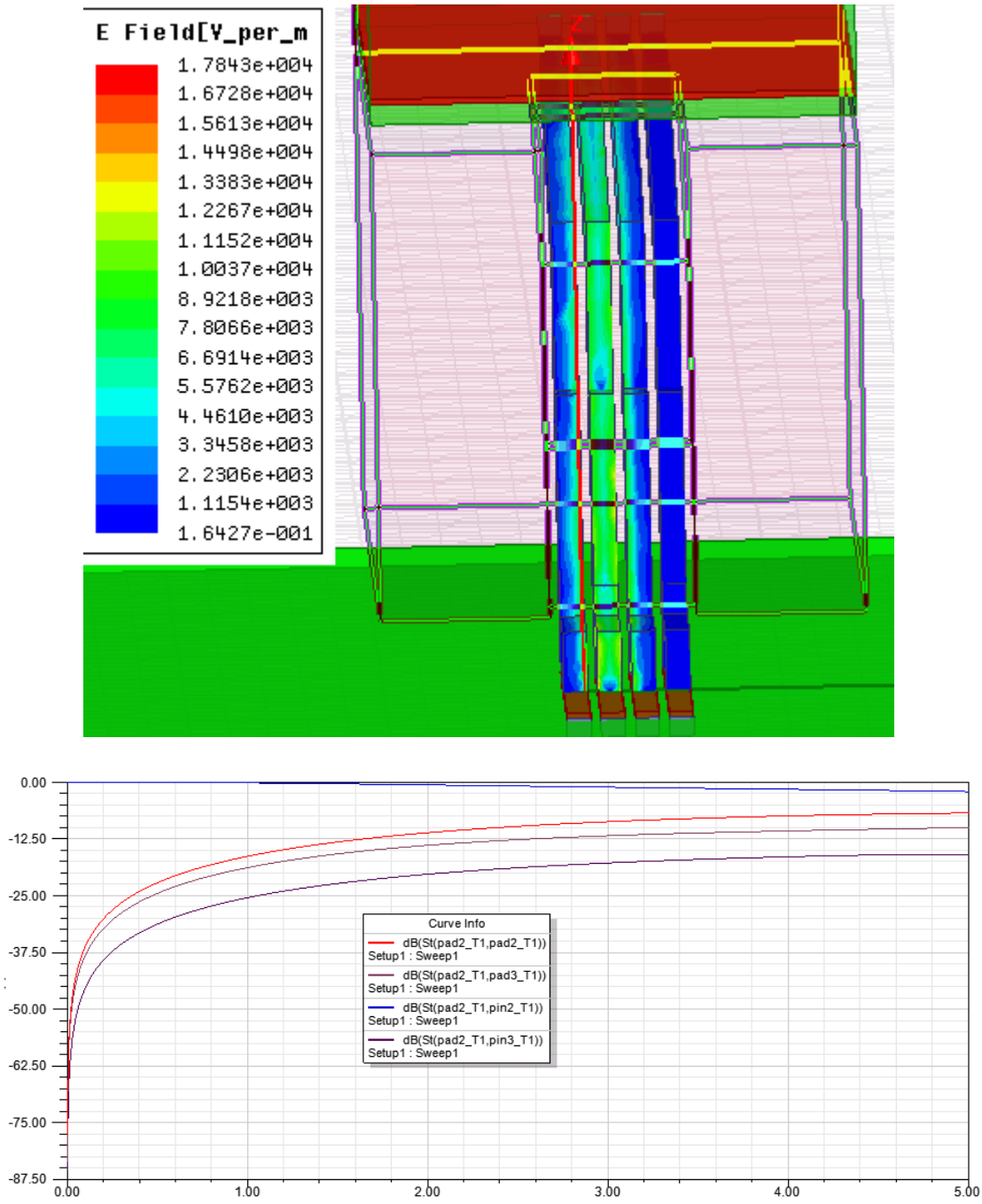

Figure 2.12: Clock strip analysis for package connector and S-Parameters. Refer Appendix C

As demonstrated in [33], system paths are generally more resistance to environmental changes when properly configured. On-chip interconnects on the other hand are more susceptible to process variations, thermal, leakage current and other environmental factors [34][35].

2.7 Conclusion

From the research on the existing bus architectures and the development of DTP-XBUS-2 for seamless IP core integration at the system level, the information to design the DTP-XBUS-2 core could be generalized as follows:

a) The DTP-XBUS-2 core bypasses the ISO-OSI protocol stacks above layer 3 for direct communication with in-system IP cores.
b) The DTP-XBUS-2 core uses dual clock rate to optimize on-chip and off-chip data transfer rate.
c) To evaluate the DTP-XBUS-2 core performance, a CPU intensive thread needs to be executed and benchmark against actual single-core and dual-core SoC systems.
d) Load scheduling methodologies and techniques could be directly implemented to optimize resource allocation for each IP cores through the DTP-XBUS-2 core.
e) For high bit rate transfer the DTP-XBUS-2 core should have reasonable well performance with significant reduction in bit error rate and high signal to noise ration.

Chapter 3: System Environment and Organization

3.1 System Architecture – The Big Picture

The exponential growth of computing power and ownership has made computer one of the most important forces shaping business and society. For many years each new generation of processor produces more heat than the one before as the number of cores multiplies with significant increase in performance. Heat causes devices to run unreliably at high speeds or high workloads. Throughput Computing is a technique that takes advantage of the thread-level parallelism that is present in most commercial workloads. Unlike desktop workload, which often has a small number of threads running concurrently, most commercial workload achieves scalability by employing large pools of concurrent threads.

3.2 Instruction-Level Parallelism (ILP), Thread-Level Parallelism (TLP) and System-Level Parallelism (SLP)

Historically, microprocessor has been designed to target desktop workload, and as a result focused on running a single thread as efficiently as possible. Single thread performance is achieved in these processors by a combination of extremely deep pipelines (over 20 stages in Pentium 4) and by executing multiple instructions in parallel (referred to as instruction-level parallelism or ILP) [10]. The tenet behind throughput computing is that the exploitation of ILP through deep pipelining has reached the point of diminishing returns, and as a result current microprocessors do not utilize their underlying hardware very efficiently. For a majority of commercial workloads, the processor will be idle most of the time waiting on memory, and even when it is executing it will often be able to only utilize a small fraction of its wide execution width. So rather than building a large and complex ILP processor that sits idle most of the time, a number of small, single-issue processors that employ multithreading are built in the same chip area. Combining multiple processors on a single chip with multiple strands per processor allows very high performance for highly threaded commercial applications (hence Thread-Level parallelism, TLP) [21]. This thesis explores a new mechanism for multi-processor-multi-chip operation, or multi-system processing by integrating IP cores externally. Operation requests generated by IP core are encapsulated as

threads which are then distributed by DTP-XBUS-2 in order to utilize resources and features available externally such as graphics memories or graphics processing units. This approach is called System-level parallelism (SLP), and the difference between SLP, TLP and ILP is shown in the figures below (Figure 3.1, Figure 3.2 and Figure 3.3).

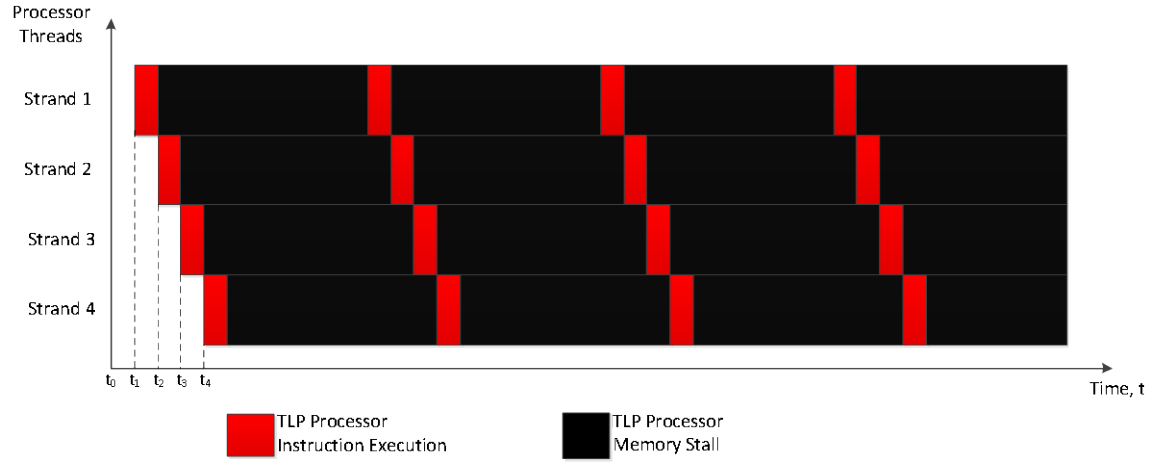

Figure 3.1: Thread-Level Parallelism (TLP). The figure shows the starts of Strand 1, Strand 2, Strand 3 and Strand 4 arbitrarily and sequentially at t_1, t_2, t_3 and t_4 respectively after time t_0 on a single TLP processing core.

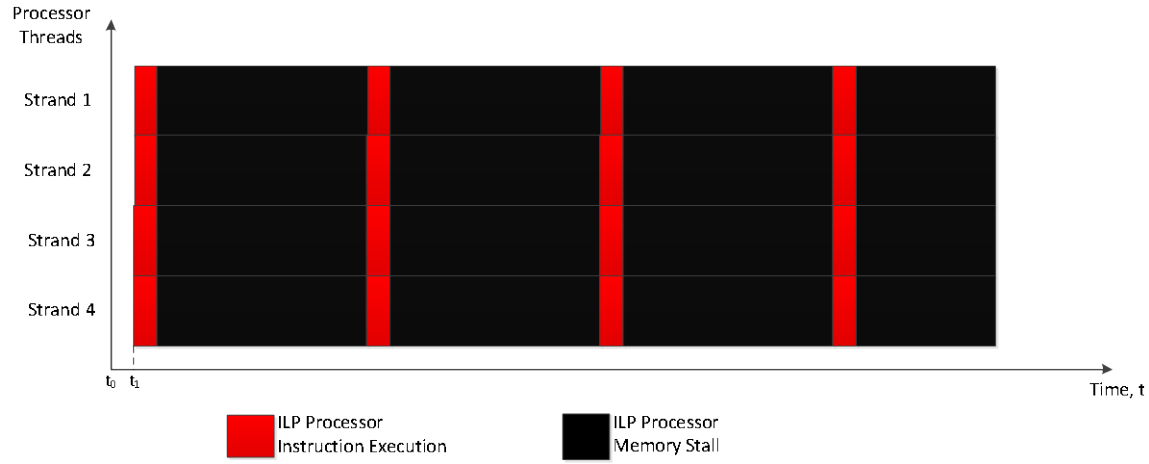

Figure 3.2: Instruction-Level Parallelism (ILP). The figure shows the starts of Strand 1, Strand 2, Strand 3 and Strand 4 arbitrarily and synchronously at t_1 after time t_0 on a single ILP processing core.

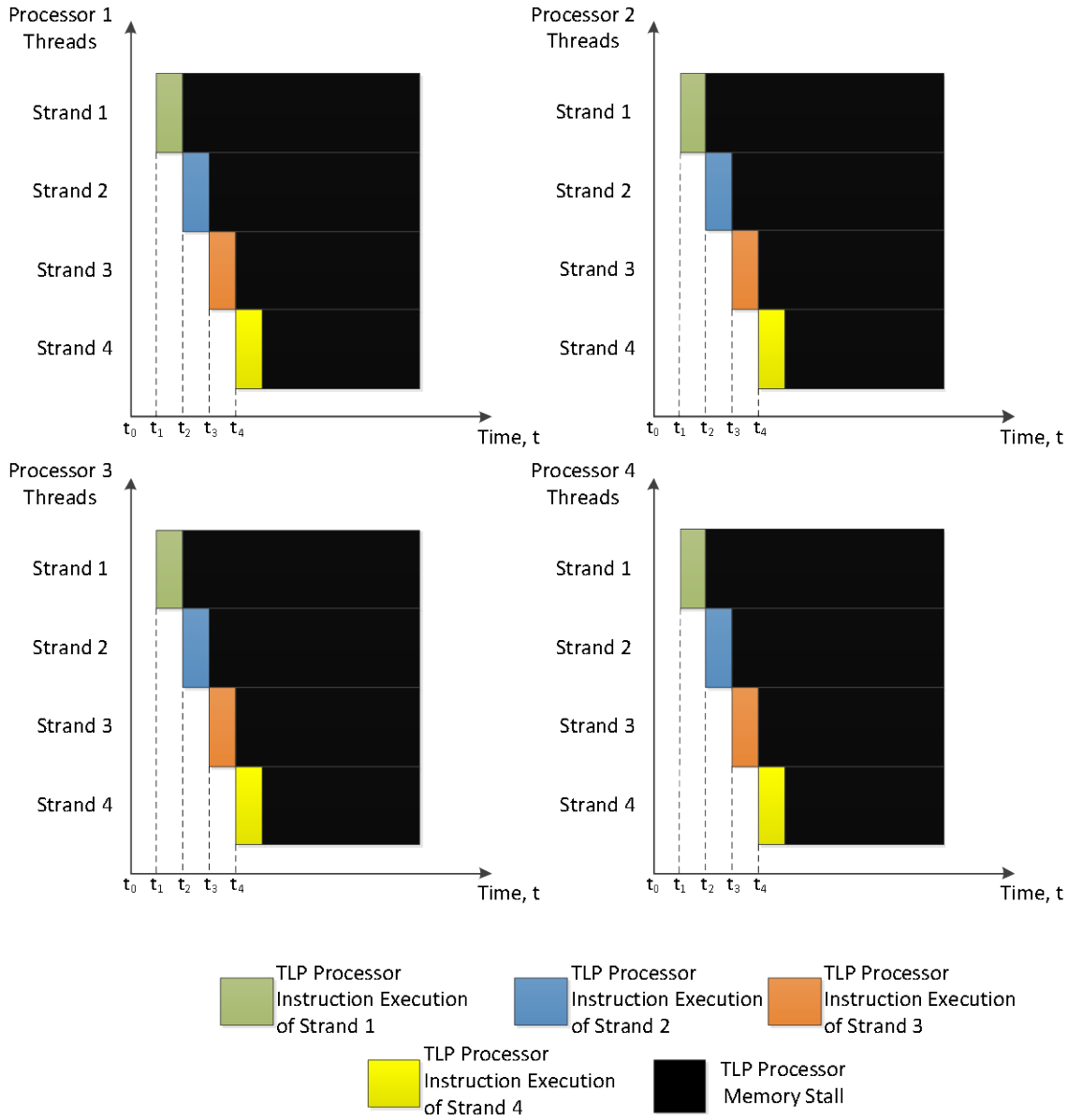

Figure 3.3: System- Level Parallelism (SLP). The figure shows the starts of Strand 1, Strand 2, Strand 3 and Strand 4 arbitrarily and synchronously at t_1, t_2, t_3 and t_4 respectively after time t_0 on multiple TLP processing cores.

The memory stall time of one strand can often be overlapped with the execution of other strands on the same processor, and multiple processors run their strands in parallel and hence completely overlap memory latency with the execution of other strands. Instruction-Level parallelism on the other hand attempts to reduce execution lead time through deep pipelining. System-Level parallelism enhances TLP by synchronizing threads at the system level. This allows TLP processors to emulate ILP at the system level. With processors capable of multiple GHz clocking, the performance bottleneck has shifted to the memory and

I/O subsystems, and TLP is more tolerance against large I/O and on-chip memory latency compared to ILP micro-architecture.

The following section elaborates in more detail about the backbone behind SLP and its implementation as DTP-XBUS-2 core.

3.3 DTP-XBUS-2 System Overview

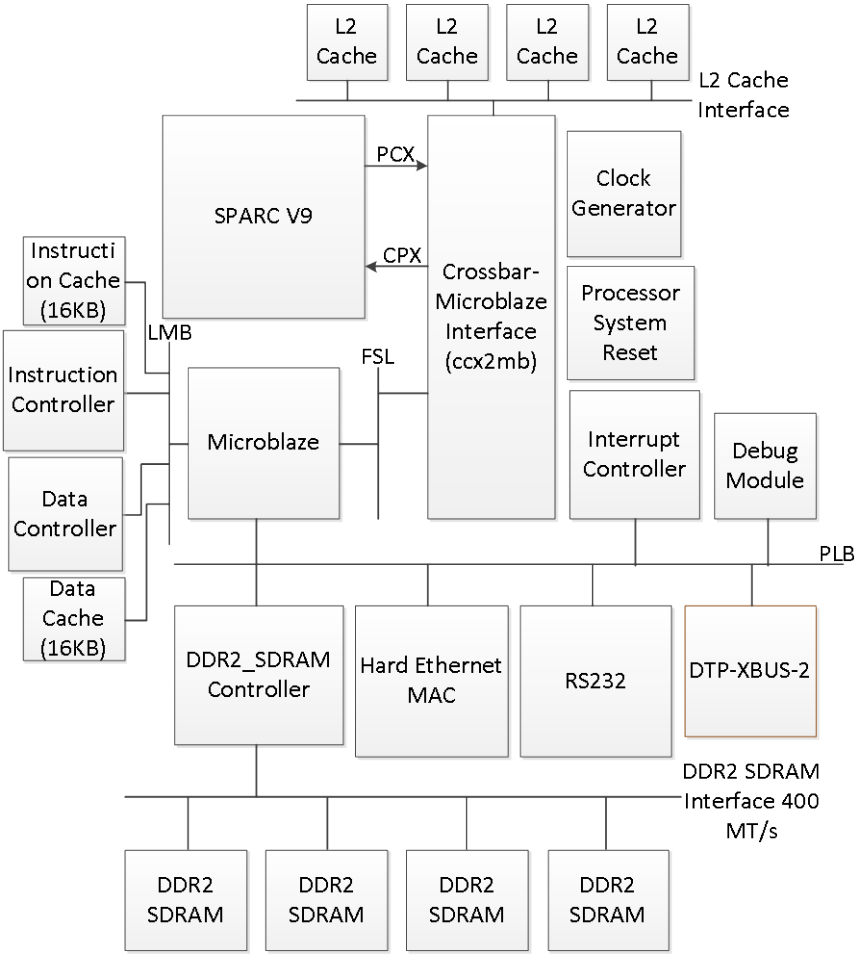

Figure 3.4: DTP-XBUS-2 System Overview. PCX and CPX are the Processor-to-Cache-Crossbar and Cache-Crossbar-to-Processor interfaces respectively. Fast Simplex Link (FSL) is used as a uni-directional point-to-point high-speed communication. Local Memory Bus (LMB) is used as the interface to on-chip Block RAM (BRAM). Processor Local Bus (PLB) is used as the interface that interconnects multiple IP cores.

The figure (Figure 3.4) above shows the complete DTP-XBUS-2 system, implemented with a single chip multi-threaded (CMT) System-on-Chip (SOC) processor that contains a single SPARC V9 physical processor core. The SPARC V9 physical processor core has full hardware support for eight strands, two integer execution pipelines, one floating-point execution pipeline, and one memory pipeline. The floating-point and memory pipelines are

shared by all eight strands. The eight strands are hard-partitioned into two groups of four, and the four strands within a group share a single integer pipeline. Hence, at any given time at most two strands will be active in the physical core, and those two strands will be issuing either a pair of integer pipeline operations, an integer operation and a floating-point operation, an integer operation and a memory operation, or a floating-point operation and a memory operation. Strands are switched on a cycle-by-cycle basis between the available strands within the hard-partitioned group of four using a least recently issued priority scheme. When a strand encounters a long latency event, such as a cache miss, it is marked unavailable and instructions will not be issued from that strand until that event is resolved. Execution of the remaining available strands will continue while the long-latency event is being resolved. The SPARC V9 core has a 16KB of 8-way associative instruction cache (32-byte lines), 16 KB of 8-way associative data cache (32-byte lines), 64-entry associative instruction Translation Look-aside Buffers (TLB), and 128-entry associative data TLB that are shared by the eight strands. The TLB provides first level translation for instruction and data accesses. If any single entry matches, the TLB generates a Physical Address (PA) by concatenating the Physical Page Number (PPN) stored in the TLB with the lower portion of the virtual address. If no entries match, then the TLB signals a data or instruction miss.

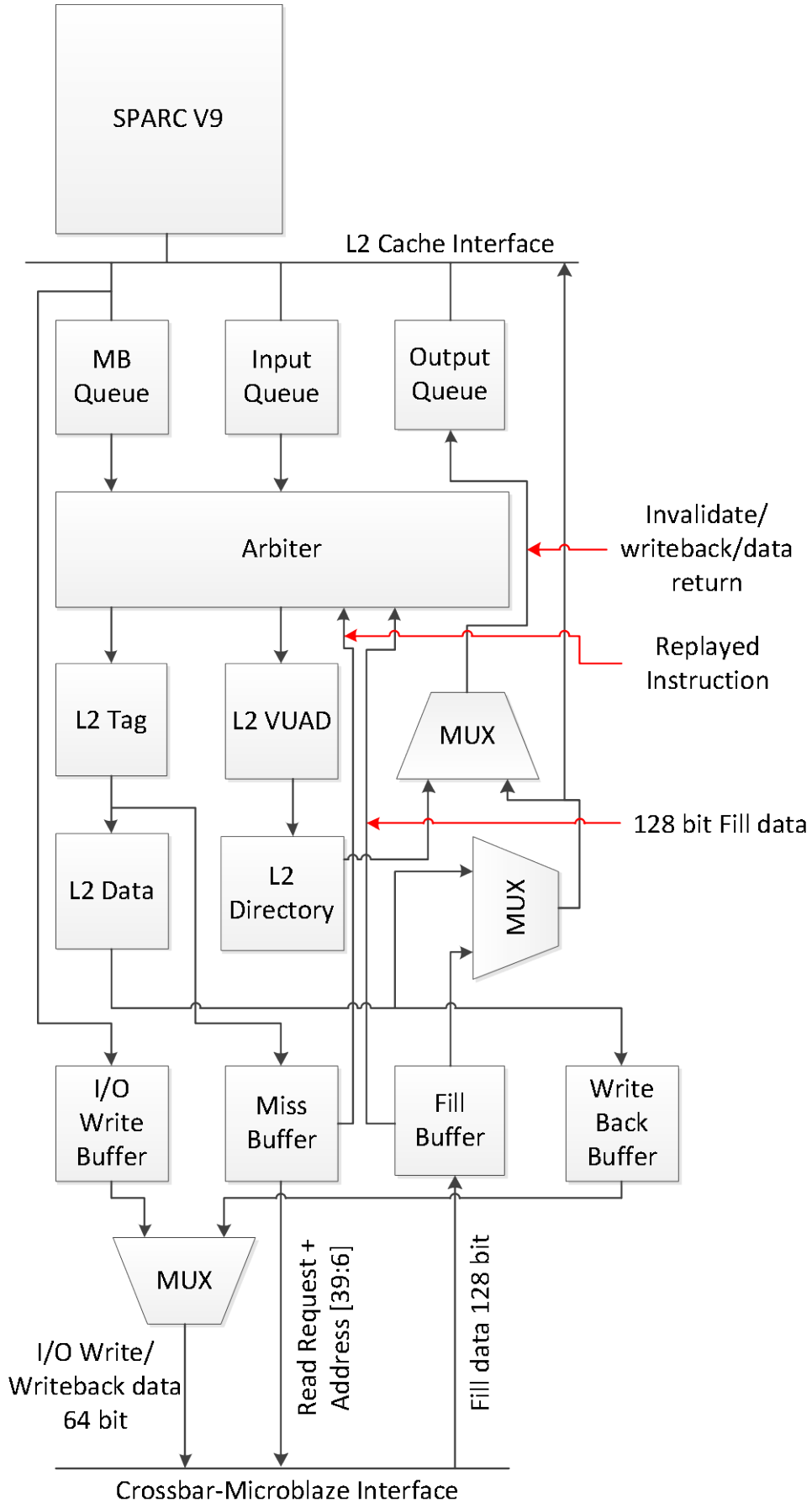

Figure 3.5: Cache Organization

The SPARC V9 physical core is connected through a crossbar to an on-chip unified 4 MB of 16-way associative L2 cache (64-byte lines). The L2 cache is banked eight ways to provide sufficient bandwidth for Data Transfer Protocol (DTP) operation that functions as a fabric of communication through the Extensible-BUS (XBUS) on a multi-chip-multi-processor platform. The L2 cache connects to four on-chip DRAM controllers, which directly interface to a pair of fully buffered DIMM (FBD) channels. In addition, an on-chip PCI-EX controller, two 1Gbit/10Gbit Ethernet MACs, and several on-chip I/O-mapped control registers are accessible to the SPARC V9 physical core. The XBUS-2 functions as an intermediate channel of communication that glues the Network Interface Unit (NIU) and the processor core through the Cache Crossbar (CCX) and System Interface Unit (SIU). XBUS-2 emulates SLP and synchronizes IP cores execution at the system level. Traffic from the PCI-EX port coherently interacts with the L2 cache. The cache organization is shown in Figure 3.5

The L2 Cache sub-blocks are described as below:

a) Input Queue. A 16 entry FIFO which queues packets arriving on the L2 Cache interface when they cannot be immediately accepted into the L2 pipe. Each entry in the queue is 130 bits wide.
b) MicroBlaze Queue (MB Queue). MB Queue Accepts packets from MicroBlaze and issues them to the pipe after arbitrating against other requests.
c) Arbiter. The arbiter manages access to the L2 pipeline from the various sources which request access.
d) L2 Tag. The tag holds the L2 tag array and associated control logic.
e) L2 VUAD contains the Valid, Used, Allocated, and Dirty bits for the tags in L2 array structure.
f) L2 data contains 512 KB of L2 Data storage and associated control logic.
g) L2 Directory maintains a copy of the L1 tag for coherency management and also ensures that the same line does not reside in both the Instruction Cache (icache) and Data Cache (dcache) in the processing core.
h) Miss Buffer. The Miss Buffer (MB) has 32 entries and stores instructions which cannot be processed as a simple cache hit. This includes true L2 cache misses, instructions that have the same cache line address as a previous miss or an entry in the Writeback buffer, instructions requiring multiple passes through the L2 pipeline, unallocated L2 misses, and accesses causing tag Error Control Code (ECC) error.
i) Fill Buffer is an eight entry buffer used to temporarily store data arriving from DRAM on an L2 miss request. Data arrives from DRAM in four 16 Byte quad-words starting with the critical quad-word.

j) Write Back Buffer. This buffer is an eight entry buffer used to store dirty evicted data from the L2 on a miss. Evicted lines are then streamed out to DRAM.

k) Input-Output (I/O) Write Buffer is a four entry buffer which stores incoming data from the PCI Express (PCI-EX) interface in the case of a 64 Byte write operation. As the PCI-EX interface bus width is only 32 bits wide, the data must be collected over 16 cycles before writing to DRAM.

As discussed earlier, implementing a digital interface is a practical solution in establishing high-speed communication between complex TLP processing units. A flexible and configurable control-architecture is required to control the XBUS transceiver's chains (TX/RX), and to transfer (or share) communication between the transceivers in each TLP processing unit. This control-architecture also configures the DTP transceivers to activate a particular standard. The XBUS-2 operates in a control-architecture being developed to incorporate dual-channel transceivers. This control-architecture comprises specialized adapters, a bus and distribution system, a multi-core debug system and the CPU Subsystem. They are also known as the Processor Local Bus (PLB) (Section 3.4)

3.4 Processor Local Interconnect Bus Standard and Implementation

Processor Local Bus (PLB) is a high performance I/O bus used to interconnect peripheral devices in applications such as computing and communication platforms (Figure 3.6). The Processor Local Bus is an all-encompassing I/O device-interconnect bus that has applications in the mobile, desktop, workstation, server, embedded computing and communication platforms. In order to improve bus performance, reduce overall system cost and take advantage of new developments in computer design, the local bus implements a serial, point-to-point type interconnect for communication between two devices. Multiple devices or cores are interconnected via the use of switches which means one can practically connect a large number of devices together in a system [22]. A point-to-point interconnect implies limited electrical load on the link allowing transmission and reception. A serial interconnect between two devices results in fewer interfaces per device which reduces overall design complexity. The processor local bus performance is also highly scalable. This is achieved by implementing scalable numbers of pins and signal lanes per interconnect based on communication performance requirements for that interconnect.

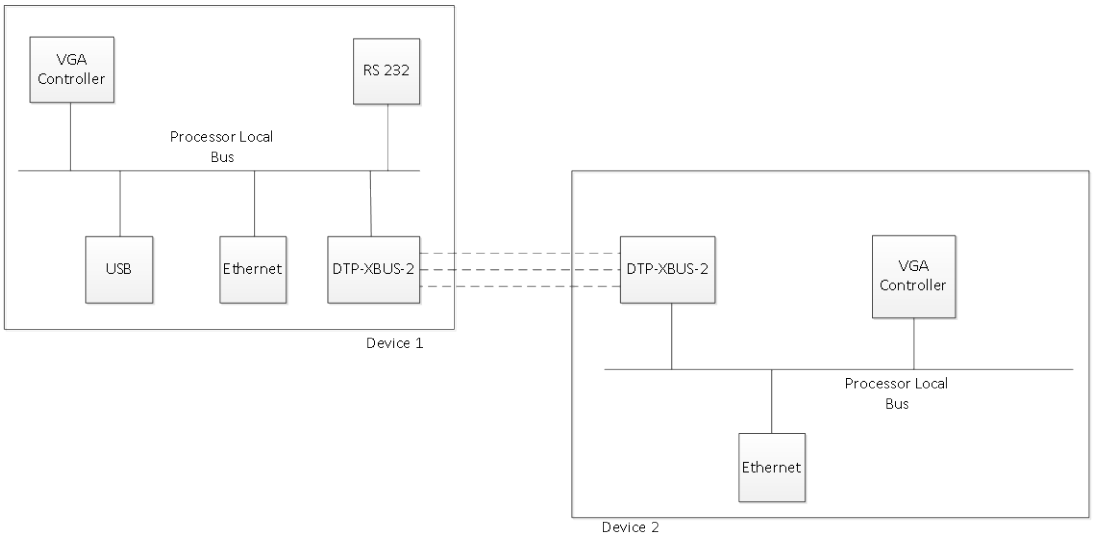

Figure 3.6: Local Bus Interconnect Implementation with XBUS-2

The PLB implements a switch-based technology (or package-switched) to interconnect a large number of devices. Communication over the serial interconnect is accomplished using a package-based communication protocol. Quality of Service (QoS) features provides differentiated transmission performance for different applications [24]. Hot Plug/Hot Swap support enabled "always-on" systems. Advanced power management features allow one to design for low power mobile applications. Reliable, Available, and Serviceable (RAS) error handling features make PLB-Interconnect Bus suitable for robust high-end server applications. Hot plug, power management, error handling and interrupt signalling are accomplished in-band using packet based messaging rather than side-band signals. This keeps the device interface count low and reduces system design complexity.

In summary, the Processor Local Bus (PLB) is a high-performance 64-bit address bus and a 128-bit data bus. The PLB provides a standard interface between the processor cores and integrated bus controllers. This allows the development of a library of processor cores and bus controllers for use or reuse in core, application-specific integrated circuits (ASICs) and system-on-chip (SoC) designs. The PLB supports read and write data transfers between master devices and slave devices that are equipped with a local bus interface and are connected through dedicated signals. Each master device (Master PLB or MPLB) is attached to the bus through separate address buses, read data buses, write data buses, and transfer qualifier signals. Slave devices are attached to the local bus through shared, but decoupled, address buses, read data buses, write data buses, and transfer control and status signals for each data bus. DTP-XBUS-2 uses a hybrid of master and slave PLB interfaces. The local bus grants access through a central arbitration mechanism that allows master devices to compete

for bus ownership. This arbitration mechanism is flexible enough to provide for the implementation of various priority schemes. Also, an arbitration locking mechanism is used to support master-driven atomic operations. The local bus is a fully asynchronous bus. A single clock source provides timing for all dedicated local bus channels. All masters and slaves that are attached to the local bus share this clock source (Figure 3.7).

The processor local bus is the high performance bus that is also used to access memory through the bus interface units. The local bus implementation consists of a serial-bus core in which all master and slave devices are attached. The logic within the serial-bus core consists of a central bus arbiter and the necessary bus control and gating logic.

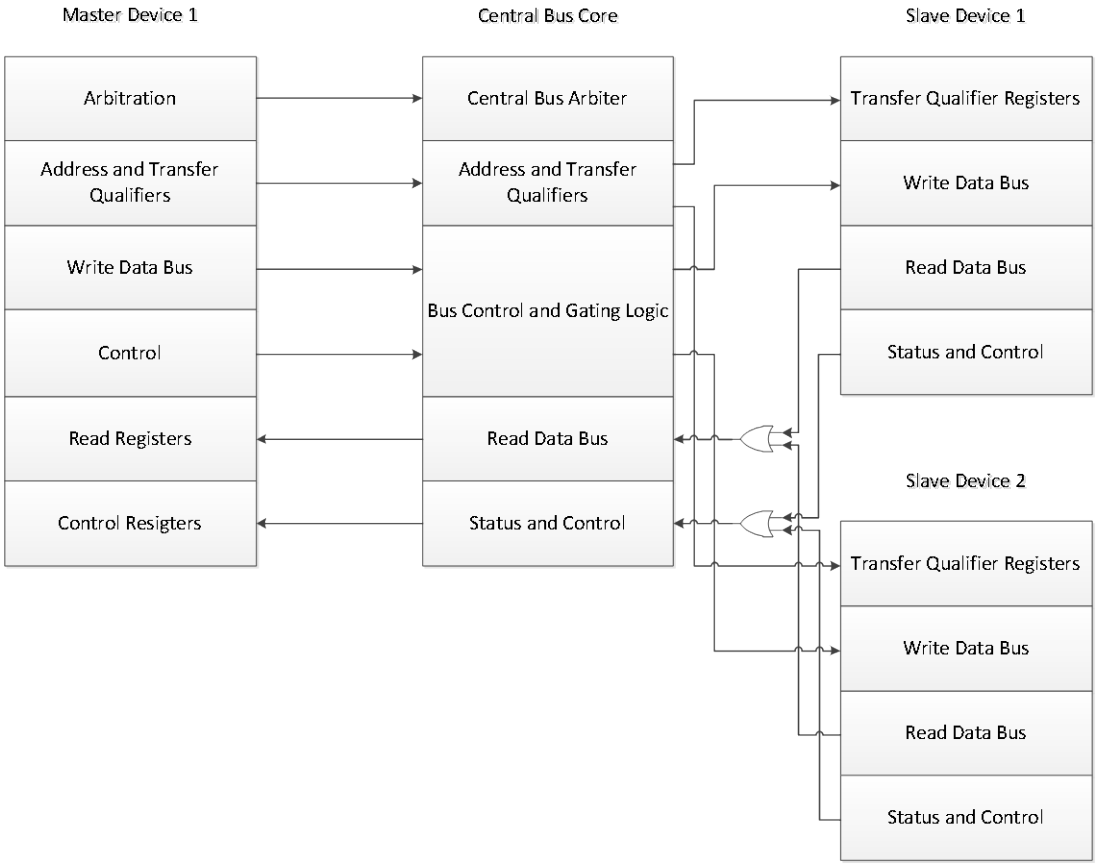

Figure 3.7: Central Bus core

The local bus architecture supports up to sixteen master devices on-chip and unlimited number of slave devices. The XBUS-2 further extends the maximum number of attached master devices by a factor of two through process sharing (thread splitting or memory object distribution) at the system level. However, the number of masters and slaves other than the

XBUS-2 that are attached to a serial-bus core in a particular system directly affects the performance of the bus core in that system.

3.4.1 PLB Transfer Protocol

The address cycle has three phases: request, transfer, and address acknowledgement (Figure 3.8). A local bus transaction begins when a master drives its address and transfer qualifier signals and requests ownership of the bus during the request phase of the address cycle. After the local bus arbiter has granted bus ownership, the address and transfer qualifiers for the master are presented to the slave devices during the transfer phase. During normal operation, the address cycle is terminated by a slave latching the address and transfer qualifiers for the master during the address acknowledgement phase.

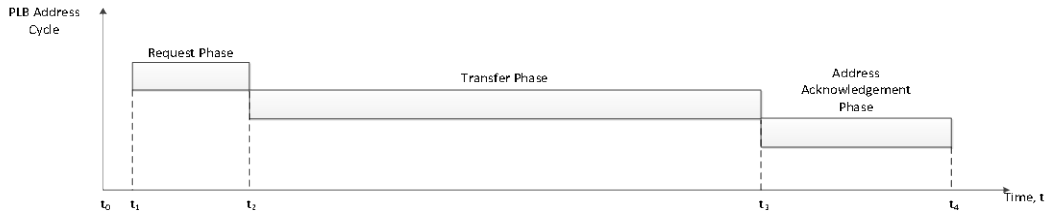

Figure 3.8: The initiation of Address Cycle arbitrarily at time t_1 after t_0. For this cycle, the Request Phase, Transfer Phase and Address Acknowledgment Phase take $t_2 - t_1$, $t_3 - t_2$, and $t_4 - t_3$, time intervals respectively.

Each data beat in the data cycle has two phases: transfer and data acknowledgment (Figure 3.9). During the transfer phase, the master drives the write data bus for a write transfer or samples the read data bus for a read transfer. Data acknowledgement signals are required during the data acknowledgement phase for each data beat in a data cycle.

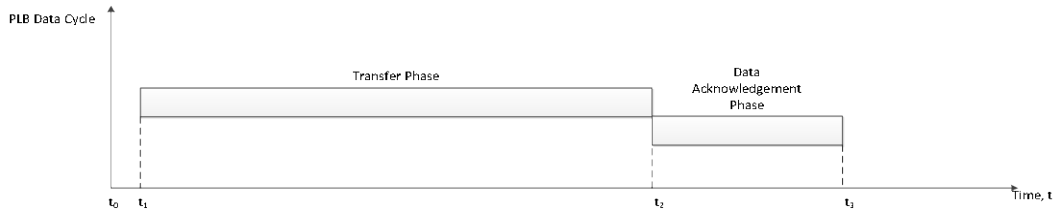

Figure 3.9: The initiation of Data Cycle arbitrarily at t_1 after t_0. For this cycle, the Transfer Phase and Data Acknowledgment Phase take $t_2 - t_1$ and $t_3 - t_2$ time intervals respectively.

PLB address buses, read data buses, and write data buses are decoupled from one another allowing for address cycles to be overlapped with read or write data cycles, and for read cycles to be overlapped with write data cycles. The processor local bus split-bus transaction capability allows the data and address buses to have different masters at the same time. The

pipelining capability allows a new bus transfer to begin before the current transfer has been completed. Address pipelining reduces the overall bus latency on the local bus by allowing latency that is associated with a new transfer request to be overlapped with an ongoing data transfer in the same direction (or same destination).

3.4.2 PLB Interface

The PLB Interface consists of the following categories:

1) System signals.
2) Arbitration signals.
3) Status signals.
4) Transfer qualifier signals.
5) Read data bus signals.
6) Write data bus signals.

3.4.3 System signals

The system clock signal provides the timing for the local bus and acts as input to all master devices, slave devices and the local bus arbiter. All master output signals, slave output signals, and arbiter output signals are asserted or negated relative to the rising edge of the system signals. All master input signals, slave input signals, and arbiter input signals are sampled relative to this edge. The master and slave, attached to the local bus are expected to operate at the frequency of the bus. Thus, any matching speed that is required because of I/O constraints is handled in the local bus interfaces of master and slaves (cycle conversion).

The system signals also contain the power-on reset signal for the local bus arbiter. This signal is used to switch the bus to an idle or quiescent state that usually and has the following characteristic:

1) No read or write bus requests are pending.
2) The bus is not locked.
3) The bus is not granted.
4) The read and write data buses are not being used.

This signal is usually asserted relative to the rising edge of the system clock signal. The duration of the assertion when forcing the bus to idle state in a system depends on the implementation of the arbiter, master and slave devices.

3.4.4 Arbitration signals

During the request phase, the arbitration signals are used to compete for the ownership of the bus. The Master Request is as shown in Figure 3.10 and Figure 3.11. When the arbiter has granted the bus to a master, the master's address and transfer qualifier signals are presented to the addressed slaves during the transfer phase. The transfer phase is marked by the assertion of the arbitration signals. The maximum length of the transfer phase is controlled by the address cycle timeout mechanism.

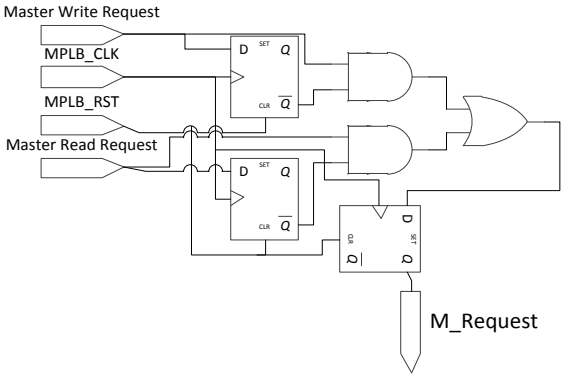

Figure 3.10: Master Request Schematic

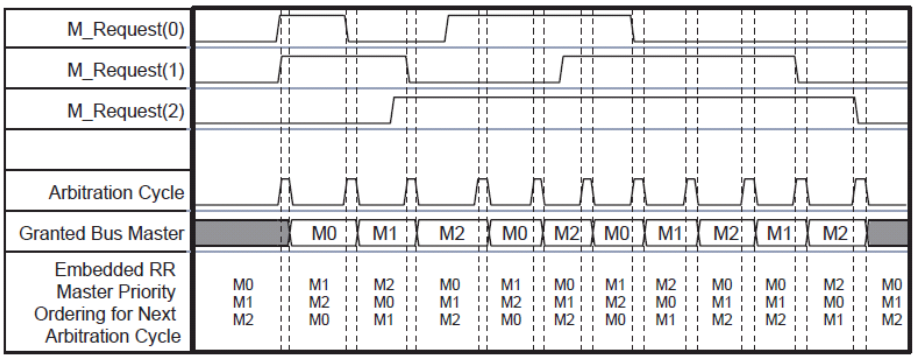

Figure 3.11: M_Request of three Master devices.

During termination phase, the address cycle in completed by the slave through assertion of acknowledgement or completely aborted by master through timing out. It is possible for all three phases (request, transfer and termination) of the address to occur in a single clock cycle in single cycle arbitration.

3.4.5 Status signals

Status signals are driven by the arbiter and reflect the ownership status of master. Master and slave devices use these signals to help resolve arbitration on the bus or DTP-XBUS-2. The arbiter modifies the status signals as indication that a master has a read request that is pending on the bus or DTP-XBUS-2 or that a secondary read transfer has been acknowledge

and is pending. The assertion is a combined logic OR of all the master request inputs for reads, secondary read bus status and interrupt requests by DTP-XBUS-2. The status signals play similar roles for a write request. The status signals also hold the slave identification of the master of current transfer or external transaction through the DTP-XBUS-2.

3.4.6 Transfer qualifier signals

The master address and transfer qualifier signals are generated when a request is asserted. The signals continue to be driven by the master, unchanged, until the clock cycle following the assertion of acknowledgement, re-arbitrate, or abortion. On the slave interface, the transfer qualifier signals are latched at the end of the address acknowledgement cycle.

3.4.7 Read data bus signals

The read data cycle is divided into two phases: transfer and data acknowledgement. During the transfer phase, the slave places data to be read on the read data bus. The master then waits for the slave to indicate that the data on the read data bus is valid during the data acknowledgement phase. A single beat transfer has one transfer phase and one data acknowledgement phase associated with it. A line or burst transfer has multiple number of transfer and data acknowledgement phases. A master begins a read transfer by asserting its request signal and by placing high value on the read-write channel. When the bus is granted to the master, the arbiter gates or shift the data onto the master data registers.

3.4.8 Write data bus signals

The write data cycle has two phases: transfer and data acknowledgement. During the transfer phase, the master places data to be written on the write data bus. The master then waits for a slave to indicate the completion of the write data transfer during the data acknowledgement phase. The write data cycle is very much similar to the read data cycle in that a write request is generated and data is shifted from the master onto slave the bus is granted to the respective master.

The Processor Local Bus provides a means of interconnecting subsystem peripherals including the memory core. The following section will discuss about the memory core implementation.

3.5 The Data Transfer Protocol (DTP) Memory Architecture

During the microprocessor evolution, memories became an integral part of microprocessor design. The first integrated microprocessors contained only register files as storage for

temporary data, while memory system was entirely located off chip. Current microprocessor chips include up to three levels of cache memory (L1, L2 and L3). Furthermore, the total on-chip memory capacity increased from a few kilobytes to several megabytes. Consequently there is a strong demand for dense, fast and energy-efficient memories. In addition, there are some trade-offs between density, speed, and energy dissipation that can be made, depending on memory design specifications or memory architecture. For instance, the primary concern of multi-port register files is their delay time and clock frequency [23].

In many systems, the peripheral devices and memory share the same busses with the processor. Since the bus is the only path in and out of the system, conflicts will arise when peripheral devices have data for the processor, but the processor is busy executing program code. Hence, the DTMP (Data Transfer Memory Protocol) efficiently handles and transfers memory control from the processor to the peripheral devices and write all of its data in a single burst of activity. The DTMP places the processor memory interface in a tri-state condition while the transfers take place. This allows other devices to take over control of the busses and implement a data transfer to or from memory while the processor idles, or processes from a separately cached memory (Figure 3.12). This performance gain will be demonstrated in chapter 6.

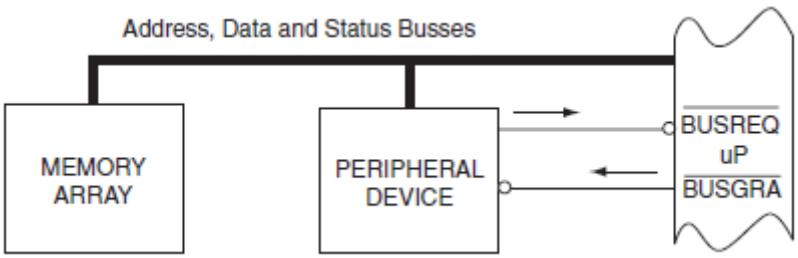

Figure 3.12: Schematic representation of DTMP transfer

The figure above shows the simplified DTMP process. In the simplest form, there is a handshake process that takes place between and the processor and the peripheral device. The process can be described as follows:

- The peripheral device requests control from the bus from the processor by asserting the BUS REQUEST (BUSREQ) signal input on the processor through the cache-crossbar (CCX).
- When processor completes present instruction cycle, and no higher level interrupts are pending, it sends out a BUS GRANT (BUSGRA), giving the requesting device permission to begin its own memory cycles.

- Processor then idles, or continues to process data internally in cache, until BUSREQ signal is negated.

3.5.1 Memory Organization

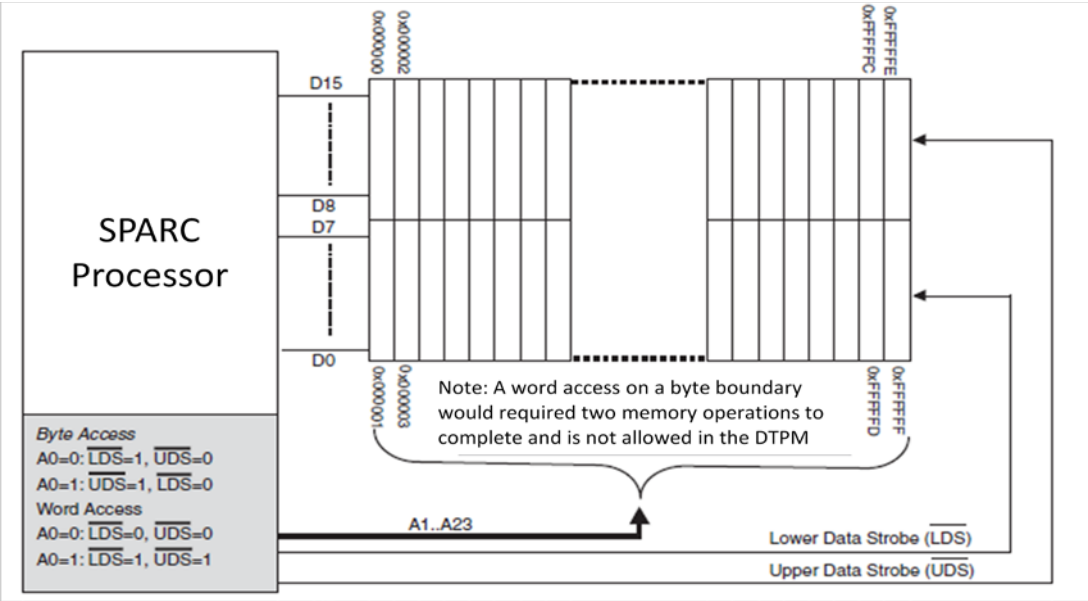

Figure 3.13: Memory addressing modes with DTMP

When the processor is performing a byte access to memory, then either Lower Data Strobe (LDS) or Upper Data Strobe (UDS) is asserted to represent the part of memory which where the word is being accessed (refer figure 3.13). If the byte at the even address is being accessed, then UDS is asserted and LDS stays HIGH. If the odd byte is being accessed, then LDS is asserted and UDS remains in the HIGH, or OFF state. For a word access, both UDS and LDS are asserted.

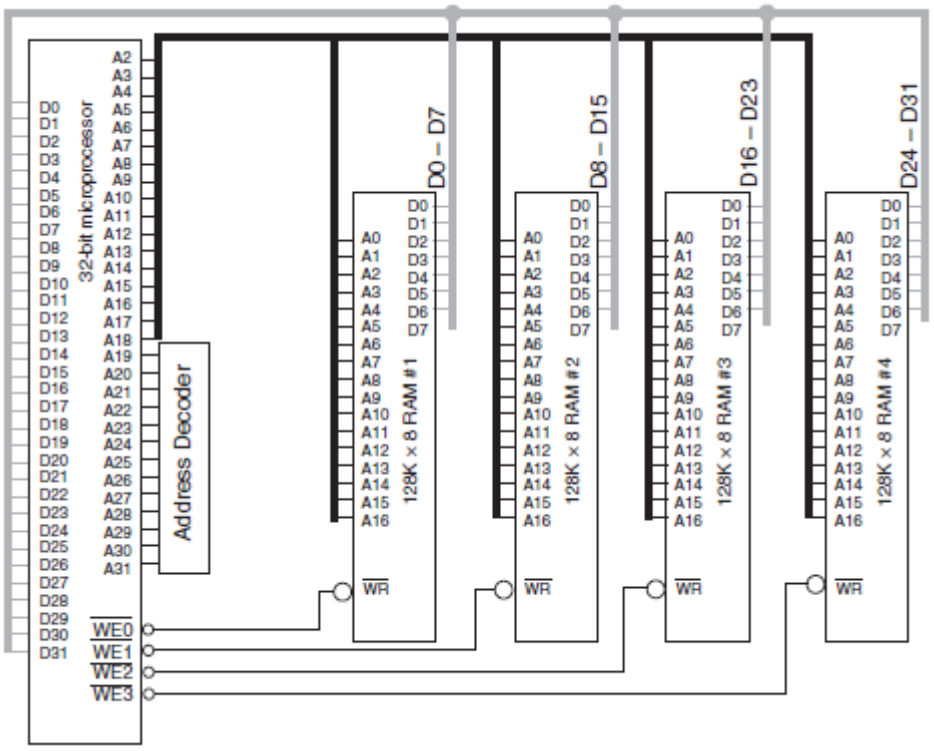

Figure 3.14: Memory Organization for DTMP

The figure above (Figure 3.14) shows the DTMP and memory system interface. The READ signal from the processor and the CHIP SELECT signals have been omitted for clarity. The processor has a 32-bit data bus and a 32-bit address bus. The memory chips represent one page of DDR RAM in the address space of the processor. The exact page of memory would be determined by the design of the Address Decoder logic block. The DDR RAM chips each have a capacity of 1 MB and are organized as 128KB by 8. The address bus from the processor contains 30 address lines, which means it is capable of address 230 long words (32-bit wide). The additional addressing bits needed to address the full address space of 232 bytes are implicitly controlled by the processor internally and explicitly controlled through the 4 WRITE ENABLE signals labelled WE0 through WE3. Address lines A2 through A18 from the processor are connected to address inputs A0 through A16 of the DDR RAM chips, with A2 from the processor being connected to A0 on each of the 4 internal cache memories, and so on. The upper address bits from the processor, A19 through A31 are used for the page selection process. These signals are routed to the address decoding logic where the appropriate CHIP SELECT signals are generated.

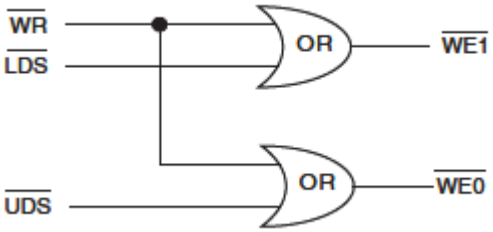

Figure 3.15: Byte write control circuit

As the signals are asserted low, this forms an equivalent of negative logic AND function (hence by De Morgan's Theorems) (Figure 3.15):

(A*B)' = A' + B'
(A+B)' = A'*B'

3.6 The Two-Dimensional Extensible Bus (XBUS-2) Architecture

The previous sections discussed the necessary system environment and components for DTP communication protocol implementation. This section will elaborate in detail about the XBUS-2 architecture or the hardware implementation.

Communication can be defined as the imparting or exchange of information. Modern living demands that we access to a reliable, economical and efficient means of communication. Telephony is an example of point-to-point communication and normally involves a two way flow of information. Another type of communication, which traditionally involves only one-way information flow, is broadcast standard electronic equipment. In these systems information is transmitted from one location but is received at many locations using many independent data terminal devices. An important objective in the design of the XBUS-2 communication system is to minimise equipment cost, complexity and power consumption whilst also minimising the bandwidth occupied by the signal and/or transmission time. Efficient use of bandwidth and transmission time ensures that as many processing units as possible can be accommodated within the constraints of these limited, and therefore valuable, resources [4].

Buses (Figure 3.16) are one of the most widely used means of communication between components in a SoC. The bus can be physically implemented as a single wire which makes up a parallel bus. This parallel bus is the typical implementation choice for a bus in almost all widely used on-chip bus-based communication architectures. Any data transmitted by a

component moves from its output pins to the bus wires and is then received at the input pins of the destination component. The destination component typically sends an acknowledgement back to the transmitting component to indicate if the data was received. A *bus protocol* is used to explicitly define a communication transaction through its temporal (e.g., duration and sequence of messages exchanged) and spatial (e.g., message size) characteristics. The bus protocol also determines which component may access the shared bus if multiple requests to send (or receive) data appear on the bus at the same time. Bus-based communication architectures usually consist of one or more shared buses as well as logic components that implement the details of a particular bus protocol.

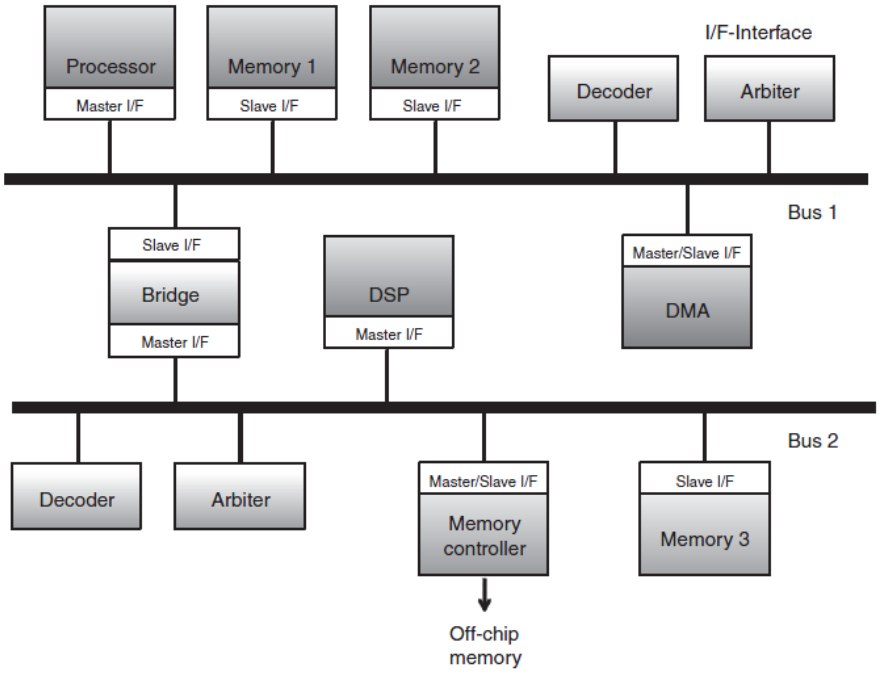

Figure 3.16: Example of a Bus-based Communication Architecture

3.6.1 XBUS-2 Communication Architecture

Bus-based communication architectures are defined by various architectural and physical characteristics (ISO-OSI physical layer) that can have many different implementations. These implementation choices have trade-offs that can significantly affect the power, performance, and occupied area of the communication architecture. Traditionally, shared buses have been implemented using tri-state buffers that drive bi-directional lines (Figure 3.17). Tri-state implementations of buses are commonly used in off-chip/backplane buses (which is the case of XBUS-2). The advantage of tri-state bidirectional buses is that they take up fewer wires and have a smaller area footprint [25].

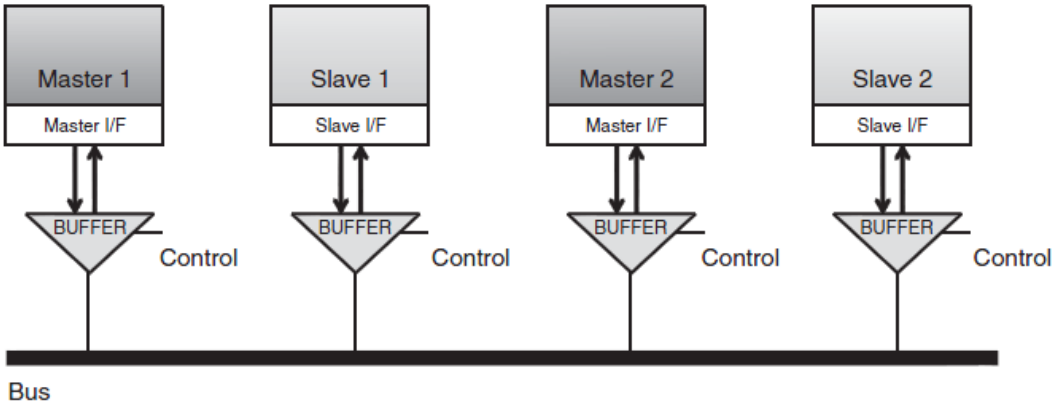

Figure 3.17: Tristate Buffer based Bidirectional Signals

XBUS-2 (Figure 3.18) is responsible for managing and directing all command and data flows from or to external chip components and the internal chip components such as the system interface unit, PCI-Express unit, and non-Cacheable unit. The XBUS-2 manages the transaction layer packet to and from both external and internal components, and maintains the ordering by queue identity. Whenever an external transaction issues complete TLP transactions to the XBUS-2, the XBUS-2 segments the TLP packet into multiple cacheline oriented system commands and issue them to the system-interface-unit. The XBUS-2 then queues the response cachelines from the unit and reassembles the multiple cachelines into one TLP packet with maximal payload size for appropriate responses to other external transactions. This form of encapsulation resolves a virtual packet addressing into an on-chip L2 cacheline physical address which can be presented on the XBUS-2 interface and the necessary functionalities to interpret interrupts, emulated interrupts, and the functionalities to post interrupt events to queues managed by software in main memory. The XBUS-2 decodes Transmit Acknowledge (xb2_tx_ack) from interrupt targets and conveys the information to the addressed device's Interrupt Function for further processing.

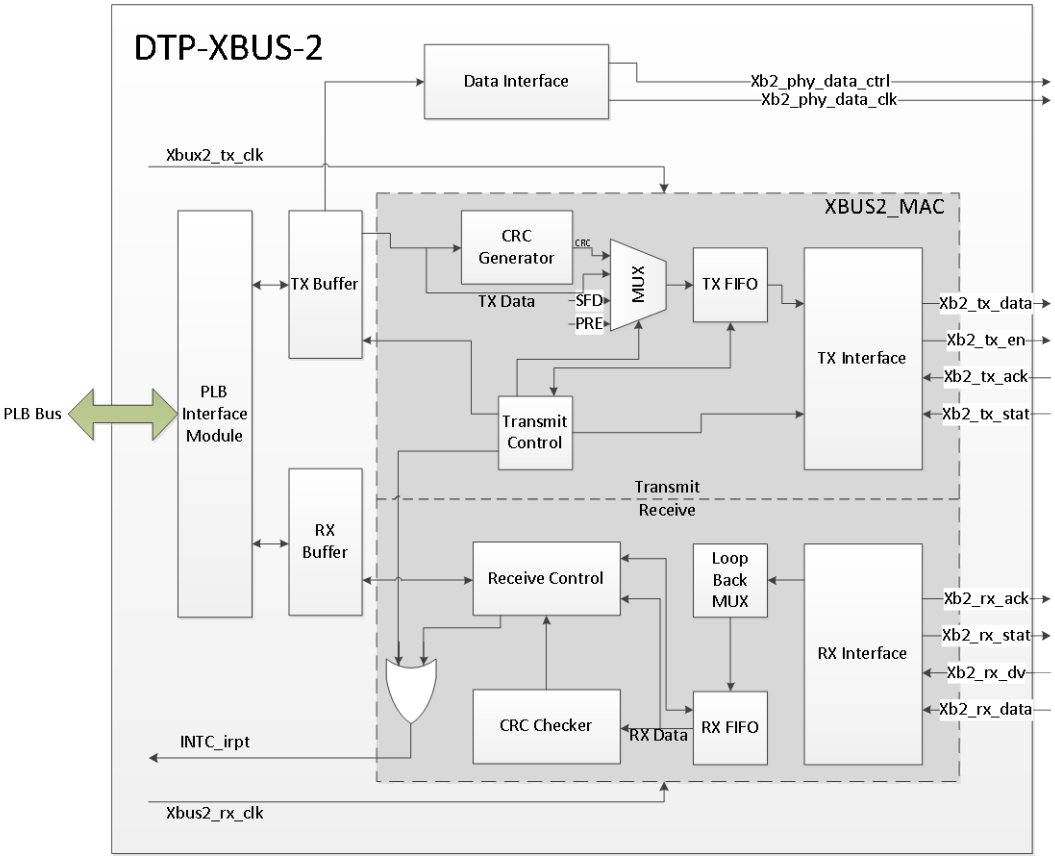

Figure 3.18: XBUS-2 Architecture

3.6.2 XBUS-2 Functional Description

The XBUS-2 contains several groups of functions as follows:

a) Transmission (TX) Buffer is a 2K Byte dual ported memory to hold transmit data for one complete frame and the transmit interface control registers. The Xb2_tx_data interface transfer data from the MAC to PHY, Xb2_tx_en for transmit enable, Xb2_tx_ack to acknowledge successful reception of transmitted data by receiver and Xb2_tx_stat reflects the status of PHY transmit register.

b) Reception (RX) Buffer is a 2K Byte dual ported memory to hold receive data for one complete frame and the receive interface control registers. Xb2_rx_data for data reception from the PHY to MAC, Xb2_rx_dv to indicate data validity, Xb2_rx_stat to reflect the status of reception buffer and Xb2_rx_ack to acknowledge successful data reception.

c) CRC Generator is used for the calculation of CRC for the frame that needs to be transmitted.

d) Transmit control multiplexer (MUX) arranges the frame that needs to be transmitted and sends pre-amble, Start of Frame Delimiter (SFD), frame data, padding, and CRC to the Transmit First-in-First-out (FIFO) in the required order.

e) Receiver Control logic is used to generate frame receive interrupt after CRC Checker verifies the CRC sequence of received frame.
f) Loop Back MUX when enabled, routes data on TX lines to RX FIFO.
g) CRC Checker module calculates the CRC of the received frame
h) TX Interface interacts with the physical layer (PHY) and sets the necessary conditions for physical transmission. The Transmit Control registers are updated after the frame is transmitted.
i) RX Interface interacts with the PHY and sets the necessary conditions for physical reception.
j) Data Interface provides access to PHY register for PHY management.

3.6.3 Data Transfer Protocol (DTP)

The DTP data is encapsulated in frames. The fields in the frame are transmitted from left to right or from the least significant bit to most significant bit as shown in Figure 3.19.

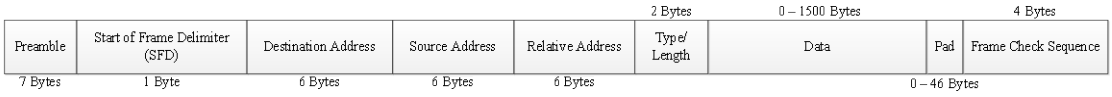

Figure 3.19: XBUS-2 Data Frame.

The description of each fields are as follows:

a) Preamble field. This field is used for synchronization. The preamble field contains seven bytes with the pattern of “10101010”. For transmission, this field is always automatically inserted by XBUS2_MAC. For Reception, this field will be stripped from the data packet.
b) SFD field marks the start of the frame and contains the pattern “10101011”. The Xb2_tx_en may go active during preamble but will be active prior to the start frame delimiter field. For transmission, this field will be inserted automatically by XBUS2_MAC. For reception, this field is always stripped from the data packet.
c) Destination Address field is six Bytes in length. The least significant bit of the destination address is used to determine if the address is an individual/unicast (0) or group/multicast (1) address. Multicast addresses are used to group logically related stations. This field is always provided in the packet data for transmission and is always retained in the receive packet data.
d) Source Address field is six Bytes in length. This field is always provided in the packet data for transmission and is always retained in the receive packet data.

e) Relative Address field is six Bytes in length. This field is used to re-route off-chip Destination Address in a daisy chained network (hence two dimensional). This field is always provided in the packet data for transmission and is always retained in the receive packet data.

f) Type/Length field is two Bytes in length. When used as a length field, this field represents the number of bytes in the following data field. As the maximum length of a data field is 1,500 Bytes, a value in this field that exceeds 0x05DC would indicate a frame type rather than length. This field is always provided in the packet data for transmission and is always retained in the receive packet data.

g) Data field may vary from 0 to 1500 bytes in length. This field is always provided in the packet data for transmission and is always retained in the receive packet data.

h) Pad field may vary from 0 to 46 Bytes in length. This is used to ensure that the frame length is at least 64 bytes in length (the preamble and SFD fields are not considered). The values in this field are used in the Frame Check Sequence calculation and not included in the length field.

i) Frame Check Sequence (FCS) field is 4 bytes in length. The value of the FCS field is calculated over the source address, destination address, relative address, length/type, data, and pad fields using a 32-bit Cyclic Redundancy Check (CRC) (Figure 2.20).

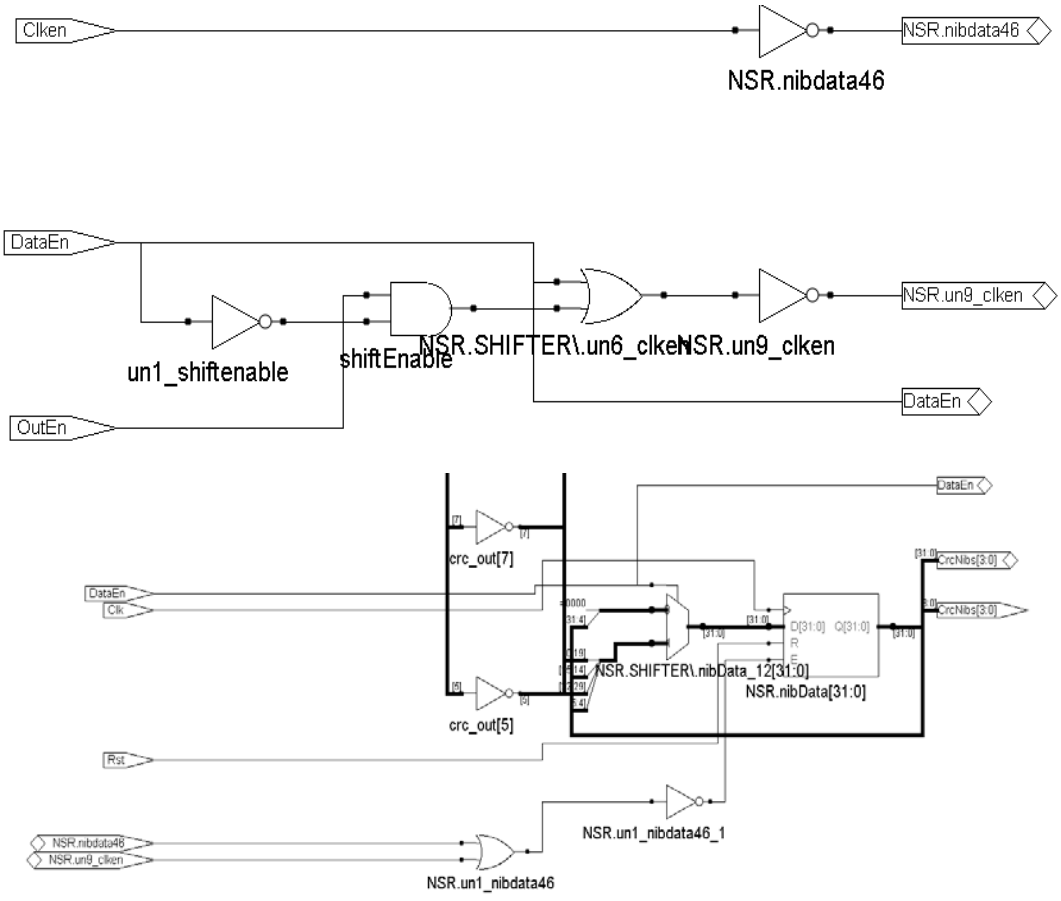

Figure 3.20: Snapshot of XBUS-2 CRC Generation Circuit.

The SPARC V9 processor is used to initiate DTP transfer protocol through the XBUS-2 interface.

3.7 SPARC V9 and the Data Transfer Protocol (DTP)

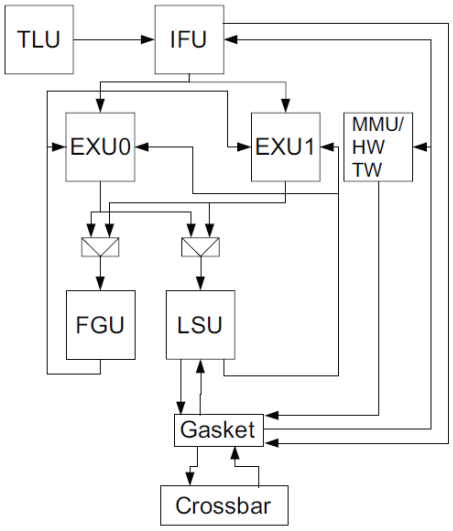

Figure 3.21: Core Block Diagram

The SPARC V9 core (Figure 3.21) has 8 pipelining stages which can be describe as the Fetch (F) stage, Cache (C) stage, Pick(P) stage, Decode(D) stage, Execute(E) stage, Memory(M) stage, Bypass(B) stage and finally the Write-back(W) stage (Figure 3.22). The pick stage enables up to two threads to be picked at each cycle. In the bypass stage, the Load Store Unit (LSU) forwards data to the integer register files. All integer operations pass through the bypass stage. Some instructions, such as load misses, fall into a long latency pipe after the bypass stage. Integer multiplies are pipelined between different threads. Integer multiplies block within the same thread. Integer divide is a long latency operation. Integer divides are not pipelined between different threads.

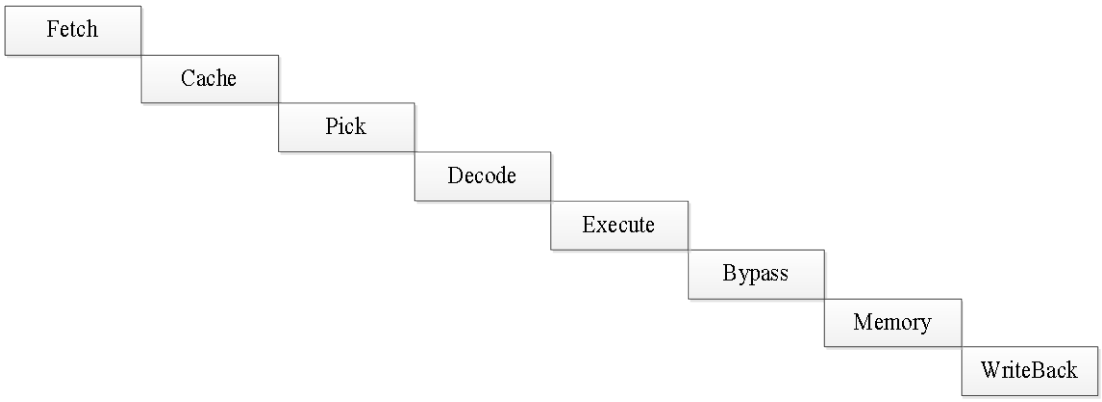

Figure 3.22: Integer Pipelining Operation

In contrast, the Floating-point operation has 12 stages (Figure 3.23). All floating point instructions are pipelined through the integer execute stage, and floating-point instructions that need integer resources obtain them during this stage. The floating-point register file (FRF) is accessed during the execute stage of the integer pipe. All floating-point operations except for divide and square root have a fixed latency of 6 cycles in the Float Graphics Unit (FGU) pipe. Floating-point data bypasses to dependent floating-point operations at execute during the float bypass (FB) and float writeback (FW) stages. Floating-point data writes into the FRF during the float writeback (FW) stage. The FGU executes all integers and floating-point multiplies. Multiplies are fully pipelined. This unit also executes all integers and floating-point divides. Up to two divides can be below pick at a time across all threads. The floating-point pipeline stages are illustrated in the figure below.

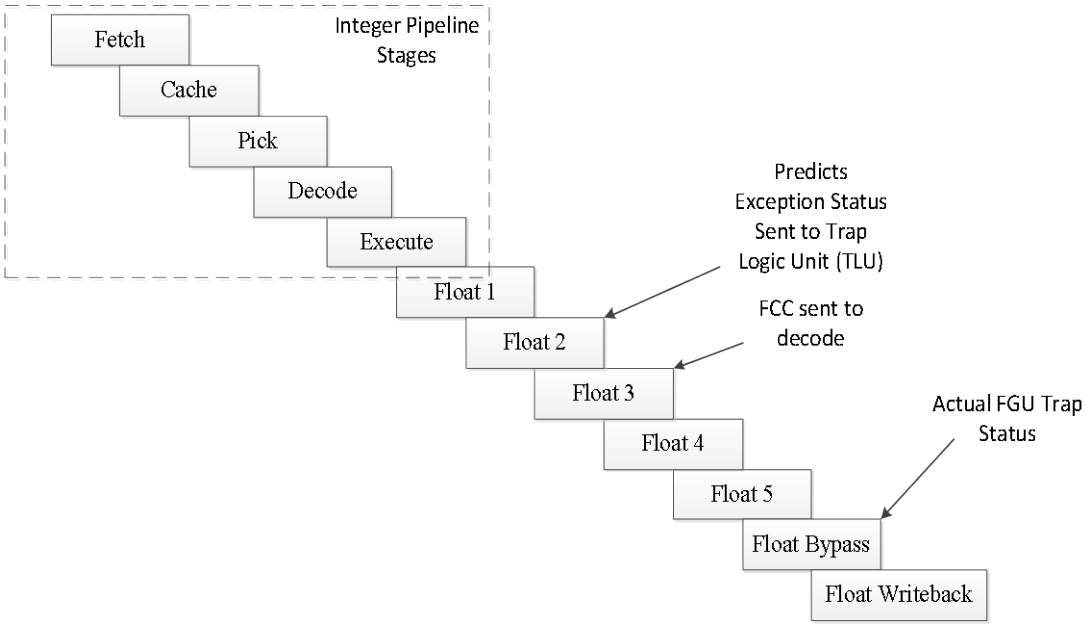

Figure 3.23: Floating Pipeline stages

The Instruction Fetch Unit (IFU) (Figure 3.24) feeds instructions from the memory or XBUS-2 to the rest of the core. This unit generates the program counter (PC) and maintains the instruction cache (icache). The IFU covers the first three stages of the pipeline operations, or the Fetch/Cache, Pick and Decode. Each cycle, the Fetch unit fetches up to four instructions for one thread. The fetched instructions are written into instruction buffers (IBs) which are then fed to the pick logic. Each thread has a dedicated 8 entry IB. The fetch unit maintains all PC addresses for all threads. The fetch unit redirects threads due to branch mis-prediction, LSU synchronization, and traps. The fetch unit handles instruction cache misses and maintains the Miss Buffer (MB) for all threads. The MB ensures that the L2 does not receive duplicate icache misses. The Fetch stage is further divided into three micro-

stages, or the “Before Fetch”, “Fetch”, and “Cache”. During the “Before Fetch” stage, the Fetch unit picks the next thread to pick. The next fetch address is calculated at this stage. In the “Fetch” stage, the icache data array, the tag array, and the instruction TLB (ITLB) are accessed in parallel. ITLB hit or miss is determined during this cycle. The data read from all 8 ways of the icache is latched at the end of this cycle. Physical address information from the ITLB and from the tag array is latched at the end of the fetch stage. Hit or miss of the icache is determined during the “Cache” stage. Way selects choose the correct instruction data in the cache stage. The cache data is aligned. This aligned data is written into the instruction buffers of the fetched thread.

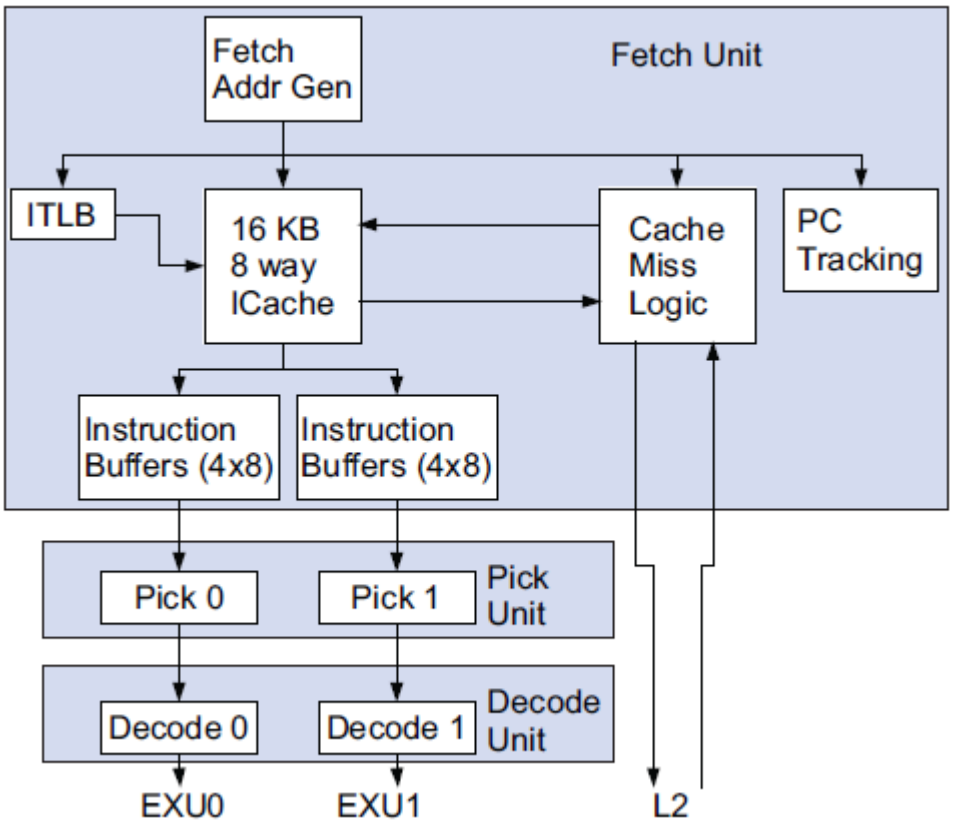

Figure 3.24: Instruction Fetch Unit

The fetch unit can only fetch 1 thread at a time because the icache has one port. A Least Recently Fetched (LRF) mechanism ensures fairness in picking this thread out of the 8 possible threads. Every cycle the fetch unit picks a LRF thread from the set of all READY threads. The picked thread ID (if there is one) is written to the current fetch thread ID register.

The pick unit attempts to find two instructions to execute among eight different threads. The threads are divided into two different thread groups (TG) of four threads each: TG0 (threads

0-3) and TG1 (threads 4-7). The Least Recently Picked (LRP) ready thread within each thread group is picked each cycle. The pick process within a thread group is independent of the pick process within the other thread group. This independence facilitates a high frequency implementation. In some cases, hazards arise because of this independence. For example, each thread group may pick an FGU instruction in the same cycle. As the core has only one Floating-Graphic Unit, hardware hazard results. The decode unit resolves hardware hazards that result from independent picking. Pick maintains a state machine per thread to indicate whether the thread can be picked. A thread is either in READY state or in WAIT state. If a thread is READY and IB entry 0 is valid, it can be picked. If a thread is not READY, then it is in the WAIT state. A thread remains in the WAIT state until the condition or conditions that caused the transition to WAIT are resolved or the thread is flushed. A thread is in WAIT state if any wait conditions exist for the thread. A thread is in READY state if no wait conditions exist for the thread. Pick is initiated before the type of instruction being picked can be determined. Once the instruction type is known, dependency and resource limitations may require the pick to be cancelled for correct machine behaviour. A cancel pick transitions the picked thread to WAIT the next cycle unless the condition or conditions giving rise to the hazard or hazards resolve this cycle. If the hazard or hazards resolve this cycle, the thread remains in the READY state.

Threads enter the WAIT state in one of two ways. A thread may enter WAIT after it has been picked to allow dependency and/or hardware hazards to resolve. Alternatively, a thread may enter WAIT before it is actually picked.

The decode unit decodes one instruction from each thread group (TG0 and TG1) per cycle. Decode determines the outcome of all instructions that depend on the CC and FCC bits (conditional branches, conditional moves, etc.). The integer source operands rs1 and rs2 are read from the IRF during the decode stage. The integer source for integer stores is also read from the IRF during the decode stage. The decode unit supplies pre-decodes to the execution units. This unit also resolves scheduling hazards that are not detected during the pick stage between the 2 thread groups.

The Execution Unit (EXU) (Figure 3.25) executes all integer arithmetic and logical operations except for integer multiplication and division. The EXU calculates memory and branch addresses. The unit also handles all integer source operand bypassing.

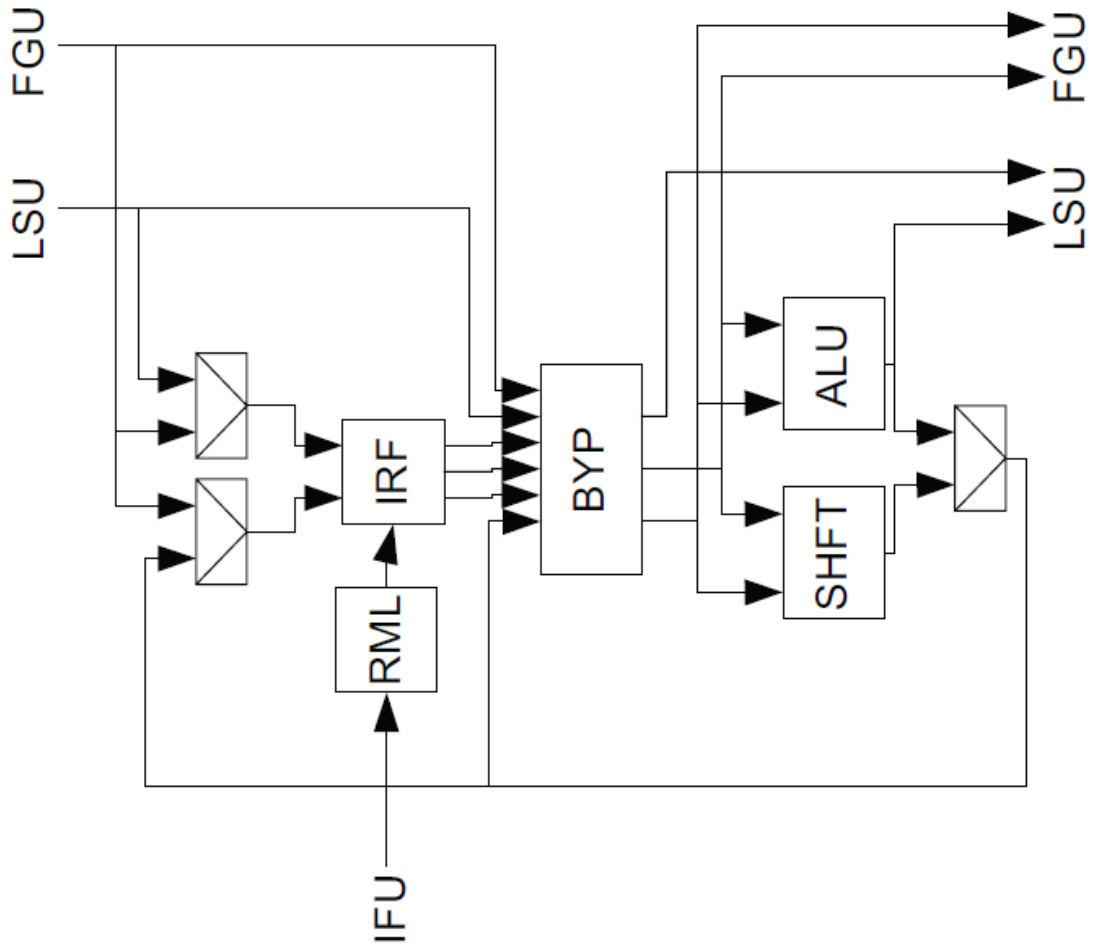

Figure 3.25: Execution Unit

The execution unit comprises ALU (Arithmetic Logic Unit), Shifter (SHIFT), Bypass (BYP), Integer Register File (IRF) and the Register Management Logic (RML). The RML tracks the list of registers for that particular instruction and feed that values held in those registers to the IRF. The BYP will decide if that instruction is a Floating point instruction, an Integer instruction or a memory instruction.

Load Store Unit (LSU) (Figure 3.26) handles memory references between the core, the L1 data cache, and the L2 cache and XBUS-2. All communication with the L2 cache is through the crossbars (processor to cache and cache to processor, a.k.a. PCX and CPX) via the gasket. The LSU ensures compliance with the Total Store Order (TSO) memory model with the exception of instructions which are not required to strictly meet those requirements (block stores, for example). The LSU is responsible for handling all Address Space Identifier (ASI) operations including the ASI decode and initiating transactions on the ASI ring. The LSU is also responsible for detecting the majority of data access related exceptions.

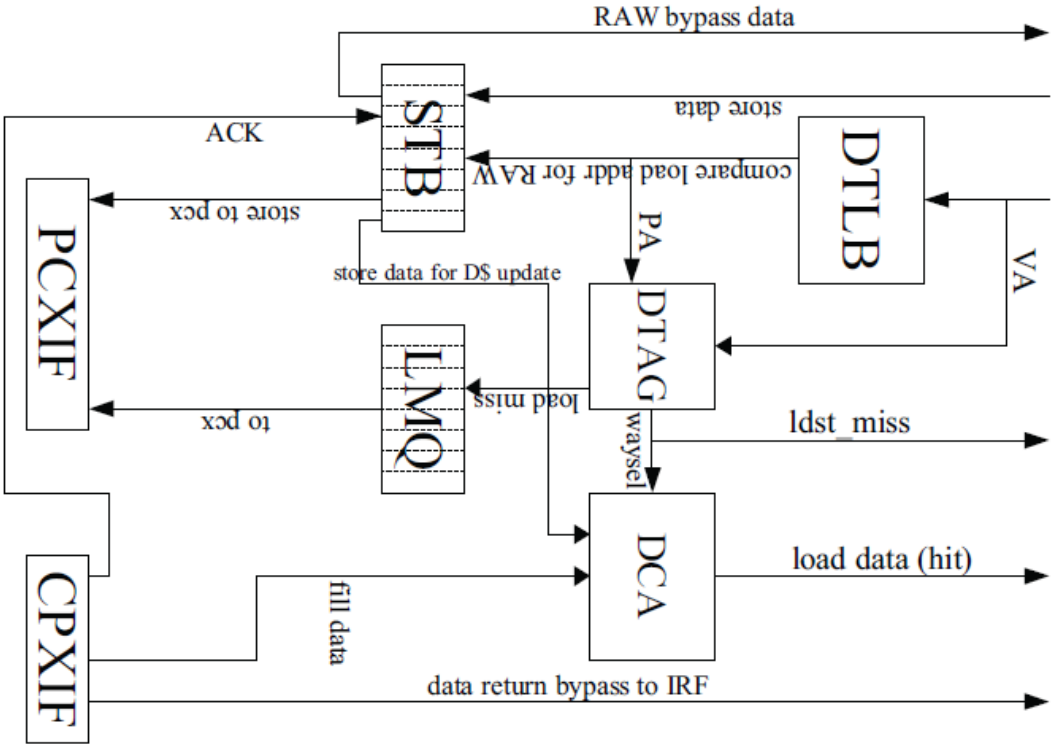

Figure 3.26: Load Store Unit

The Data Cache Array (DCA) and Data Tag (DTAG) make up the level 1 data cache. The DTLB provides virtual to physical and real to physical address translation for memory operations. The Load Miss Queue (LMQ) stores the currently pending load miss for each thread (each thread can have at most one load miss at a time). The Store Buffer (STB) contains all outstanding stores. The PCX interface (PCXIF) controls outbound access to the PCX and ASI controller. The CPX interface (CPXIF) receives CPX packets (load miss data, store updates, ifill data, and invalidates), stores them in a FIFO, and sends them to the dcache. The dcache is write-through, so the LSU sends all stores to the L2. The L2 maintains a copy of the L1 tags for coherency. Hit or miss in the L1 for stores is determined by the L2. Stores which hit the L1 will update the dcache. Stores which miss do not allocate. Cache updates and invalidations for stores occur after the ack has been received from the L2. All stores within a thread are processed in order. When the L2 sends the store ack, the LSU writes the ack into the FIFO. When the ack reaches the head of the Cache Processor Queue (CPQ), there are two possibilities. If the ack indicates a cache update is required (if the store hit to the L1 cache) it must wait for a hole to open in the dcache pipe before the update can proceed and the store dequeued from the store buffer. If the store missed the cache and no update is indicated, the store can be immediately dequeued from the store buffer. (Store misses do not allocate in the L1 dcache.) The L2 directory controls allocation since it has the

most current copy of the L1 tags and valid status. The allocation information is embedded in the invalidation vector that is part of the store ack packet.

3.8 Ultra-High-Bandwidth Data Transfer Operation

Today's communication systems demand very high computational performance and energy efficient signal processing. The traditional way to move large amounts of data between devices is to use a bus, a collection of signals that carry similar data and perform a common function. XBUS-2 performance includes a range of components and concepts. This section will discuss the performance of the XBUS-2 channel itself. Monitoring the total amount of traffic on the XBUS-2 requires a device that operates in promiscuous receive mode, reading in every frame seen on the XBUS-2 Transceivers (Figure 3.27). Looking at every frame with a general-purpose computer requires a network interface and computer system that can keep up with high frame rates. This is achieved through the integrated JTAG interface, re-programmed to perform a port scan test on the transceivers and loopback on-chip into the monitoring system to gauge the overall transmission performance. The very same method is also used to benchmark the performance of SoC with DTP-XBUS-2 system against those without in the computation of statistics for large image (Image Processing). The figure below illustrates the frame composition on the XBUS-2 channel.

The preamble field which consists of 7 bytes is used to announce the frame and to enable all receivers on the network to synchronize themselves to the incoming frame. The start of frame delimiter is a continuation of the preamble. Both the preamble field and the start-of-frame delimiter field are removed by the controller when it places a received frame in its buffer. Similarly, when a controller transmits a frame, it prefixes the frame with those two fields or a preamble field. The destination address identifies the recipient of the frame. Each field can consist of two or more subfields, whose settings govern such network operations as the type of addressing used on the XBUS-2 channel, and whether the frame is addressed to a specific device or more than one device.

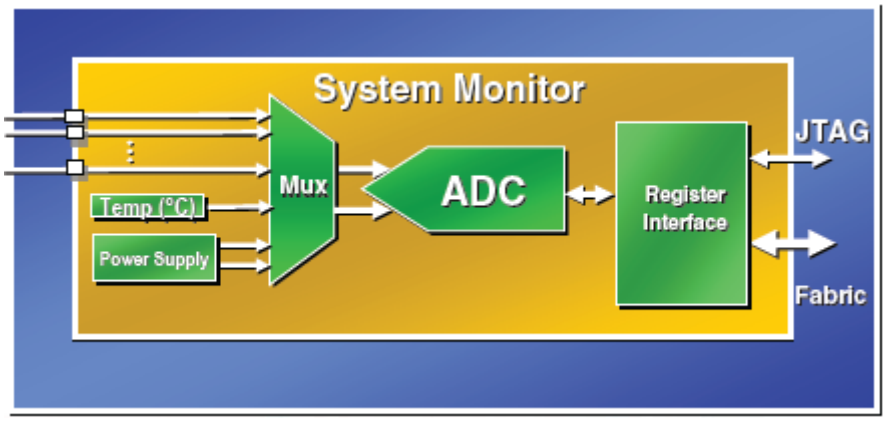

Figure 3.27: On-chip System Monitor

The source address field identifies the station that transmitted the frame. Like the destination address field, the source address can be either two or six bytes in length. Both destination and source addresses are normally displayed by network monitors in hexadecimal, with the first three bytes separated from the last three by a colon (:) when six-byte addressing is used. When a destination address specifies a single device, the address is referred to as a *unicast address*. A group address that defines multiple devices is known as a *multicast address*, while a group address that specifies all devices on the network is referred to as a *broadcast address*. The reference address field defines the offset for both the destination address in multi-devices network. The two byte length field indicates the number of bytes contained in the data field. The minimum size frame must be 64 bytes in length from preamble through FCS fields. This minimum size frame ensures that there is sufficient transmission time to enable Xb2 transceivers to detect collisions accurately, based on the number of extensions specified in the network and the time required for a frame to propagate through the chain. Based on the minimum frame length of 64 bytes and the possibility of using two-byte addressing fields, this means that each data field must be a minimum of 46 bytes in length. At ultra-high bandwidth data transfer operation, this specification will not provide a frame duration long enough to permit a 100-device extension. This is because at this data rate there is a high probability that a device could be in the middle of transmitting a frame before it becomes aware of any collision that might have occurred at the other end of the segment. Hence, a carrier extension is introduced to extend the frame to a minimum of 512 bytes rather than 64 bytes. If the information to be placed in the field is less than 46 bytes, the remainder of the field must be padded. The maximum length of the data field is 1500 bytes. The frame check sequence provides a mechanism for error detection. Each transmitter computes a cyclic redundancy check (CRC) that covers all the address fields, the length field, and the data field. The transmitter then places the computed CRC in the four-byte FCS field. The CRC treats all the fields as a single long binary number. The *n* bits to be covered

by the CRC are considered to represent the coefficients of a polynomial M(X) of degree n − 1. Here, the first bit in the destination address field corresponds to the Xn−1 term, while the last bit in the data field corresponds to the X0 term. Next, M(X) is multiplied by X32, and the result of that multiplication process is divided by the following polynomial:

G(X)=X32+X26+X23+X22+X16+X12+X11+X10+X8+X7+X5+X4+X2+X+1 ----------(1)

Note that the term Xn represents the setting of a bit to a 1 in position n. Thus, part of the generating polynomial X5 + X4 + X2 + X1 represents the binary value 11011. This division produces a quotient and remainder. The quotient is discarded, and the remainder becomes the CRC value placed in the four-byte FCS field. This 32-bit CRC reduces the probability of an undetected error to 1 bit in every 4.3 billion, or approximately 1 bit in 2^32 − 1 bits. Once a frame reaches its destination, the receiver uses the same polynomial to perform the same operation upon the received data. If the CRC computed by the receiver matches the CRC in the FCS field, the frame is accepted. Otherwise, the receiver discards the received frame, as it is considered to have one or more bits in error. The receiver will also consider a received frame to be invalid and discard it under two additional conditions. Those conditions occur when the frame does not contain an integral number of bytes, or when the length of the data field does not match the value contained in the length field.

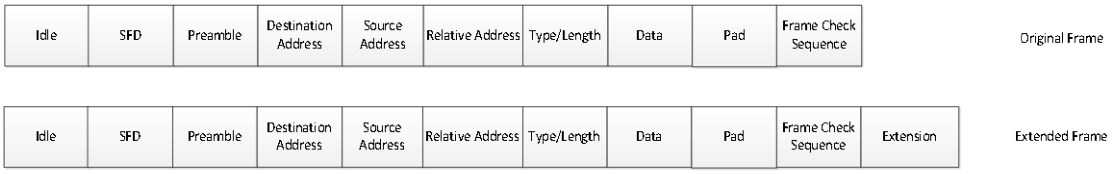

Figure 3.28: Frame Extension for collision detection prior to frame bursting.

Frame bursting is used to counteract the overhead associated with transmitting relatively short frames. Under frame bursting, each time the first frame in a sequence of short frames successfully passes the 512-byte collision window using the carrier extension scheme and subsequent frames are transmitted without including the carrier extension. The effect of frame bursting is to average the wasted time represented by the use of carrier extension symbols over a series of short frames (Figure 3.28). The limit on the number of frames that can be carried is a total of 1500 bytes for the series of frames, which also represents the longest data field. In addition to enhancing network use and minimizing bandwidth overhead, frame bursting (Figure 3.29) also reduces the probability of collisions occurring.

This is because the burst of frames are only susceptible to a collision during the first frame in the sequence (Figure 3.28).

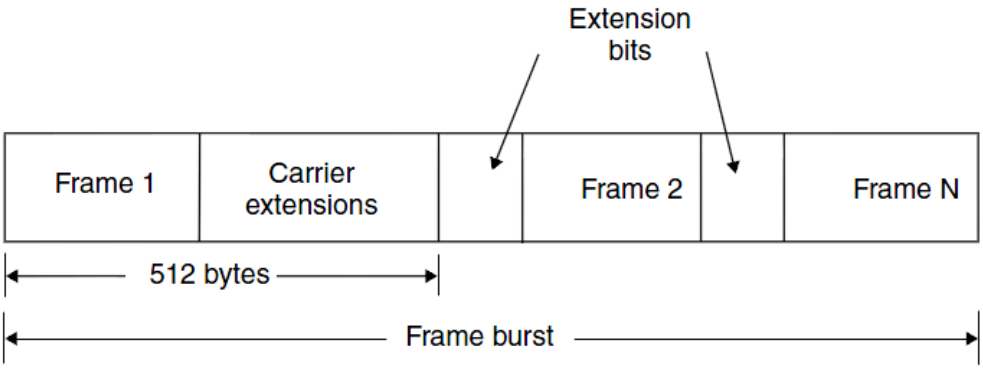

Figure 3.29: Frame Burst

3.9 Power-On Framework

This section will discuss the power-on framework for DTP-XBUS-2 system. The standard SPARC V9 tool-chain is used to generate the memory initialization sequence for power-on. The framework requires the following packages as shown in the figures below (Figure 3.30 and Figure 3.31).

```
sudo apt-get update
sudo apt-get -y install build-essential
sudo apt-get -y install make
sudo apt-get -y install gcc
sudo apt-get -y install g++
sudo apt-get -y install flex
sudo apt-get -y install bison
sudo apt-get -y install patch
sudo apt-get -y install texinfo
sudo apt-get -y install libncurses-dev
```

Figure 3.30: Framework packages

```
:100100001820F000A82104501860F000A863000037
:1001100018800000A88403DC18A00000A8A5041C17
:10012000E0A52002BC0500001000000A1500000038
:1001300084C40000D40330009C6300049C84000449
:0C03D000844100044400480009C2100749B
:1003DC000000000100000002000000030000000407
:1003EC000000000500000006000000070000008E7
:0400000300000100F8
:00000001FF
```

Figure 3.31: Memory Initialization Sequence (Hex)

The power-on sequence comprises of the **Start Code**, represented by ":", **Byte Count:** The first two hex digits, after the start code in order to indicate the number of bytes (hex pairs) in the data field e.g. byte count 0x10 or 0x20 represents 16 or 32 bytes of data respectively. **Address:** The four hex values, after the byte count, to identify the 16 bit big-endian address of the beginning of data in the memory. **Record type:** The two hex values, after the address, to define the type of the data field. There are six types of data fields identified by the record type (00 to 05). The record type **00** indicates a data record of 16-bit address. The record type **01** identifies an end-of-file record and record type **03** identifies a start segment address record. **Data:** A sequence of **n** bytes (2^n hex values) of data, where the byte count specifies **n**. And finally the **Checksum:** The last two hex-values which are the two's compliment sum of the values in all fields except the start code (:) and the checksum itself.

The compiled program is boot-loaded into DTP-XBUS-2 during power-on to initiate basic tests that verify basic functionalities of DTP-XBUS-2. The following scripts are needed for the compilation.

Linker Script

A linker script is used to set up a memory-map of applications (Figure 2.32). This script defines the addresses of the memory space, the positions and sizes of the stacks and heaps in the memory and the contents of each memory-mapped space.

Startup Script

The startup script prepares the RAM for data initialization. The script also includes additional code for the stack initialization and code for transferring initialized data and static variables from the ROM to the RAM.

```
PROVIDE (__stack = ADDR(.bss) + SIZEOF(.bss) + STACKSIZE + OFFSET);
PROVIDE (__copy_start = _copy_start);
PROVIDE (__copy_end  = _copy_end);
PROVIDE (__copy_adr  = _copy_adr);

MEMORY
{
      rom (rx)   : ORIGIN = 0x00000000, LENGTH = 0x0000FFFF
      ram (rwx) : ORIGIN = 0xF0000000, LENGTH = 0x000F0000
}

SECTIONS
{
      .text 0x100 :
      {
         _stext = .;
         *(.text)
         _etext = .;
      } > rom
```

Figure 3.32: Linker script

At the beginning of power-on, the HEX file will be loaded into the ROM starting from the reset address of SPARC V9. The data sequences (instruction/data) are then parsed and loaded into the ROM as RAM is not initialized at this stage. Finally, a test bench will be created to instantiate the DTP-XBUS-2 system and drive the system clock and the system reset sequence. The system clock for the CPU Subsystem is set to 100 MHz. When the test bench asserts a system reset, the SPARC V9 sends the READ request and initiate DTP-XBUS-2 for basic transfer operation as described in chapter 6. After the completion of basic verification, the DTP-XBUS-2 system is configured accordingly for performance analysis as described in Chapter 6.

3.10: Conclusion

In this chapter, the environment structure and processor architecture relevant to DTP-XBUS-2 are presented. Furthermore, this chapter describes the basic operations of the DTP-XBUS-2 backbone devices. The building blocks are presented in the following sequence: Complete system implementation, the DTP-XBUS-2 integration strategy through the on-chip processor local bus structure with the SPARC v9 processor, DTP-XBUS-2 memory controller architecture and implementation, the DTP-XBUS-2 communication protocol and architecture, a complete treatment about the customization of SPARC v9 processor for DTP-XBUS-2 implementation, DTP-XBUS-2 transfer operation and finally the framework for power-on.

Chapter 4: Verification Concepts

This chapter discusses the various verification concepts and methodologies. Product reliability is of major concern for many companies. The goals of product verification are to screen out defects in CPU architecture, manufacturing defects, bin for speed, and verify that devices meet the published Direct-Current (DC), Alternating-Current (AC), and frequency specifications. Today, product verification is the only way to accomplish these goals.

4.1 Minimal Verification Requirements

There are many attributes of a good product test. A good product test has the following attributes [26]:

1) Passes only good product and fails only bad product. This optimizes yields and screens out defects.
2) Has a short test time. This minimizes product costs and reduces capital equipment needs.
3) Is comprehensive. This ensures coverage of all structures in the device under test.
4) Is maintainable. This facilitates update in order to reflect product changes and testing improvements by anyone with the need and with minimal risk (for example, in a common format).
5) Is repeatable. The adopted test method should provide consistent results over time.
6) Enables line yield and process learning; it provides data in support of yield improvement and other manufacturing optimization activities.

Defects in the device, often modelled as faults should be screened out during the product verification phase. Often, adding screens increases test time, hence arises the need to reduce test time and cost. This dilemma of increased quality via added testing and a constant drive to reduce test time poses a major conflict to the test and manufacturing community [26].

4.2 Test Methods

There two general types of verification approaches for silicon devices.

1) Functional Test
2) Structural Test

4.2.1 Functional Test

Functional test (Figure 4.1) causes a device to operate very much like it would in actual operation. Certain patterns are fed in as input on the input pins and the correct output is watched for on the output pins.

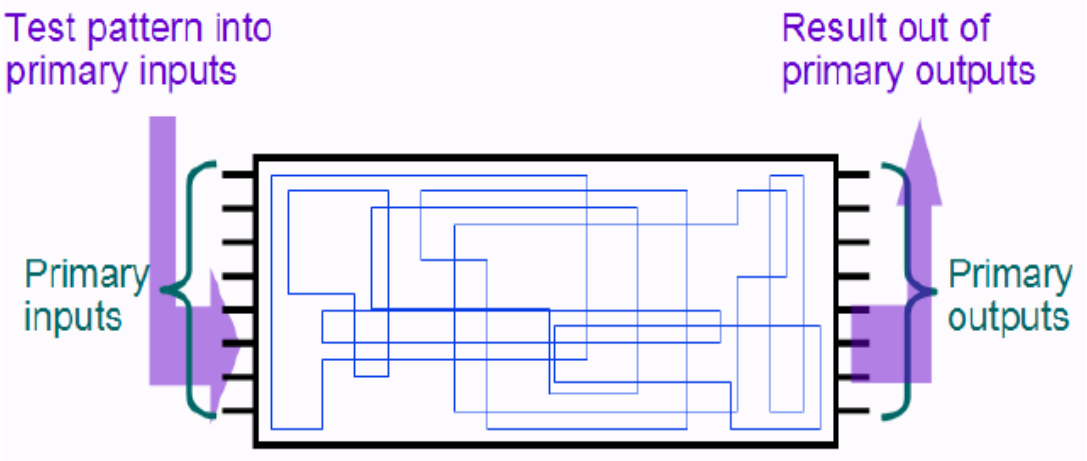

Figure 4.1: Functional Test

The following are the properties of functional test:

1) Functional test is used to verify product features and functionalities.
2) Test patterns must be customized for each product features in functional test.
3) Functional test can be used to operate the device at full or functional speed in order to characterise device performance.
4) Functional test does not require extra on-die logic nor consume any area on-chip.

The sources of functional errors in semiconductor devices are usually associated with the following reasons:

1) Ambiguities in product feature specifications. Unclear product feature definitions prior to actual design implementation usually lead to design error(s), hence functional faults.
2) Ambiguities in product operation specifications. Improper selection of device architecture often lead to mismatch in device operation hence causes functional errors.
3) Design implementation errors.

The main objective of functional verification is to resolve design discrepancies versus the expected architectural specifications and to ensure proper device operation [27]. However, there are certain errors that cannot be observed in functional verification during simulation.

These errors include functional faults that can only be observed during power-on or hardware reset or hardware specific design errors, faults that can be observed but require significant computational resources, and the occurrence of multiple faults at the same time that that prevent a clear observation of cause(s) of the errors.

In functional verification, different approaches are used to increase the test coverage of the DTP-XBUS-2, as described below:

4.2.1.1 Black-Box Verification

In this approach, the design under verification (DUV) is treated as a black box (closed box) without consideration of the device architecture. The DUV refers to Xilinx ML505 Development Board with DTP-XBUS-2 architecture implemented as a soft-Intellectual Property (IP) core. The DUV is accessed only through available external interfaces or JTAG chain, and hence without access to its internal components. This verification approach lacks controllability in terms of setting up a certain functional state of the design, isolation of a particular functionality or the ability to correlate the output response to a particular input stimulus. The test bench is developed in parallel to the design implementation. However, this methodology is not suitable for large design verification due to significant discrepancy in the number of functional blocks versus controllability or test coverage.

4.2.1.2 Gray-Box Verification

In this approach, the DUV is treated as a closed box with knowledge of the device internal architecture (DTP-XBUS-2). The input stimulus is applied through the JTAG interfaces with the mission to activate specific macro components of the DUV, for example to set an internal Finite-State-Machine (FSM) to a particular state. This approach significantly increases the verification coverage and at the same time reduces computational requirements. *Design-for-verification* is the phase in which device architecture is constantly being fine-tuned to increase test coverage. An example of such modifications is the addition of easily controllable registers to set up a particular internal state of the design.

4.2.1.3 White-Box Verification

This verification approach offers full controllability and observation of the performance and operation of each functional component in the device such as setting up a particular state or bypassing some internal units. Such verification methodology heavily depends on the particular device implementation hence the test bench can only be developed once the device architecture is implemented.

4.2.2 Structural Test

Structural test (Figure 4.2) adds scan chains throughout a device to carry (or "scan") test patterns deep into the device (Figure 4.3). These scan chains are called design-for-testability (DFT) circuitry [37]. The device is clocked and the data from one scan chain passes through the device's internal circuitry and into the next scan chain, where it is carried away to be examined in the tester [28].

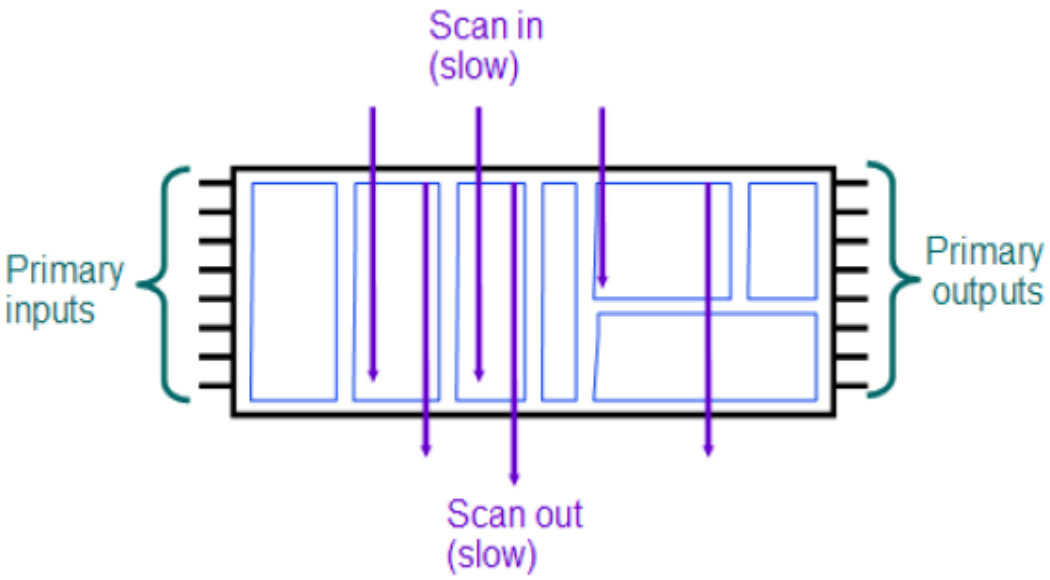

Figure 4.2: Structural Test (Overview)

Structural test has the following attributes:

1) Structural test is used to detect manufacturing defects. This test is used to ensure all transistors are present, connected correctly, and operate at expected specifications. Structural test however does not ensure that the "structures" provide the desired features and capabilities. Design validation or functional validation must be carried out separately.
2) Structural test require "design for testability" (DFT) circuitries on-die that acts as internal probes to monitor DTP-XBUS-2 component connections.
3) In structural test, test patterns can be algorithmically generated. This decreases the number of-hours required to generate the test programs, and hence a significant saving to test-time reduction.
4) Structural test can be operated at less than actual full device operating speed. This provides more flexibilities in on-chip test implementation and greatly simplifies overall DTP-XBUS-2 system design.

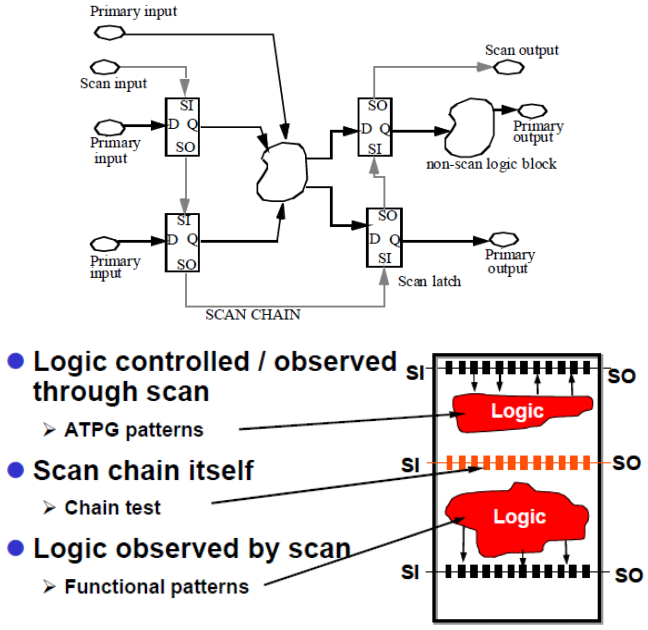

Figure 4.3: Scan chain in structural test

The ultimate challenge in product verification has always been the identification of effective methodologies that screen out defects which at the same time also provide sufficient test coverage for the complex silicon devices at a low cost. Device complexity and performance are beginning to scale from the conventional functional test and tester capabilities, and hence the only probable solution in order to keep up with these new emerging test requirements is the adoption of design for testability (DFT). DFT is used to achieve higher coverage on the DTP-XBUS-2, which also enables the production of high quality product verification in a cost-effective manner. DFT techniques have enabled the generation of high quality product tests, debug of early silicon and analysis of failing parts in the DTP-XBUS-2.

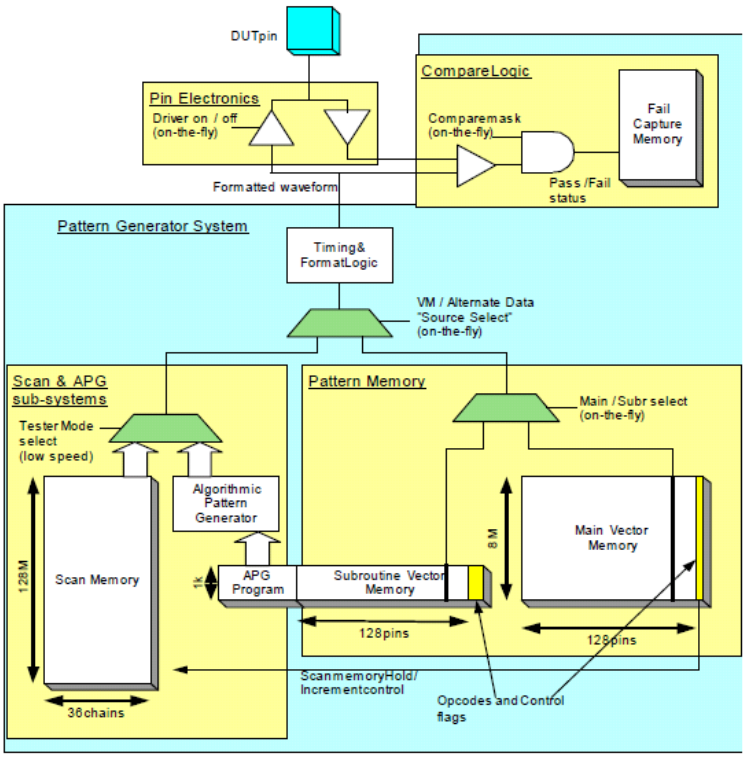

Figure 4.4: Structural Tester minimum requirements.

In the past, microprocessors relied solely on functional test for product verification. With the continued increase in design complexity, size, speed, plus the emergence of new architectural features, this practice is proving to be both impractical and cost ineffective. In addition, the high cost of semiconductor automatic test equipment (ATE) is proving to be an economic and technical bottleneck. As devices get more complex, test-time increase and the test infrastructures become more complex where more I/O pins and tester memory will be required (Figure 4.4). As test time goes up this also increases the overall test cost. All these factors will unnecessarily increase the total cost of functionally testing the DTP-XBUS-2 device. The incorporation of relevant DFT features helps to resolve these challenges, hence allowing the DTP-XBUS-2 device to be tested more completely, quickly, and economically.

In summary, structural tests target manufacturing defects and attempt to ensure the manufacturing correctness of basic devices such as wires, transistors, etc. Functional tests, on the other hand, target device functionality and attempt to ensure that the device is functioning correctly. Functional tests are written primarily for architectural verification and silicon debug. They can be used for manufacturing testing also, as is done with the DTP-XBUS-2 device. Structural tests, on the other hand, are used primarily for manufacturing testing, and can be used for failure analysis and fault isolation.

The various verification technologies are discussed in the next section.

4.3 Verification Technologies

Due to the increasing complexity of system specifications, various verification methodologies are required for detecting design errors at the early stage of the development process as well as for ensuring the performance characteristics of the final product. System level simulation enables the evaluation of system specifications against the requirements at early stages of the development, before proceeding to hardware implementation. Simulation allows one to execute the system specification at different levels of abstraction, hence allowing verification of the correct functionality of the system's specification with respect to its functional requirements.

In summary, the technologies available to perform product verification are given below.

1) Modelling and Simulation
2) Formal Verification
3) Product Emulation

4.3.1 Modelling and Simulation

The simulation is the process of executing a given system specification in a computer based environment. The DTP-XBUS-2 system consists of a coherent combination of hardware and software. As such, simulation has to be applied not only to each of the hardware and software partitions and components in part, but also to the entire system as a whole (co-simulation). The DTP-XBUS-2 simulation environment is divided into two phase, **executable specifications** and **simulation-based validation**. The idea behind executable specifications is to remove the process of parsing through a large quantity of documents describing the desired functionality of the DTP-XBUS-2 and at the same time this method of verification is used to provide a better insight into the working mechanism of the system (hence modelling). The executable specification is gradually refined to contain more and more implementation details during system development. The refinement of the original executable specification can be targeted towards reaching a higher abstraction level simulator of the DTP-XBUS-2. Ideally, this specification is expected to provide complete and accurate system specifications.

As the system design size and complexity increase, high abstraction level design methods are needed to rapidly explore design space and verify system functionality. Hence, this modelling method can be used for simulating the system and evaluating its functionality and performance. Based on the results obtained from simulating the system models, the initial models of the systems may be refined and improve until satisfactory ones are obtained.

Moreover, system verification can be done before actual device implementation using high-level models, which alleviates the burden of verification. The system models that are created during the development process should satisfy the requirements of the actual designs, which in this case is the DTP-XBUS-2.

Simulation, on the other hand, requires complete documentation of each of the functional components in the DTP-XBUS-2 device. Simulation is a more commonly adopted method used to identify design errors during the validation phase. In the *simulation-based verification* next inputs values and expected responses of the components are predicted in terms of its current state and input values. This type of verification methodology will require a test bench together with an actual design implementation. The test bench is used to apply input values to the DUV, which in this case is the DTP-XBUS-2. The next state values of the DUV are computed based on these input values. The captured output value is finally verified against the computed state. In order to increase test coverage, this verification methodology will require the computation of all expected response, hence making it relatively impractical for large-scale design.

4.3.2 Formal Verification

Formal verification is a practical solution to handle limitations of simulation-based verification in large scale designs. In formal verification, a behavioural model of the system (Executable Specification) is mathematically derived. Formal verification will proceed to verify or reject a given property of a hardware implementation through a set of logical methods and mathematical equations. In formal verification, a system is considered verified if the derived mathematical equations (Executable Specification) that describe the system are proven to be correct. Hence, any property proved by the formal verification holds for all possible input vectors applied to that implementation. The major advantage of formal verification techniques is the ability to make universal statements about a property of a design implementation. These statements hold for all possible input streams without requiring test vectors to be applied or re-applied. There are two major categories of formal verification techniques, as given below:

1) Equivalence checking and
2) Property checking.

4.3.2.1 Equivalence Checking

In this technique, two formal representations of the design implementation (before and after a given transformation) are provided as input to an equivalence-checking tool. This tool

creates a canonical representation of each implementation. Since the canonical representation is unique for every Boolean function under an assumed set of conditions (e.g., variable ordering), the methodology to prove the equivalence of these two representations is typically straightforward. The most common input representations of a design to equivalence-checking tools are RTL (Register Transfer Level) and design net-list. The development effort however, scales with the size of the design, making it a difficult task. Moreover, creating a canonical representation for very large system is not practical.

4.3.2.2 Property Checking

Property checking is another form of formal verification approach that uses the Executable Specification. Given a formal description of the design implementation (e.g., an RTL description) property checking approach verifies that a given property described in temporal logic holds for the given implementation. The design properties are specified as a set of assertions. The following are the advantages of the property checking method.

1) The properties can be described at any level of the product specification and the design creation. They can be collected incrementally as specification and development proceeds.
2) Property checking can be performed in the early stages of the design even when a verification environment is not available to provide a test-bench.
3) The properties can be used with emulation-based verification and simulation-based verification.
4) Property checking provides the coverage collection that is needed to check the verification completeness.
5) Property checking is a static technique in which no test bench or logic simulator is required.

There are multiple languages to facilitate property checking, including

(i) Property Specification Language (PSL) and

(ii) Verilog Hardware Description Language (properties are defined in the form of assertions).

4.3.2.3 Limitations of Formal Verification

It is an often-repeated myth that formal verification ensures complete test coverage for a particular system. However, an absolute flawlessness of systems cannot be guaranteed with a vigorous formal verification alone. Since this verification method only allows the detection

of design faults and does not identify fabrication faults or faults while a system is in use. The verification checks the correctness of statements according to the formal specification of a design which can be incomplete or faulty itself. Moreover, the verification tools may contain faults in their programs. Hence, the formal verification should be taken as an adjunct to but not as a substitute for standard quality assurance methods.

4.4 Verification Methodologies

The previous section describes the various technologies available for product verification. This section describes the available methods for implementation. Complete verification coverage will require a combination of different technologies and multiple facilities. Different methodologies will be required to bring together these tools and facilities. Most commonly used types of these methodologies are:

1) Assertion-based Verification
2) Coverage-Driven Verification (CDV).

The assertion-based verification focuses on how assertions can be involved throughout the design flow and across multiple tools. The coverage driven verification is concerned with the best approach for designing and implementing the verification project. Both approaches overlap each other because an assertion can be considered as a coverage point for the coverage analysis. The following subsections briefly describe these two methodologies.

4.4.1 Assertion-based Verification

In this verification methodology, assertions will be used as an integral part of the functional verification flow. The main components in this methodology are as follow:

(i) Identifying main properties to be asserted.
(ii) Deciding when those properties must be asserted.
(iii) Verification tools used to confirm asserted properties.

The main categories of properties that must be verified are as below:

(i) Operating environment assumptions.
(ii) Verification related assumptions.
(iii) Design specifications.
(iv) Design and implementation decisions.

It is not necessary that all properties must be satisfied at all time during device operation e.g., any device property may fail during the reset sequence. Therefore, such properties may not be asserted during the reset sequence.

4.4.2 Coverage-driven Verification(CDV)

CDV is a simulation-based verification approach particularly developed to focus on the productivity and efficiency related challenges faced in any functional verification. The coverage-driven approach improves the verification completeness and correctness. The basic idea behind this approach is the random generation of the stimulus, which is the main source of the productivity gained in this methodology. The coverage collection is a necessary part when the stimulus generation is randomized. As in the absence of coverage no information is available about scenarios covered. Some examples of CDV approaches are listed below.

1) *Transaction-driven verification:* It allows scenarios to be specified at a higher level of abstraction.
2) *Constrained random stimulus generation:* It leads to productivity gains in generating the scenarios.
3) *Automatic result checking:* It provides confidence that the design works for all randomly generated scenarios.
4) *Coverage collection:* It is a mandatory approach as in the absence of coverage it is not obvious which scenarios have been randomly generated.
5) *A directed-test-based verification:* It is also a necessary approach because not all scenarios can be generated efficiently by only using random generation techniques.

4.5 Verification Environment

The verification environment has to be implemented in a way that it should allow all scenarios in the verification plan to be verified according to the guideline of the target verification methodology. A verification plan includes identification of all factors that relate to device execution, preparation of planning sessions and planning documents, product functionalities documentation, structuring the verification plan, capturing features and attributes, and formulation of the verification environment and the coverage implementation. Generally, there can be different verification environment architectures available to achieve this target. This section briefly discusses a verification environment architecture that facilitates the application of the CDV methodology and the assertion-based verification methodology. The Open Verification Methodology (OVM), which provides the best outline to accomplish a CDV, is also discussed in this section. This section emphasizes on the

architectural blocks of a verification environment, how these blocks are generally used in the verification environment and the features that should be supported by each block.

4.5.1 Abstract View of a Verification Environment

A verification environment is connected to a DUV through the boundary signals of that DUV (Figure 4.5). The boundary signals can be grouped into interfaces that are comprised of multiple ports. Each port represents interconnected signals that jointly describes an interface protocol supported by the DUV. In this way, a DUV will be viewed as a block with a number of abstract interfaces suggesting a layered architecture for its verification environment. The figure below shows a layered architecture of a verification environment in which the lowest layer components directly interact with DUV interfaces.

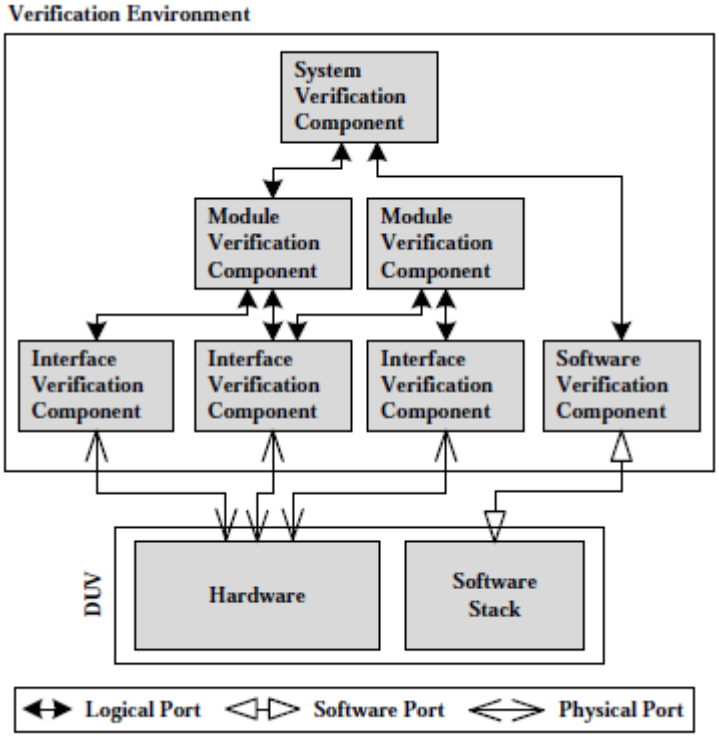

Figure 4.5: Verification Environment

Each higher layer component deals with increasingly higher levels of verification abstraction that correspond to more complex verification scenarios. This verification environment is structurally comprised of interface verification components (IVCs) and system/module verification components (SVC/MVCs). The IVCs provide abstraction for physical ports to interact with the DUV. The SVCs/MVCs make use of this feature to interact with the DUV at the level of abstraction provided by the IVCs. In this architecture, software verification components are a specific type of IVCs that interact with the software stack of the DUV. There are two operational modes for every verification component, as given below:

1) Active mode and

2) Passive mode.

A SVC in an active mode generates transactions for lower layer verification components while an IVC in active operational mode generates transactions at DUV ports. A passive verification component does not include any stimulus generation capability. It only monitors the verification environment traffic. These modes must be correctly implemented when a verification component is reused in the next design integration step.

4.5.2 Interface Verification Component (IVC)

The IVC (Figure 4.6) is used to interact with one or multiple DUV ports that support the same protocol. The IVCs also include supplementary features to monitor and collect coverage information of the physical port they interact with and hence suitable for performance analysis of DTP-XBUS-2. The architecture of an IVC is geared less towards generating full verification scenarios since concurrent interaction with multiple ports is required for this purpose. However, this architecture is more equipped to give an abstract view of DUV ports to higher layer verification components. They monitor the traffic on DUV ports by protocol checking and coverage collection.

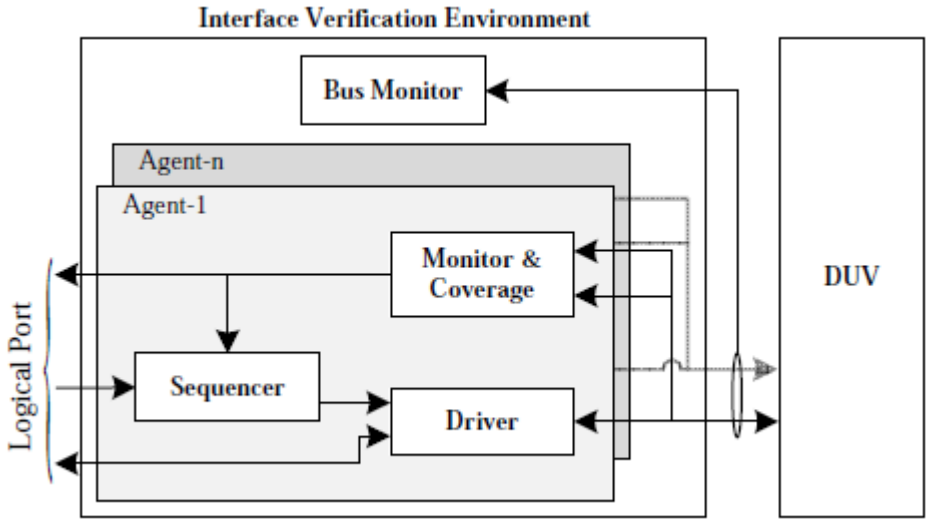

Figure 4.6: Interface Verification Component

The figure shows the architecture of an IVC that contains an agent components and a bus monitor. Each IVC interacts with a DUV port through an agent component that again includes following components:

1) A driver.
2) A monitor.
3) A sequencer.

4.5.3 Module/System Verification Component

A three-layer verification environment is shown in Figure 4.5, which is composed of IVCs, MVCs and SVCs. The SVCs include system level set-up generation functionality and perform end-to-end validation. The internal architecture of MVCs and SVCs is similar because they both interact with higher and lower layer verification components. The architecture of IVCs is different since they interact directly with the DUV ports. The SVCs generally emphasize on the end-to-end behaviour of the DUV rather than the behaviour of the individual blocks. In this approach it is assumed that smaller blocks have already been verified.

SVC emphasizes on:

1) Design errors in modules that can be verified only as a part of the overall system.
2) Inaccurate assumptions about the module operation.
3) Misconnection between system modules.
4) Errors in module interactions arising from protocol mismatches.

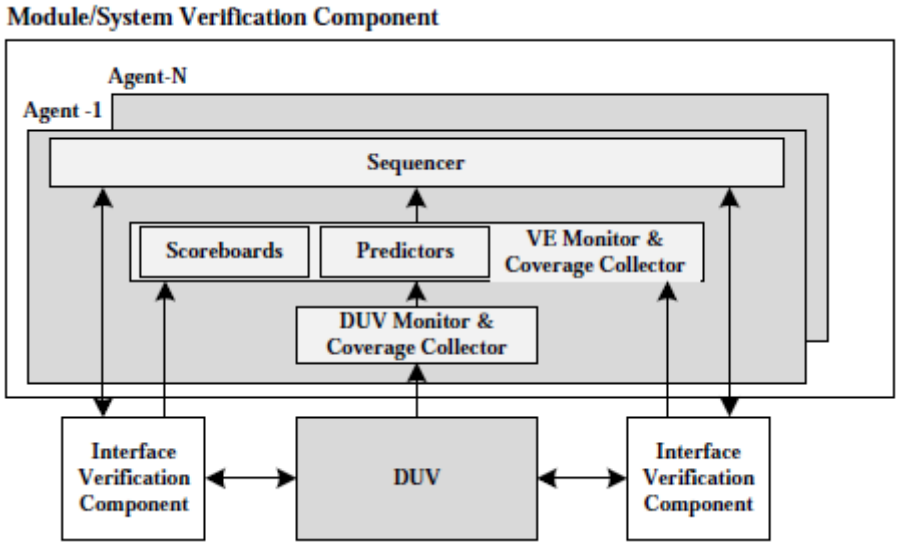

Figure 4.7: Module/System Verification Component

Figure 4.7 shows the architecture of a SVC containing multiple agents where each agent provides the same functionality while interacting with a different set of lower layer verification component. Each SVC includes the sequencer, the verification environment (VE) monitor and coverage collector, and the DUV monitor and coverage collector. To provide information about the current state of the DUV, the *VE monitor* interacts with monitors in the lower layer verification components. For example, system monitors track the monitors in the IVSs and in the MVCs. Since internal signals of the DUV cannot be tracked

through monitors attached to the DUV ports, a DUV monitor is used to track these internal signals. However, a thin layer of a wrapper between the DUV monitor and the DUV enables the reusability of the verification environment. A combination of both monitors (the VE monitor and the DUV monitor) allows a gray-box verification approach. A sequencer uses the information provided by these monitors to generate end-to-end scenarios. In an SVC, the sequencer is generally responsible for operations including the initialization of the DUV and the verification environment, the configuration of the DUV and the verification environment, and end-to-end scenario generation for the DUV verification. The *score boarding* is used to check for potential problems including:

1) Data values being different than expected.
2) Packets being received when not expected.
3) A packets not being received when expected.

The *coverage collection* is an SVC that focuses on collecting information including the basic traffic of each interface, the combined effective traffic at all interfaces, the states of the internal design, the generated sequences, delay and throughput information (performance information), the configuration modes, resets and restarts, and errors observed and errors injected.

4.6 Conclusion

This chapter presents the possible verification methodologies for DTP-XBUS-2. These methods are divided into two broad groups: Functional Test and Structural Test. Function test performs device emulation and verifies the VLSI chip when in operation. Functional test is divided further according to the abstraction of VLSI chip architecture description: black-box, grey-box and white box. In the black-box approach, the internal chip architecture is completely ignored. This method verifies basic transfer functionality of the DTP-XBUS-2. The grey box approach enables the internal FSM of DTP-XBUS-2 to be monitored. White-box approach enables full customization of the DTP-XBUS-2 registers and control. On the other hand, structural test has a more specific focus on VLSI chip interconnection. Structural tests are carried out through internal DFT logic in which test vectors are shifted in through a scan-chain. Also discussed in this chapter is the verification environment: IVC and MVC. IVC customizes the test interface according to the accepted communication protocol by the DUV. MVC comprises the test programs to be executed on the DUV.

Chapter 5: DTP-XBUS-2 Verification

Due to the imperfect nature of manufacturing process, defects may be introduced during fabrication, resulting in chips that could potentially malfunction. The designed chip architecture may not be suitable for a particular manufacturing process due to on-chip timing constraint (Inter-Symbol Interference or ISI). This chapter provides insights about the DTP-XBUS-2 verification plan and implementation. The objective of verification plan or test generation is the attempt to produce a set of test vectors that will uncover any defects in the chip (Figure 5.1).

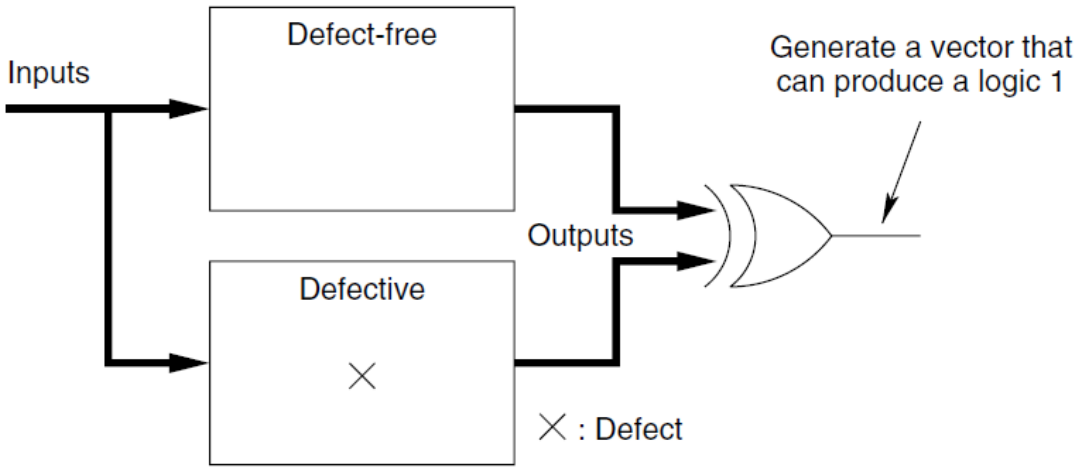

Figure 5.1: Test Generation

5.1 Memory System Verification

With the advent of deep submicron semiconductor manufacturing technology, embedded memory has become an attractive solution. Because almost all system chips contain some types of embedded memory, memories are considered one of the most universal cores. There are many challenges in regard to memory integration with logic despite the process technology issues, guaranteeing the performance, quality and reliability is yet another question to address. Testing embedded memory is more difficult than discrete memory due to the fact that accessing memory unit from external test is costly and might incur performance penalty and signal integrity issues due to pin/area overhead. DTP-XBUS-2 is incorporated with design-for-testability (DFT) logics for core isolation and tester access. While exploring various test methodologies available, we have also explored the possibility of Algorithmic Built-in-Self-Test (AGBIST). The figure below (Figure 5.2) shows the current implementation of BIST [38].

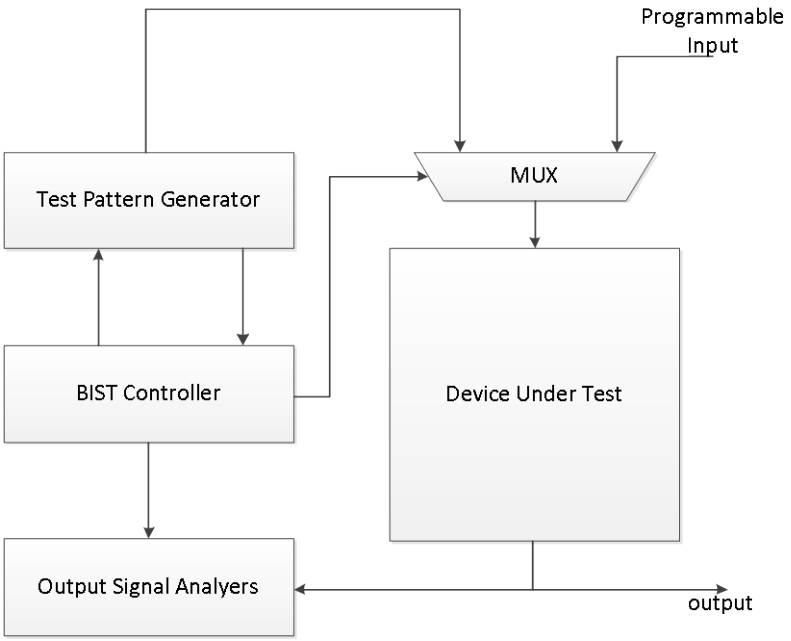

Figure 5.2: Built-in Self Test (BIST)

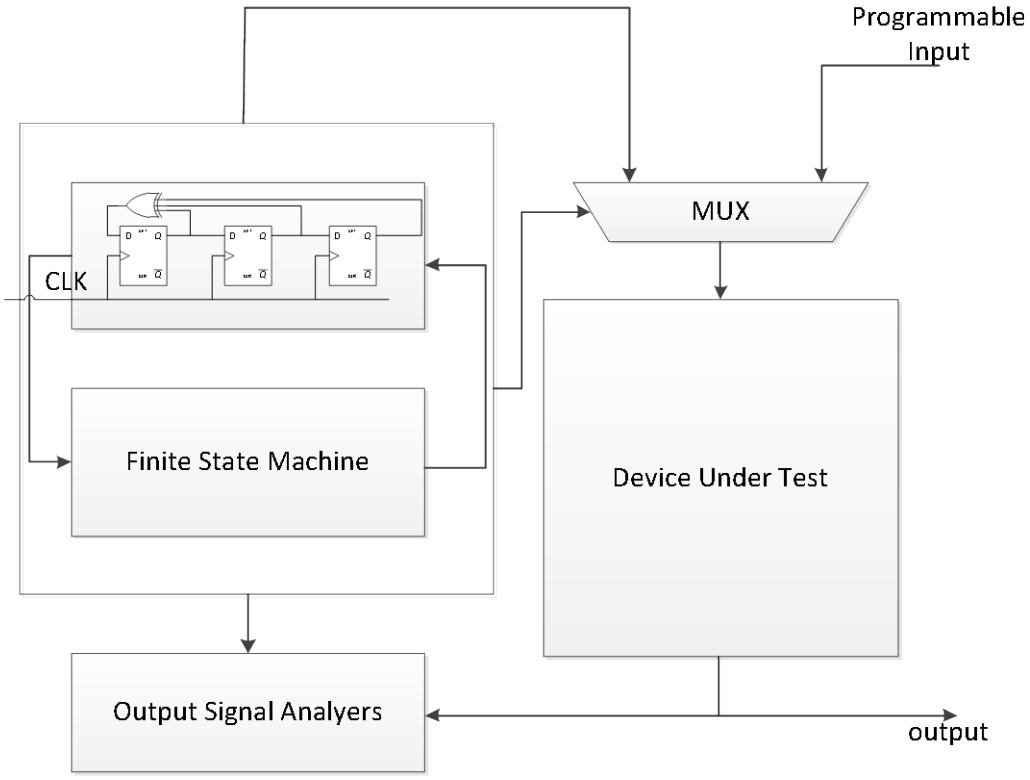

Figure 5.3: Algorithmic Built-in-Self-Test (AGBIST)

The main difference between AGBIST (Figure 5.3) and BIST is that AGBIST algorithmically generates a state dependent test vector which is suitable for state machine tests and test vector compression. BIST uses Linear Feedback Shift Register (LFSR) to generate test vectors which is sufficient for combinational logic tests.

A complete treatment of AGBIST will be a new topic of research itself. Hence the discussion of AGBIST will be omitted in this thesis.

The DTP-XBUS-2 memory systems consist of embedded RAM and ROM that interact with other sub-systems through the processor local bus. The plan of verification includes the following key features.

1) An interface structure to interact with the memory unit
2) Development of a bus functional model complying with the processor local bus specifications.
3) Test library/test cases development.
4) Test bench development.

5.2 Interfacing with the Memory

The interface structure is the front-end of the test bench that interacts with the memory unit. This structure is used for:

1) Modeling the communication between functional blocks
2) Structural connectivity between IP blocks

The implemented bus functional model (BFM) complements the interface structure by replicating the behavioural model of the local bus master-slave components. The BFM includes an interface for cycle based communication with the memory system. This model emulates the following communication cycle.

1) Idle cycle
2) Read request
3) Write request
4) Burst read
5) Burst write

The BFM drives the signal to the device under validation and the responses are captured through the interface structure that complies with the local bus specifications.

The test library is the test program that contains several test cases for the functional verification of the memory system. The BFM is used to execute the test program which includes the following:

1) Sequential write/read test

a. Generates a sequential address, randomizes data items and sends the write request to the device with a single beat write request.
b. After successful completion of the write request, a read request is sent for the same address with a single beat read request.
c. The received data is compared with the written data. If the test passes, the whole procedure is repeated for the next sequential address.

2) Random single beat write/read test
   a. Adopts the same methodology as the sequential write/read test except that it generates random address instead of sequential.
   b. This test is used to test real-time scenarios where memory accesses are random.
3) Random burst write/read test
   a. This test randomly generates arrays of addresses and data, and sends a burst write request to the device.
   b. After successful completion of burst write, this test sends a burst read request to the device under validation.
   c. The received data array is compared with the written data array. If the test passes, the whole process is repeated again with new set of randomized arrays of addresses and data.

The test bench instantiates all components and connects them together. In addition, the test bench also drives the system clock and system reset signals. The test bench includes the following:

1) Memory unit
2) Processor local bus (PLB) interface
3) Bus functional model
4) Test library

The figure below (Figure 5.4) shows the architecture of the memory test bench.

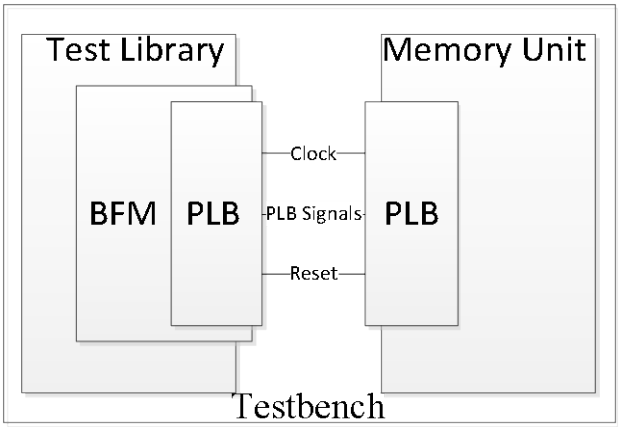

Figure 5.4: Memory Test bench

5.3 DTP-XBUS-2 Functional Verification

The DTP-XBUS-2 verification environment includes:

1) A configurable test library, enhanced on existing RAM test library
2) Coverage model
3) Test bench

5.3.1 The Library

The DTP-XBUS-2 system has both processor local bus master and slave interfaces. Hence, the interface structure will connect the master and slave components to its interfaces respectively. As the test library (initiator) initializes the Instruction Transfer Mode (DTP-TXI) to drive different tests on the DUV, the library will be configured for a particular mode of operation. This test library (initiator) is an extension of the memory test library which is capable of driving only a single BFM and of generating the addresses for a complete given address space.

In order to initiate the DTP-XBUS-2 transfer operation, the XBUS-2 must be configured correctly. To access a slave component through the XBUS-2, the transferring end from external device has to transmit an address that qualifies the address space of this slave component. As the test library is responsible of generating the addresses for the BFMs, thence it must be configurable so that it can generate the addresses within a specific sub-space of an address space (Figure 5.5). Hence, this test library (initiator) is required to divide the total address space of the Sub-bus system into four sub-spaces. The addresses generated for a slave component include the slave-id in the MSBs (configurable). The address space of each connected slave component is divided into three sub-spaces, one for each BFM. A BFM can only access this particular address space inside the memories. This subdivision of the slaves' address spaces is necessary to handle the overlapping problem.

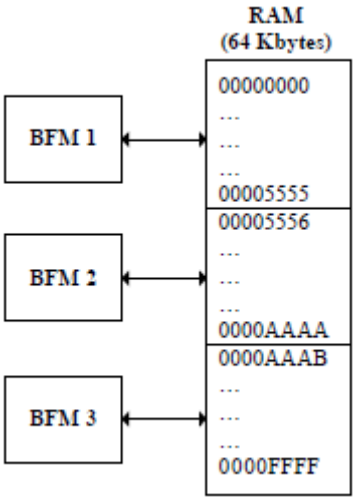

Figure 5.5: Memory partition

Three instances of the test library will be required, with each configured to drive the respective BFMs. Each instance drives a single BFM and generates all addresses within the accessible address range of its BFM. However, each BFM can randomly access the connected slave components over the XBUS system. Each instance of the test library (initiator) can execute all tests which are provided by the test library.

5.3.2 Coverage Model

A *coverage model* is used to ensure that the DUV has been exposed to a satisfactory variety of the stimuli and it is functioning correctly. A database of Verilog bins has been created to store a histogram of the addresses accessed by each BFM. The model covers the requested addresses by the BFMs and the slaves. In this way, the number of times in which an address is accessed by a BFM and how many times a Slave component correctly responded the requests for this address could be cross verified.

5.3.3 Test Bench

This test bench (SPC_TB) is used for the functional verification of the Sub-bus system. It instantiates all components those are required for the verification, correctly configures them, connects them together, and drives the system clock and the system reset signals to these components. The including components are

1) A XBUS-2 system,
2) Four RAM components,
3) Three interfaces,
4) Three bus functional models,
5) Three test libraries (one for each BFM)

Figure 5.6 shows the architectural look of the test bench that was used for the functional verification of the XBUS-2 system.

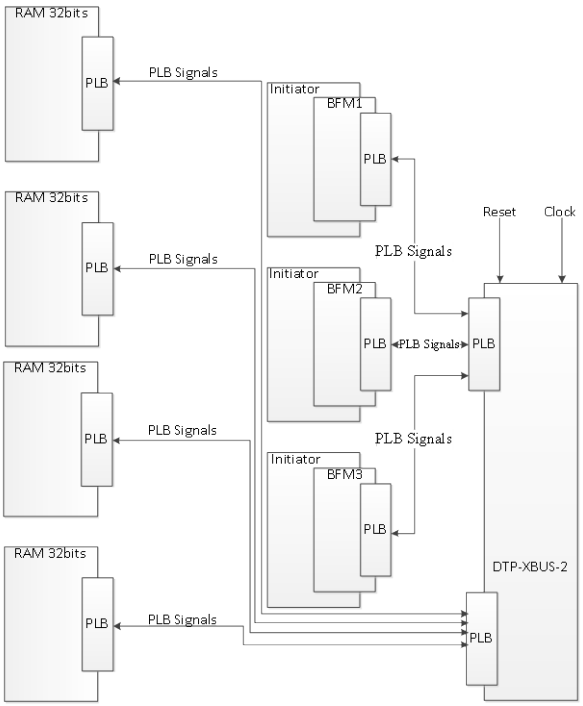

Figure 5.6: XBUS-2/Sub-bus Test bench

5.4 SPARC V9 Functional Verification

The functional verification of a heavily pipelined processor is a challenging task. The constrained random generation methodology is used in the verification of the SPARC V9 core. A grey-box verification approach is used for more complete test coverage. Therefore, the internal signals of the SPARC V9 core will be monitored along with a *reference model* for the comparison. The architectural model (simICS) of the SPARC V9 core was used as the golden model. As the verification environment uses Verilog interfaces to communicate with the DUV (DTP-XBUS-2), a *Verilog wrapper* was implemented around the SPARC V9 core. This wrapper provides interfaces to access the DUV.

Since the SPARC V9 core is a complex implementation and its verification is a challenge, hence the most important aspects to be verified must be identified. These aspects play a vital role in the correct execution of this processor. The correct working of these aspects verifies that the core is correctly operational. As in literature [36], SPARC V9 is being continuously verified and proven to be stable. This alleviates the SPARC V9 verification requirements. The aspects taken into account are listed below.

1) SPARC V9 always generates a correct program counter (PC).

2) SPARC V9 correctly updates its state in its supervision register (SR).
3) SPARC V9 correctly saves its context in case of an exception (ESR/EEAR/EPCR).
4) SPARC V9 always stores correct data to corresponding addresses in the data memory.
5) SPARC V9 correctly stores the execution results in its general purpose registers.

To identify the correct time interval to monitor the DUV's features would require a thorough understanding of the core's architecture, in particular the instruction pipeline execution. This task becomes more complex when the exception model and the variable execution time of different instructions are taken into account. Jumps/branches and delay slot executions need to be handled appropriately as well. Another important side is to consider the freeze logic and flush-pipeline logic of the SPARC V9 core. These two logics vigorously control the processor pipeline execution. This information is contained in registers. Hence, they all have enable signals for their update. These register enable signals identify the correct points to monitor these registers. However, along with these enable signals to manage the pipeline's control logic, the exception control logic, the freeze logic and the flush-pipeline logic will also be required since these logics control the register enables. In pipeline execution, different pipeline stages may operate on different registers or may operate on different parts of a single register. Thus, identifying a correct execution stage to monitor a register is a very important aspect.

A robust verification environment will be required in order to feed the instructions into the SPARC V9 core, handle Load/Store requests from the core, and correctly monitor the important registers of the core. Additionally, it is very important for an exhaustive verification to fill the complete instruction pipeline of the core and account the dependencies between the instructions. In this verification environment, an instruction is first sent to the golden model (simICS). After its execution the status of the golden model is obtained and stored. This instruction is then sent to the DUV and all important registers are monitored when this instruction updates them in different pipeline stages. These registers' values are compared with the status which was received from the model. The model executes every instruction in zero time while the SPARC V9 core is hardware implementation (can be registered) having eight instructions in its pipeline. Hence, this verification environment must include a synchronization mechanism between the golden model and the DUV.

5.4.1 Instruction Set Simulation

The Instruction Set Simulator (simICS) of the SPARC V9 core is used as a reference model for the functional verification. This simICS is an architectural simulator, and a generic simulator capable of emulating SPARC V9 architecture based systems. It provides high-level and quick architectural simulation for early code analysis and performance analysis of systems. It supports most peripherals and system controller cores. The current version of the simICS provides a network socket for remote debugging with a GNU debugger (GDB) support for different environments (OAK processor model, memory configurations and sizes, configuration of peripheral devices). This version also offers the choice to either use the simulator standalone or as a library. The new version also includes an Open SystemC Initiative (OSCI) Transaction Level Modelling (TLM) 2.0 interface. Its standard configuration can model the main memory, the CPU, and a numbers of other peripherals. The existing ISS was embedded into a SystemC module in order to use the simICS as a golden model. The module is also required to support the Programming Language Interface (PLI) to incorporate the Verilog based verification environment. The development involves several steps, as given below.

1) Modify the existing Oak (simICS) library to provide a set of public interfaces to access it.
2) Define a SystemC module as a wrapper around this library that can access its public interfaces.
3) Implement the PLI support inside this SystemC wrapper so that it can be integrated within the Verilog based verification environment.

5.4.2 Compiling simICS Library

The installation of the SPARC V9 GNU tool-chain package includes the simICS simulator as standalone for an early code analysis and a performance analysis of the system. In order to make use of this simulator as a reference model, the set public interfaces need to be compiled to a library. This library can be configured to model DTP-XBUS-2 system. The simICS library needs to be configured to model only the CPU and some generic peripherals. It does not model the main memory, the cache system, the memory management or other peripherals, since the DUV does not include such components. Similar to the standalone implementation of the simulator, a configuration file has to be used to configure the library to model different components in the system. The simICS offers the facility to use itself as a library, and provides two upcall functions to call up to the SystemC model of which it is a part, to read and write from the peripheral address space. An additional upcall function needs to be implemented in order to access the status of the ISS. Further, a PLI needs to be

implemented in order to access this library within the Verilog-based verification environment.

5.4.3 Using simICS as a Library

In the standalone implementation of the simICS (Figure 5.7), the main function initializes the instruction set simulator. After that it stays in a loop and executes the instructions. This is similar to creating a new simICS workspace for SPARC V9 simulation. However, in the library implementation this main function is replaced by a series of functions those form the interfaces to the library. The header file (config.h) contains the declaration of these functions while their implementation is provided in the libtoplevel.c file. These functions are described below.

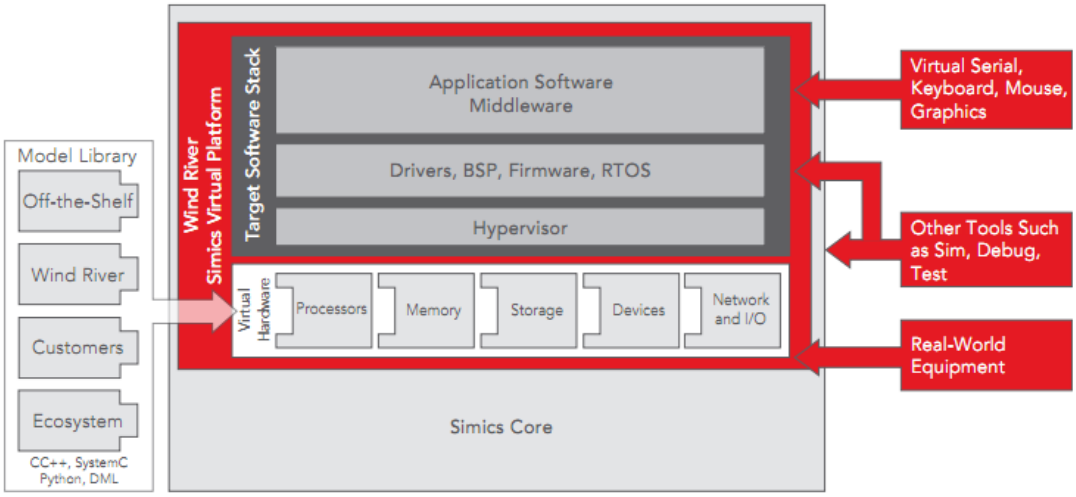

Figure 5.7: simICS

1) **simICS_init (...)**

   This function initializes the simulator. It has several arguments those are given below. *Config_file:* This file provides the configuration data to the simulator.

   Image_file: This argument is used to pass the program image to the simICS. By default, the simICS takes the .ELF executable format of program images. However, it can also take the .IHex executable format. Since we want to fetch instructions and data from external test bench, an empty .ELF image will be passed in this argument. To read or write from the peripheral address space the ISS needs to be able to call up to the SystemC model of which it is part. A standard implementation of the ISS library provides two upcall functions to read and write from the peripheral address spaces. These functions are defined by the **upr** and **upw**, fourth and fifth function parameters of the simICS_init(). The upcall functions in the golden model have been

modified according to requirement. In this implementation the ISS uses the "upr" upcall function to read the next instruction from the SystemC model. If this is a Load instruction the same upcall function is used again to read data. However, in case of a Store instruction the "upw" upcall function is used to write data up to the SystemC model. Since it is required that the internal status of the ISS to be accessible (after every instruction's execution), a third upcall function (**upcpustatus**) is implemented in the simICS to write its status up to the SystemC model. This ISS status includes:

(i) The PC register,
(ii) The supervision register (SR),
(iii) The exception supervision register (ESR),
(iv) The exception program counter register (EPCR),
(v) The exception effective address register (EEAR),
(vi) All general purpose registers (GPRs) and
(vii) The instruction that was just executed on the ISS.

The implementation of these upcall functions is provided in the SystemC model (C++), while the simICS (C) can access them on demand. Therefore, upcalls were implemented as *static functions* in the SystemC model. The SystemC model calls the simICS_init(). To enable the upcall functions for invoking the member functions of this SystemC model a pointer (class_ptr) to this SystemC module instance is passed as an argument to these upcall functions. Third argument (class_ptr) is the pointer to the SystemC module class that initializes the simulator by calling the simICS_init().

2) **simICS_run (...)**

This function is used to run the simulator for a specific period of time, passed in its argument (in seconds). The duration of -1 runs the simulator forever. **int** simICS_run (**double** duration) ;

5.4.4 simICS Library and Generic peripherals

The library implementation of the simICS makes provision for any additional peripheral to be implemented externally. Any access (READ/WRITE) to this peripheral's memory map generates the upcall to an external handler. **Generic** (Figure 5.8) is a new extension in the simICS to model external peripherals. Any READ or WRITE access to the memory map of an implemented generic component generates an upcall. All peripherals of the simICS are configured in a configuration file (*.cfg*). A new section generic is introduced in this file to describe the external peripherals. Multiple external peripherals can be described by multiple generic sections. Each generic section includes multiple parameters to specify an external peripheral.

```
section generic
    enabled          =          1
    baseaddr         = 0x00000000
    size             = 0x7FFFFFFF
    byte_enabled     =          1
    hw_enabled       =          1
    word_enabled     =          1
    name             = "Gen_dev1"
end
```

Figure 5.8: Generic

The parameters of a **generic** component are as given below.

1) enabled = 0|1

   The option 1 is to enable and the option 0 is to disable this AT Attachment and AT Attachment Packet Interface (ATA/ATAPI). If you do not specify the value, default is 1 (enabled).

2) baseaddr = value

   It is the starting address of this generic peripheral's memory map. Its default value is 0 (not a sensible value). The size of the memory mapped register space is controlled with a parameter i.e., size. It is described below.

3) size = value

   This parameter controls the size of the generic peripheral's memory mapped space in bytes. Any access (READ/WRITE) from the ISS to this address space (baseaddr 7! size-1) will be directed to the external interface (upcall). The value of this parameter should be in power of 2.

4) name = "str"

   This string specifies the name of the generic peripheral.

5) The generic peripheral can be configured to have support for byte, half-word and word accesses. If the value is 1 (default) the respective support is enabled.

   byte_enabled = 0|1

   hw_enabled = 0|1

   word_enabled = 0|1

The requirement for the golden model is to generate the upcalls for a complete 32 bit address space (0x0000_0000 to 0xFFFF_FFFF) which is byte-addressable. The maximum size that can be supported by a single generic peripheral is 0x7FFF_FFFF bytes. Hence, three generic peripherals are needed to cover the complete 32 bit address space. With this configuration the golden model (simICS) always generates the upcalls either to READ/WRITE data or to fetch a new instruction. The verification environment feeds the instructions and data to the simICS.

5.5 SystemC Wrapper and Reference Model

After modifying the simICS (or ISS) and generating the library, a SystemC wrapper around this library needs to be implemented in order to incorporate the reference model in the verification environment. The key features this SystemC wrapper was required to implement are as given below.

1) Provide the implementation of the upcall functions (upr, upw, upcpustatus).
2) Call the simICS_init() function and pass its arguments.
3) Run the simulator forever by calling the simICS_run() function in a thread.
4) Provide a PLI for these upcall functions to be accessible in the verification environment.
5) Implement a synchronization mechanism between the SystemC upcalls and the PLI functions.
6) Handle the host machine's byte order (little-endian/big-endian).
7) Provide the implementation to qualify valid data bytes inside the data array by using the selection bits.
8) Parse out the required status information of the ISS and make it available to the PLI functions.

5.5.1 Upcalls

Three static member functions must be implemented in the SystemC wrapper in order to provide the implementation of the upcall functions of the simICS library. These static functions take a pointer of the SystemC module's instance which starts the simICS ISS. This pointer is provided as a third argument to the simICS_int function. Each static function calls inside another C++ class member function. This member function actually provides the implementation of its respective upcall function. When the ISS generates an upcall to its corresponding interface function, it simply calls this static function because its interface function is a pointer to a C++ static function. This static function calls a member function which actually implements the upcall. The code snippet below is taken from the

implementation of this SystemC wrapper class. It gives insight about the upcalls' working in the wrapper. The *staticWriteUpCPUStatus* is a static function of the wrapper. Its pointer is passed into the upcall i.e., upcpustatus (in the simICS_init) for writing up the ISS status. When the ISS generates this upcall, since the upcpustatus is a pointer to the staticWriteUpCPUStatus function, the ISS in fact calls this static function which actually calls a C++ member function inside (writeUpCPUStatus). This member function parses the incoming status information of the ISS and makes it available to its respective PLI function.

```
/ *===Access the CPU state after every instruction's exectution===* /
void spc_sc_module_pli::staticWriteUpCPUStatus(void*instancePtr, void*cpu_statusPtr)
{
        spc_sc_module_pli* classPtr = ( spc_sc_module_dpi*) instancePtr;
        cpu_state_up* cpu_state_up_ptr = ( cpu_state_up*) cpu_statusPtr;
        classPtr−>writeUpCPUStatus ( cpu_state_up_ptr ) ;
} / / staticWriteUpCPUStatus( )
```

The declaration of the staticWriteUpCPUStatus function is given below.

```
static void staticWr iteUpCPUStatus ( void * instancePtr , void * cpu_statusPtr ) ;
```

simICS_init Initialization

The simICS_init library function is called within the SystemC wrapper to initialize the ISS. A configuration file, an empty ELF file, the wrapper's own pointer (itself starting the ISS) and the pointers to its static functions are passed as arguments to this library function, as given below.

```
simICS_init("../example.cfg", "../example.elf", this, staticReadUpcall, staticWriteUpcall, staticWriteUpCPUStatus) ;
```

5.6 Programming Language Interface

As discussed, the implementation is provided in a SystemC model and it is imported inside a Verilog model by using the **include "PLI"** declaration. On the other hand, the implementation of is provided in a Verilog model and it is then exported to a SystemC model. In this verification environment only the PLI functions need to be imported in order to be called within a Verilog based test bench while their implementation is provided inside the SystemC wrapper of the golden model. Three imported PLI functions have to be

implemented in the SystemC wrapper respective to three upcall functions. The hooked-up *member functions* of the SystemC wrapper take data and instructions from these imported PLI functions and feed the ISS with this data and instructions. These member functions also make the simulator's status and data available to these imported PLI function so that it can be sent to the test bench. The definition of these three imported PLI functions in the SystemC wrapper is given below (Figure 5.9).

```
int sv_readUp(const int rinsn, const int rdata, int *read_addr, int *read_addr_mask);
int sv_writeUp(int *waddr, int *wdata);
int sv_writeStatusUp( cpu_state_ref *iss_status);
```

Figure 5.9: PLI functions

All included PLI functions must be registered in the SystemC module by using SC_PLI_REGISTER_CPP_MEMBER_FUNCTION().

SC_PLI_REGISTER_CPP_MEMBER_FUNCTION("sv_readUp", &spc_sc_module_pli::v_readUp);

SC_DPI_REGISTER_CPP_MEMBER_FUNCTION("sv_writeUp",&spc_sc_module_pli::v_writeUp);

SC_DPI_REGISTER_CPP_MEMBER_FUNCTION("sv_writeStatusUp",&spc_sc_module_pli::v_writeStatusUp);

The PLI identifies an imported function by its name only (not by its parameters). Hence, only one copy of overloaded functions can be supported.

Note: The composite data types (e.g., structure/union) being transferred through the PLI from SystemC to Verilog (or opposite) make provision for each element to be 32-bit aligned. For example, if a structure contains a char data type (8 bits), 24 bits should be padded to it to make it 32-bit aligned.

5.6.1 Golden Model Synchronization

When the ISS starts the execution it fetches the first instruction through an upcall function (upr) from the reset address (0x0000_0100). It executes the instruction in zero time and comes up again to fetch the next instruction. As the ISS is running forever in a SystemC thread, it will never give the control to any other process if there is no mechanism to block it. A SystemC FIFO was implemented based mechanism with blocking READ/WRITE to synchronize the system. Four FIFOs of a single element depth were implemented between

the hooked-up member functions and the imported PLI functions. With this strategy, when the ISS upcalls to fetch a new instruction it writes the PC address to the pc-fifo and is blocked until the instruction is available in the read-fifo. If this instruction is a Store, the ISS makes an upcall to write data up and it is blocked until the write-fifo is empty. However, if this instruction is a Load, the ISS upcalls to read data and it is blocked until data is present in the read-fifo. After completing the execution of an instruction the ISS upcalls to write its status up and it is blocked until the status-fifo is empty. When the ISS is blocked the control is transferred to other running processes. On the other ends of these FIFOs the test bench uses the imported PLI functions to feed the instructions and data to the ISS to read data and addresses (for the Store instructions) and to get the status of the ISS after the execution of every instruction.

5.6.2 Golden Model Architecture

Figure 5.10 shows the architecture of the *golden model*. The ISS accesses the wrapper functions through its upcalls. The communication between the ISS and the test bench is synchronized by means of SystemC FIFOs. Test bench implemented in Verilog (OVM) accesses these FIFOs through the imported DPI functions.

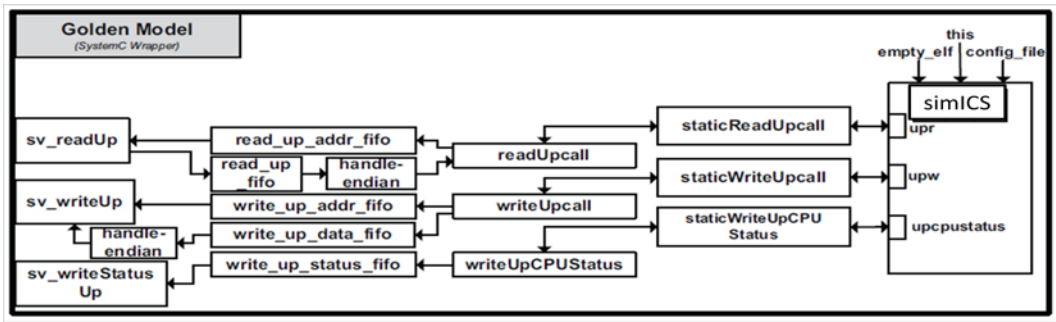

Figure 5.10: SPARC V9 Golden Model

5.7 Verilog Wrapper and SPARC V9 Core

A Verilog based wrapper has to be implemented around the SPARC V9 core (DUV) which includes three Verilog interfaces named as:

(i) The *insn-if.*

(ii) The *data-if* .

(iii) The *status-if.*

These are used to access the instruction XBUS-2, data XBUS-2 and the internal signals of the core respectively. The status-if of this wrapper makes all required internal signals of the DTP-XBUS-2 available at its ports. The internal signals include the status registers (to be monitored) and the control signals (to control the monitoring). The status registers include

(i) Some important SPRs,

(ii) All GPRs.

(iii) The program counters (PC).

This wrapper also implements a translation block to translate the SPARC V9's internal signals to a usable form e.g., the GPRs are implemented as a dual-port synchronous memory and their translation to thirty two 32-bit registers is needed. Further, this wrapper also implements a control block to manipulate the internal control signals according to the requirements e.g., delay a control signal for two clock cycles. All components of the verification environment interact with the DUV only through the wrapper's interfaces.

5.8 Verification Environment

Figure 5.11 elaborates the architecture of a verification environment (spc_tb_top) and was used for the functional verification of the SPARC V9 core. This verification environment includes

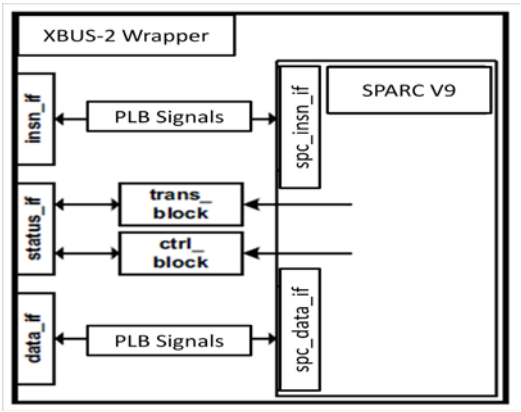

Figure 5.11: Verification Environment

1) The golden model,
2) The DUV wrapper (XBUS-2_wrapper) (Figure 5.9),
3) The main test bench component (spc_tb),
4) The global package (v_sc_package) and
5) The test library (spc_tb_test_example_inst).

The main test bench (spc_tb) is a reconfigurable and reusable component. It interacts with the golden model through its imported PLI functions and uses its physical interfaces to interact with the DUV wrapper. The main test bench executes the configurable tests of the test library where all tests are constrained random generation of the scenarios which are comprised of SPARC V9 instructions. spc_tb first sends an instruction to the golden model, writes/reads data (if the instruction is Load or a Store) and receives the ISS status once the instruction has been executed. Further, it sends this instruction to the DUV. While this instruction passes through different pipeline stages in the DUV the main test bench keeps eye on the state of the DUV and reacts accordingly. It examines the control state machine of the DUV along with the data-path. spc_tb monitors the control signals of the DUV to determine the right time to examine the status of the DUV (e.g., PC, SPRs, etc.) and the execution results (GPRs). It compares the status of the golden model with the DUV status and scoreboards it. The main test bench also implements a coverage model to assess the completeness of the verification. Most of the components of the verification environment can be configured according to implementation's requirements. For example,

(i) The coverage model or the scoreboard should be implemented or not,

(ii) An agent component will operate as a passive component,

(iii) Which tests of the test library to be executed.

The components of this verification environment will be described as below (Figure 5.12).

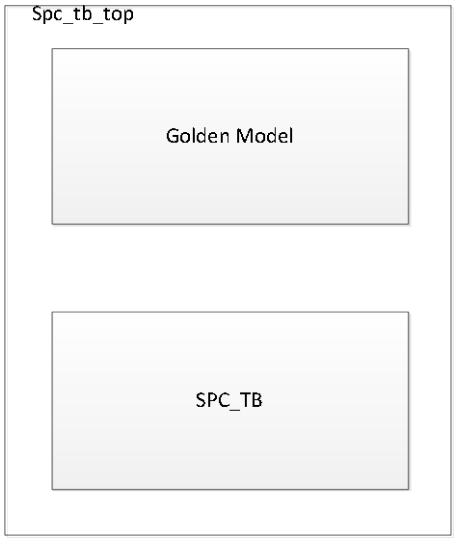

Figure 5.12: Verification Components

5.9 Main Test Bench for DTP-XBUS-2

Figure 5.13 shows the structural design of the main test bench used for the functional verification of the SPARC V9 core. It is comprised of three main components:

1) The interface verification component (ivc_spc),
2) The system/module verification component (svc_spc) and

3) The virtual sequencer (spc_virtual_sequencer).

All components inside the spc_tb interact with each other through standard TLM interfaces.

5.9.1 Interface Verification Component (IVC)

The main test bench interacts with the DUV (SPARC V9) through its interface verification component. This IVC includes

(i) Three physical interfaces (instruction, status, data),

(ii) An instruction agent,

(iii) A data agent and

(iv) A bus monitor.

The instruction, status and data interfaces of the IVC are respectively connected to the instruction, status and data interfaces of the DUV wrapper. The other side of the instruction, status and data interfaces are respectively connected to the instruction agent, the bus monitor and the data agent of the IVC. The instruction interface is used to send the instructions to the DUV. The status interface is used to read the internal status registers and the control signals of the DUV. The data interface is used to send or receive data of Load or Store accesses from the DUV. Figure below shows a detailed view of this IVC.

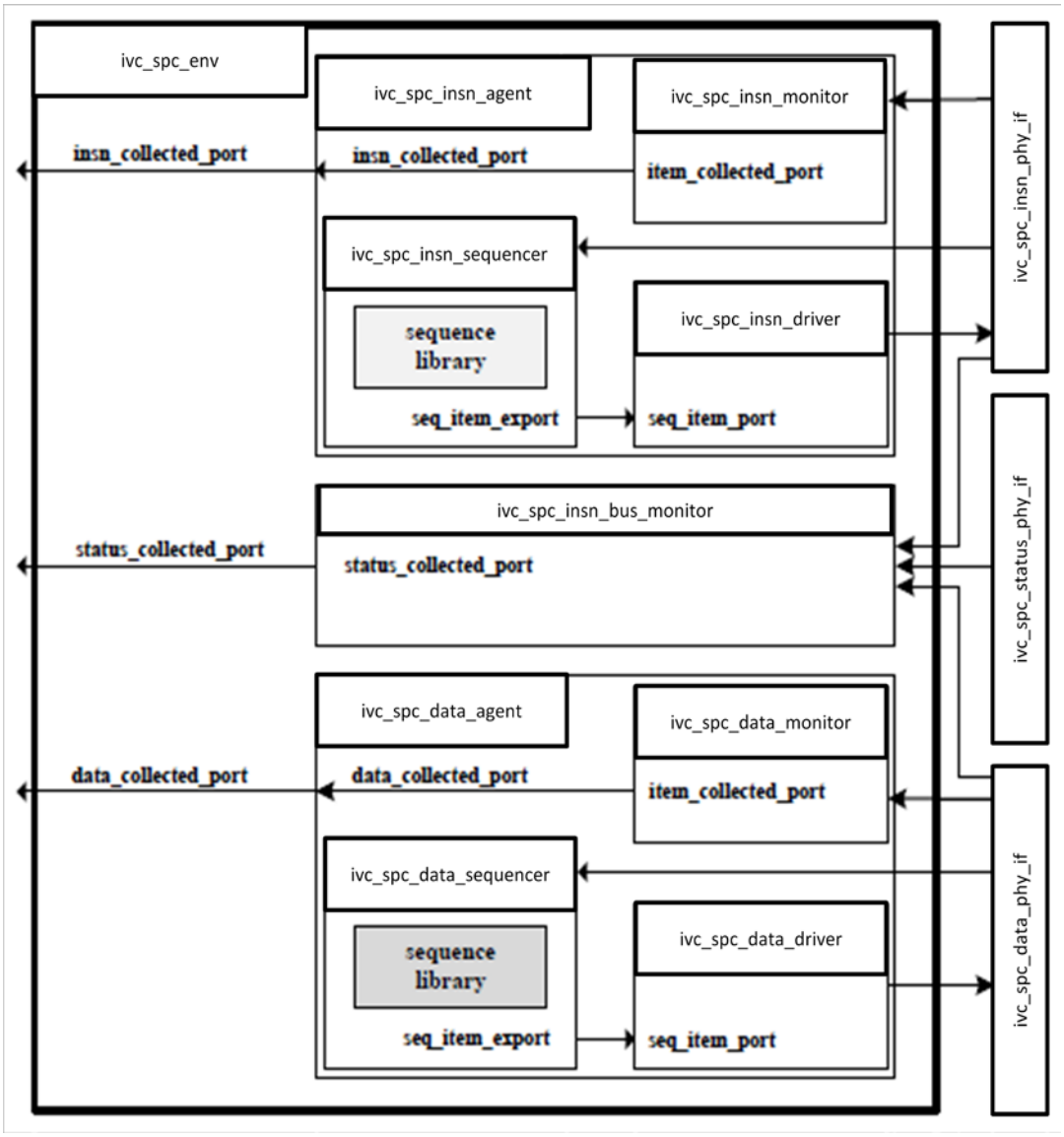

Figure 5.13: Interface Verification Component

5.9.2 Physical Interfaces

These interfaces provide the port-level connection to the DUV interfaces and the helper functions for the IVC to read or write the values on these ports. These interfaces implement the XBUS-2 protocol checking using *concurrent assertions* e.g., the *ack* and the *err* signals must not be asserted together. These *concurrent assertions* are checked throughout the simulation to ensure that the interconnection protocol is always obeyed.

5.9.3 Instruction Agent

This instruction agent contains

(i) An instruction driver,

(ii) An instruction monitor and

(iii) An instruction sequencer.

It operates as a Master component which is connected to the instruction interface of the DTP-XBUS-2. On receiving a request from the core, its instruction driver requests a new transaction (instruction) from the instruction sequencer and sends it to the DUV over the instruction interface (ivc_spc_insn_phy_if) by using its helper functions. These transactions are required to be translated to the port level signals. The instruction driver follows the XBUS-2 interconnection standard. It synchronously asserts the termination signal (i.e., ack, err, rty) for one clock cycle after each request from the DUV. The instruction monitor only reads (does not drive) the signals of the instruction interface when the instruction driver acknowledges a request.

After reading the interface signals by using helper functions, this instruction monitor translates them into an instruction transaction and sends this transaction to the *system verification component*, over a TLM port (insn_collected_port). An instruction transaction encloses the instruction that is sent to the DUV and the address of this instruction. The instruction driver requests a new instruction from the instruction sequencer. It sends the next transaction (instruction) in the sequence to the driver. These sequences are a constrained random generation of SPARC V9 instructions. The instruction sequencer contains a library which encloses several sequences of instructions those can be generated on demand.

5.9.4 Data Agent

The data agent contains

(i) A data driver,

(ii) A data monitor and

(iii) A data sequencer.

It operates as a Slave component which is connected to the data interface of the DTP-XBUS-2. On receiving a READ request from the DUV, its data driver requests a new transaction (a data item) from the data sequencer and sends it to the DUV over the data interface (ivc_spc_data_phy_if) by using its helper functions. These transactions are required to be translated to the port level signals. The data driver follows the XBUS-2 protocol. It asserts the synchronous termination signal (ack, err, rty) for READ requests while asserting asynchronous termination signal for WRITE requests. These termination signals are asserted for one clock cycle. The data monitor only reads (does not drive) the signals of the data interface when the data driver acknowledges a request. After reading the interface signals, it translates them to a data transaction and sends this transaction to the system verification

component over a TLM port (data_collected_port). This data transaction encloses the address and the data item along with the write enable (we_i) and the byte select (sel_i) XBUS-2 packets. On the data driver's request, the data sequencer sends a new transaction (a data item) to the driver. The data sequencer contains a library which encloses several data sequences.

5.9.5 Bus Monitor

The bus monitor is used to access the internal control signals and the status registers of the DUV through the status interface of the IVC. It can also access the instruction and data interfaces.

This bus monitor reads the XBUS-2 status signals every cycle, translates them to a status transaction and sends the transaction to the system verification component over a TLM port (status_collected_port). This status transaction is comprised of

(i) the PC register,
(ii) the SR,
(iii) the ESR,
(iv) the EPCR,
(v) the EEAR,
(vi) all GPRs and
(vii) some important control signals of the DTP-XBUS-2 e.g., pc_we, esr_we, except_start, etc.

5.10 System Verification Component (SVC)

The ultimate goal of the system verification component is to verify the end-to-end characteristics of the DTP-XBUS-2 core. This SVC is one step higher at abstraction level than the IVC. It is comprised of the following components:

1) The module monitor (mvc_monitor),
2) The scoreboard (mvc_scoreboard) and
3) The coverage model (mvc_coverage_model).

5.10.1 Module Monitor

This module monitor, shown in Figure below (Figure 5.14), collects the transactions (instruction/data/status) sent from the IVC. It interacts with the golden model to read its status and data along with the store address (in case of Stores). It accesses the golden model by accessing the PLI functions (sv_writeStatusUp, sv_writeUp) through the local Verilog tasks (sv_readstatusUp_t, sv_readUp_t) respectively. The golden model executes every

instruction in zero time while the DTP-XBUS-2 is an 8-stage pipeline processor. Therefore, a synchronization mechanism must be implemented to correctly compare their status and results. This mechanism is implemented in the module monitor using Verilog FIFOs where the depth of each FIFO is four elements. The module monitor receives information from the golden model and stores it into the corresponding FIFO (e.g., SR to SR-fifo, PC to PC-fifo). The main test bench keeps on sending the instructions to the ISS first and then to the DUV. The module monitor keeps on filling its FIFOs by receiving the status and results from the ISS. These FIFOs are full by the time the first instruction executes on the DUV (in the execution pipeline stage). The module monitor takes the status information of the ISS from the top of the FIFOs, parses out the status of the DUV from the transactions (status/data) received from the IVC and sends both information to the scoreboard. The control block (ctrl) implements interactive control logic to monitor the control state machine of the DUV and react accordingly to decide the right time of comparison between the ISS and the DUV statistics. This control block also sends a few control signals (e.g., except_start) to the *virtual sequencer* which are needed for the reactive scenario generation.

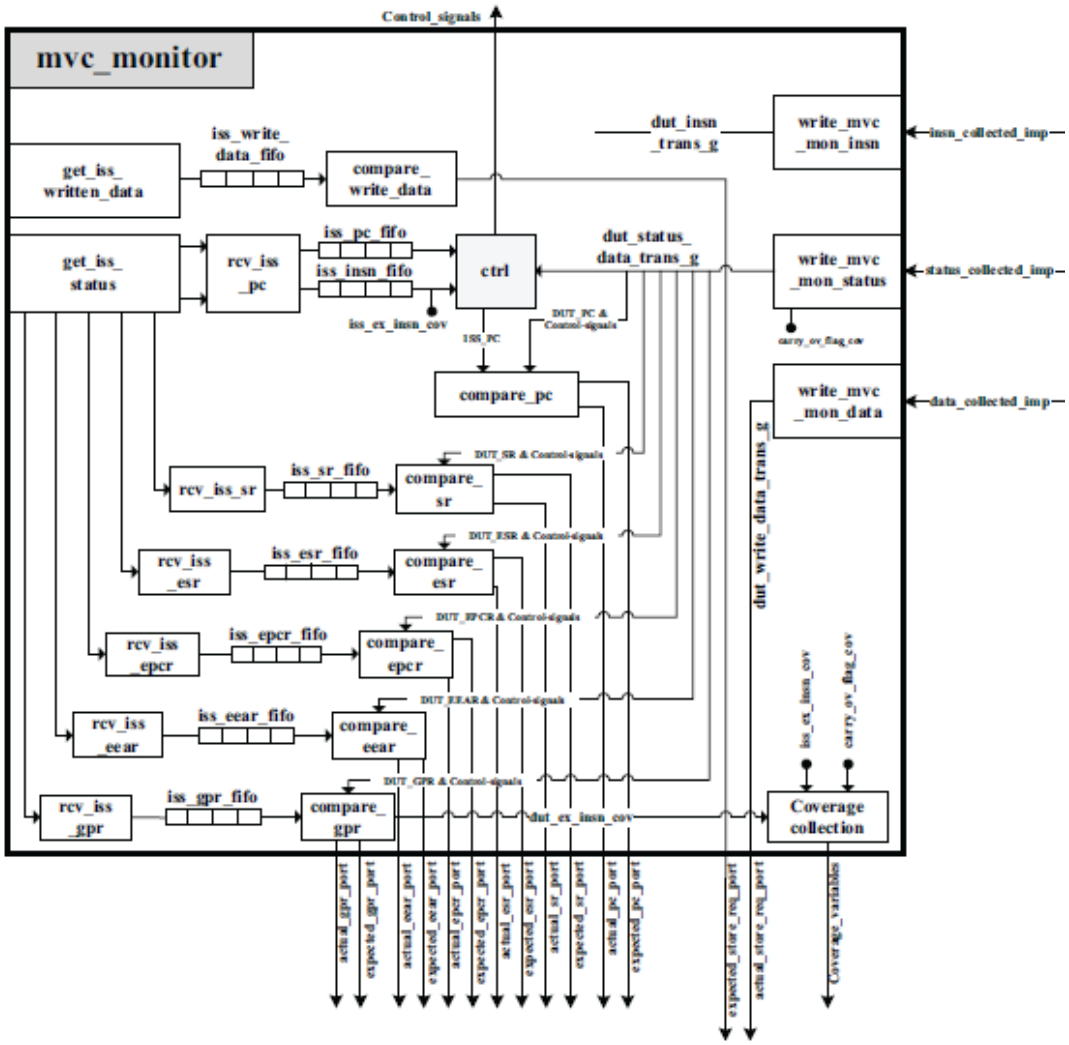

Figure 5.14: Module Monitor

To compute the verification coverage the module monitor sends the instructions of those that are executed on the golden model and on the DTP-XBUS-2 to an implemented coverage model. Additionally, it sends a few status flags of the DTP-XBUS-2 which are essential for a satisfactory coverage collection. These flags include

(i) The *carry flag,*

(ii) The *overflow flag* and

(iii) The *conditional branch flag*.

5.10.2 Scoreboard

The scoreboard receives the status registers and data along with the address (for Stores) from the module monitor through standard TLM ports. It receives the status of the golden model (expected_*_port) and the status of the DUV (actual_*_port). It implements an individual comparator for each stakeholder in the status and data transactions e.g., PC, SR, address to store data, etc. After comparison the scoreboarding is executed to generate the final report for each stakeholder.

5.10.3 Virtual Sequencer

The verification environment contains a virtual sequencer to synchronize the timing and data between

1) The golden model (ISS or simICS),
2) The instruction interface and
3) The data interface.

The instruction sequencer generates sequences of instructions. The data sequencer generates sequences of data. There is no co-ordination between these sequencers. This co-ordination is necessary to control the sequence generation on the instruction and data interfaces. Moreover, the instruction and data transactions have to be transmitted to the golden model first and then to the DTP-XBUS-2 (DUV). Therefore, a regulating agent must be implemented at a higher level of abstraction in order to allow the fine control of the verification environment for a particular test. The virtual sequencer contains the instances of the instruction sequencer and the data sequencer along with a virtual sequence library. This library encloses the virtual sequences which are executed on the virtual sequencer and control the coordination between the instruction sequencer, the data sequencer and the golden model. The virtual sequences are a constrained random generation of the scenarios (a sequence of instruction types e.g., ADD, MUL, etc.). When the DTP-XBUS-2 sends an

instruction fetch request the virtual sequencer picks the next instruction in the sequence (e.g., ADD) and generates its constrained random transaction (binary code of ADD instruction e.g., 0xe0841800). The transaction is generated on the instruction sequencer by using the local sequence library of the instruction sequencer. Before sending this transaction to the instruction driver the virtual sequencer first sends it to the golden model. If this instruction is a Load, the virtual sequencer also provides a randomized data to the golden model.

The golden model finishes execution and sends the status and result back to the module monitor.

After this the virtual sequencer allows the instruction sequencer to send this instruction's transaction to the instruction driver. If the instruction is a Load, the virtual sequencer uses the same data sent to the golden model and generates a constrained data transaction on the data sequencer by using its local sequence library (data sequence library). When this instruction is executed on the DUV and sends a READ request, this data transaction is sent to the data driver. The virtual sequencer also implements a complex mechanism to offer interactive behavioural modelling by using control signals of the DUV received from the module monitor. One instance of this mechanism is to stop sending instructions to the golden model (sending instructions to the DUV never stops) if an exception has been signalled in the XBUS-2 pipeline. It is because the XBUS-2 instruction pipeline is flushed and following instructions will never be executed. Whereas, the golden model (simICS) executes instructions at once in zero simulation time as it is fed with instruction before transmitting to the DTP-XBUS-2 (DUV).

5.11 Conclusion

This chapter presents the verification implementation strategy for DTP-XBUS-2. This chapter begins with the discussion about memory verification strategy through the construction of BFM. A section is dedicated to the discussion of the golden model of simICS which is used to verify the operation of SPARC v9. In this section, the PLI interfaces are discussed as the possible implementation strategy for the data agent that communicates with the simICS golden model. This chapter then proceeds to explain the implementation of IVC and MVC as discussed in Chapter 3.

Chapter 6: Experimental Results

6.1 Introduction

This chapter presents the actual test results obtained with DTP-XBUS-2 (XB2) core implemented on the ML505 Xilinx FPGA Development board and the comparison of the performance with single and dual core SoCs in computing statistics for large image using the designs and verification methodologies as described in the previous chapters. The functional and structural verification results during power-on are also presented.

The processor usage, timing measurements and eye measurement are gauge through the Xilinx ChipScope Pro with IBERT core. After a connection is established, the values are updated every second. The setup is as shown in the figure below (Figure 6.1)

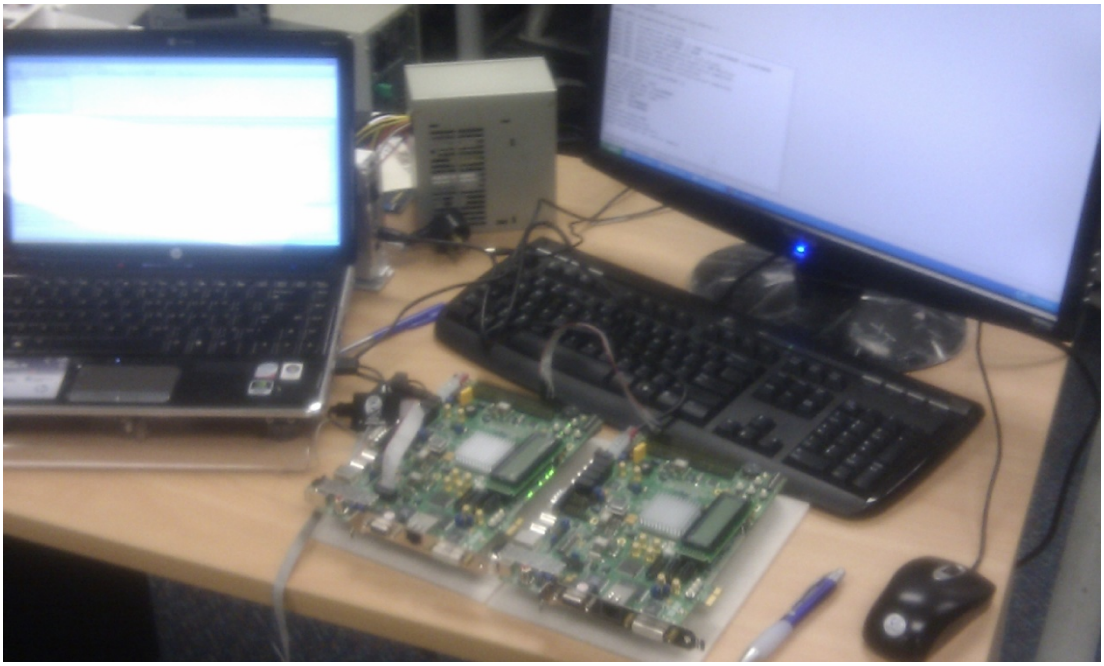

Figure 6.1: Experimental Setup

6.2 DTP-XBUS-2 Power-On Test Results

6.2.1 Overview

Functional verification is performed with all components of the XB2 interconnected and the test-bench as setup in chapter 4 using the methodologies as described in Chapter 3. These components of Xb2 are as described in Chapter 2. A test program is executed through the

JTAG chain to validate the basic functionalities of the XB2 core during power-on. For this purpose, a memory initialization file (IHex) of the test program is first generated using the SPARC V9 tool chain and boot-loaded into the ML505 development board. The SPARC V9 will execute this binary encoded file to perform in-circuit functional and structural tests and initiate a basic transfer operation through the XB2 core.

6.2.2 SPARC V9 Execution Results

6.2.2.1 Execution Results

The following are the most important aspects during XB2 functional verification.

1) SPARC V9 should fetch correctly the instructions from the correct addresses inside the Random Access Memory (RAM).
2) SPARC V9 should calculate the correct execution result and initiate XB2 for a single frame transfer operation.

6.2.2.2 Instruction Fetch

On power-on or reset, SPARC V9 should fetch the first instruction correctly from its reset address (0x0000_0100) from the DDR RAM. Figure 6.2 shows the expected waveform of the SPARC V9 instruction interface (PLB) that fetches instructions from the DDR RAM and feeds them to the processor. After the first instruction fetch, the processor should initiate the following tests as described in chapter 5 before coming to a halt:

1) Sequential Single Write/Read Access test
2) Random Single Write/Read Access test
3) Idle Cycle test
4) Random Block Write/Read Access test

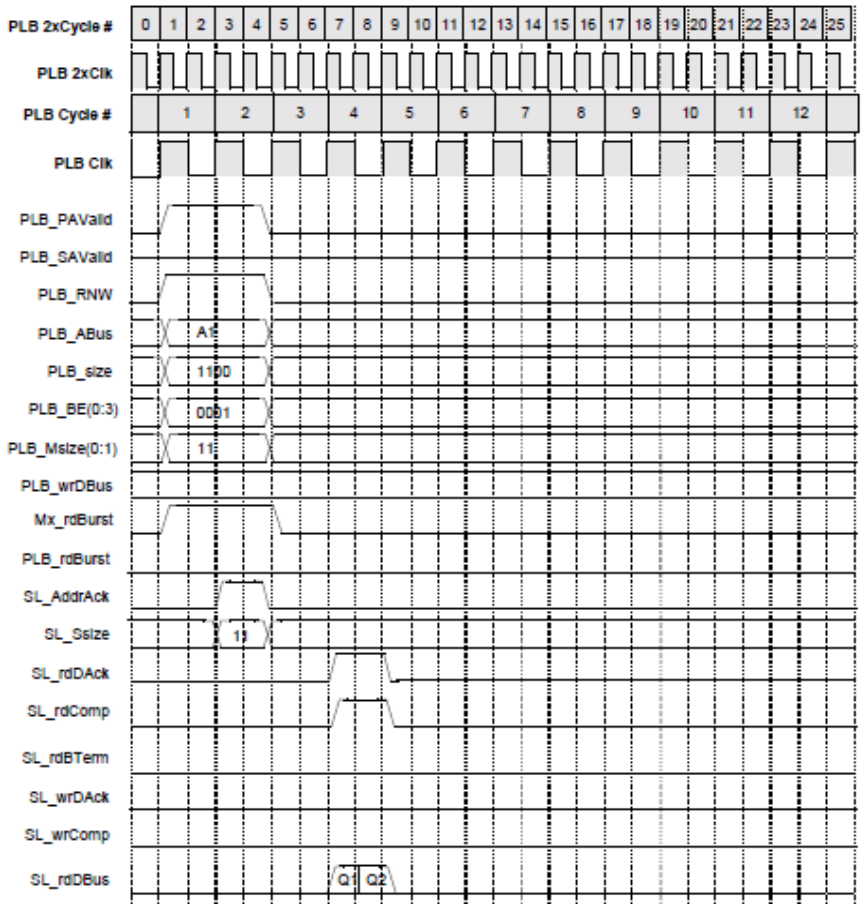

Figure 6.2: SPARC V9 expected instruction fetch waveform.

Figure 6.3 shows the test executed on the SPARC V9 processor. At this stage the SPC_TB has been completely initialized and ready for performance analysis. The tests are designed to cover the complete address space of the RAM. It first writes a data value to an address, then reads from the same address and finally compares both data values. The test results indicate that the SPARC V9 core and memory unit are fully functional.

```
# Top level module:
#       Spc_tb_top
# Loading SPC_TB
# Initializing simICS..... simICS initialized
# Loading sv_std.std
# Loading work.dtp_xb2(fast)
# Loading work.xbus_tb(fast)
# Loading work.plbif(fast)
# Loading work.ramtest(fast)
# Loading std.standard
# Loading ieee.std_logic_1164(body)
# Loading ieee.numeric_std(body)
# Loading work.global_pack
# Loading std.textio(body)
# Loading work.mem_pack(body)
# Loading work.ram(rtl)#1
run -a
# Testing sequential Single Write/Read Access
# Testing sequential Single Write/Read Access: Passed
# Testing random Single Write/Read Access
# Testing random Single Write/Read Access: Passed
# Testing Idle Cycle
# Testing Idle Cycle: Passed
# Testing random Block Write/Read Access
# Testing random Block Write/Read Access: Passed
# Power-on Test successfully completed. Proceeding to load main test xb2_drv
```

Figure 6.3: Memory Test Results

6.2.2.3 Single Frame Transfer

After the completion of memory test, the ML505 initiates the OBP boot-loader (Figure 6.4) and load the SunOS operating system.

Figure 6.4: OPB Boot-loader

Once the operating system is loaded, the kernel will invoke the xb2_drv program through SPC_TB which will then initiate a single frame transfer (Figure 6.5).

```
# xb2_drv
# Loading [ loopback:default starting (loopback network interface) ]
# Loading [ physical:default starting (physical network interfaces) ]
# [ milestone/name-services:default starting dtp ]
# [ system/identify:node starting (system identity (nodename)) ]
# [ system/metainit:default starting (xb2 initialization) ]
# xb2 dummy transfer
# xb2 dummy transfer: passed
```

Figure 6.5: Single frame transfer.

The results obtained validate the basic functionality of the DTP-XBUS-2 core prior to the execution of more tests as in the following section to screen out defects.

6.3 DTP-XBUS-2 Complete Verification

Figure below (Figure 6.6) presents the results of the address coverage for DTP-XBUS2 and other attached master and slave components. This coverage model is implemented to verify the transfer operation initiated by the components and transferred through the xb2 channel.

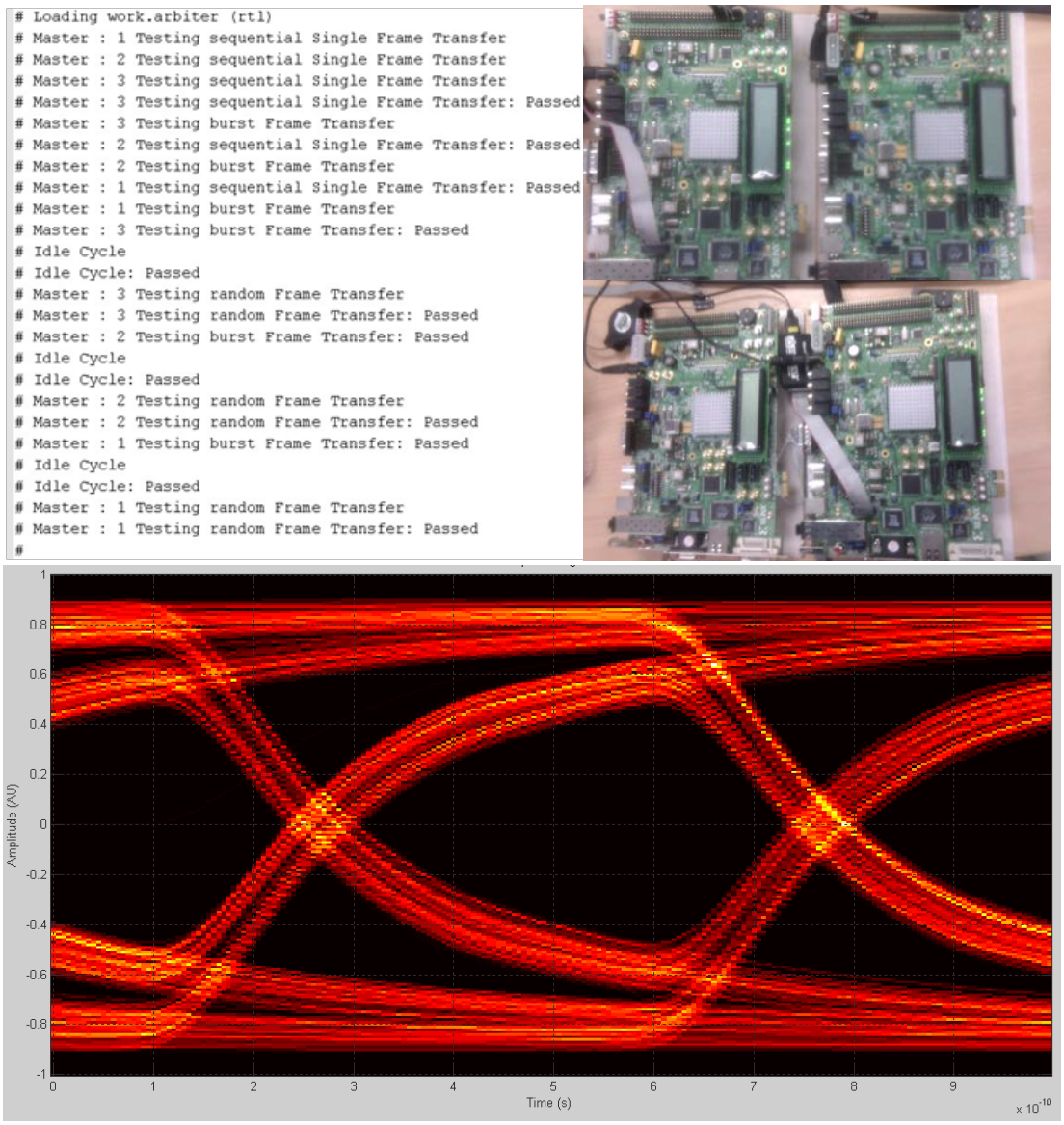

Figure 6.6: Verification Coverage

The results obtained confirm that DTP-XBUS-2 core is fully operational at the clock rate of 100MHz. The eye diagram on the receiver shows healthy system performance.

To evaluate the performance of the DTP-XBUS-2 core, the following Measures of Effectiveness (MOE) are selected: Single Core DTP-XBUS-2 SoC against Single Core SoC, Single Core DTP-XBUS-2 SoC against Single Core SoC with extended memory, Single Core DTP-XBUS-2 SoC against Dual Core SoC, Single Core DTP-XBUS-2 SoC against Dual Core SoC with extended memory. The same tests are repeated with Dual Core DTP-XBUS-2 SoC. A separate analysis is performed to benchmark the performance gain by integrating different IP cores externally. The system is setup as shown in figure 6.7

Figure 6.7: System setup

As the properties of natural images are important not only in image compression but also for the study of sensory processing in biology, medical applications in imaging and more, the computation of statistics for large image is selected for this purpose. The steps involved are as listed below.

1) Truecolor composite construction (Figure 6.8)
2) First attempt on image enhancement (Figure 6.9)
3) Histogram Accumulation Class examination (Figure 6.10)
4) Accumulation Class sampling (Figure 6.11)
5) Truecolor composite enhancement with a contrast stretch (Figure 6.12)

The throughput, P, CPU Performance, CPUP and memory usage, M are normalized according to the relation

$$P_{norm} = \frac{cumulative(P)}{(P_{step1} + P_{step2} + P_{step3} + P_{step4} + P_{step5}))} \quad \text{---------- (2)}$$

$$CPUP_{norm} = \frac{CPU_{secondsperinstruction}}{\Delta t_{computationtime}} \quad \text{---------- (3)}$$

$$M_{norm} = \frac{M_{usage}}{M_{totalavailablememory}} \quad \text{---------- (4)}$$

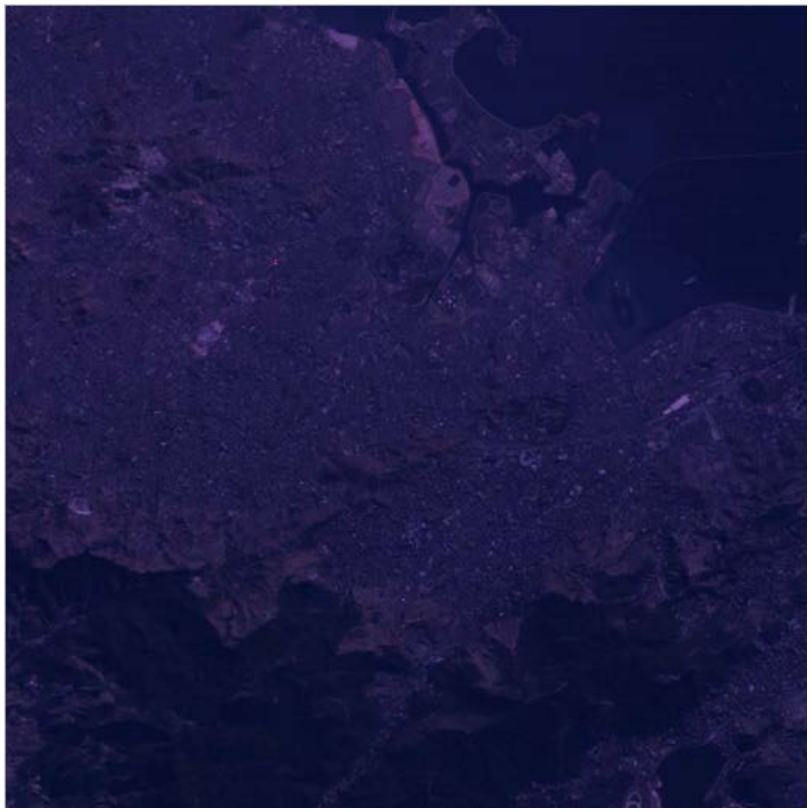

Figure 6.8: Truecolor composite

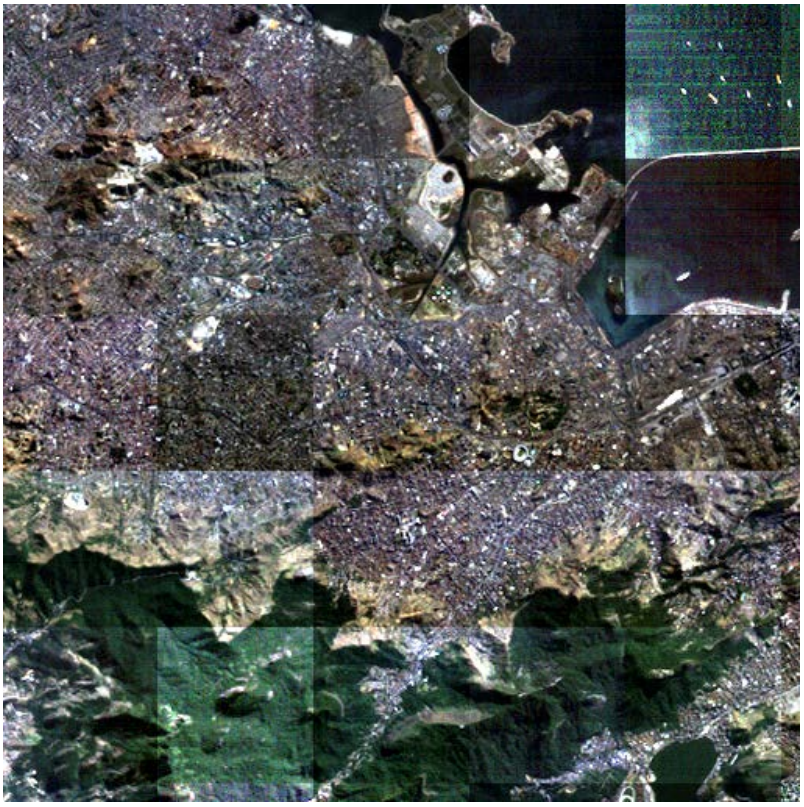

Figure 6.9: First attempt enhancement

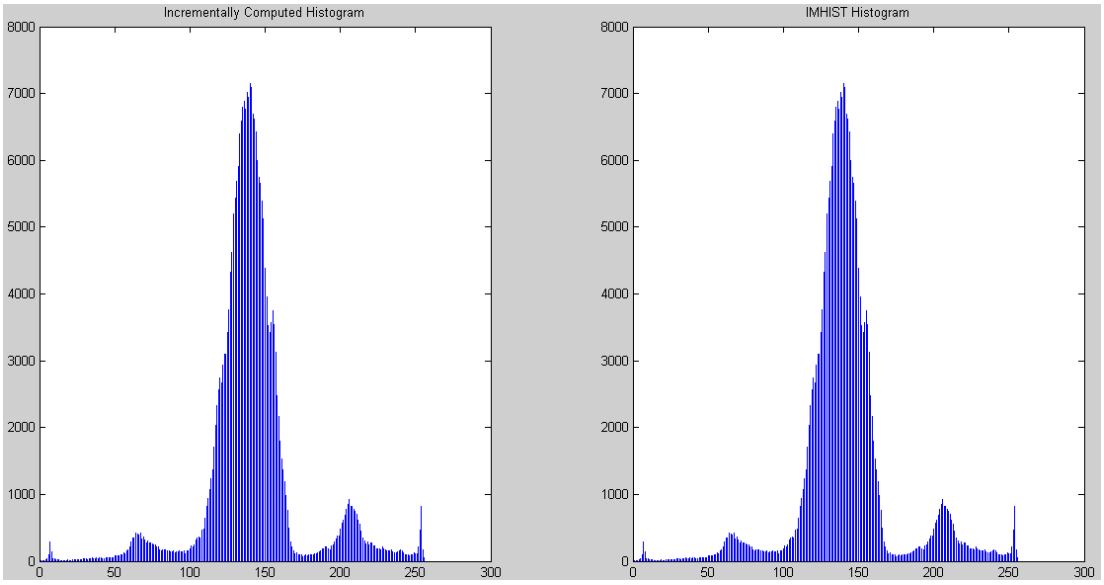

Figure 6.10: Histogram Accumulation Class examination

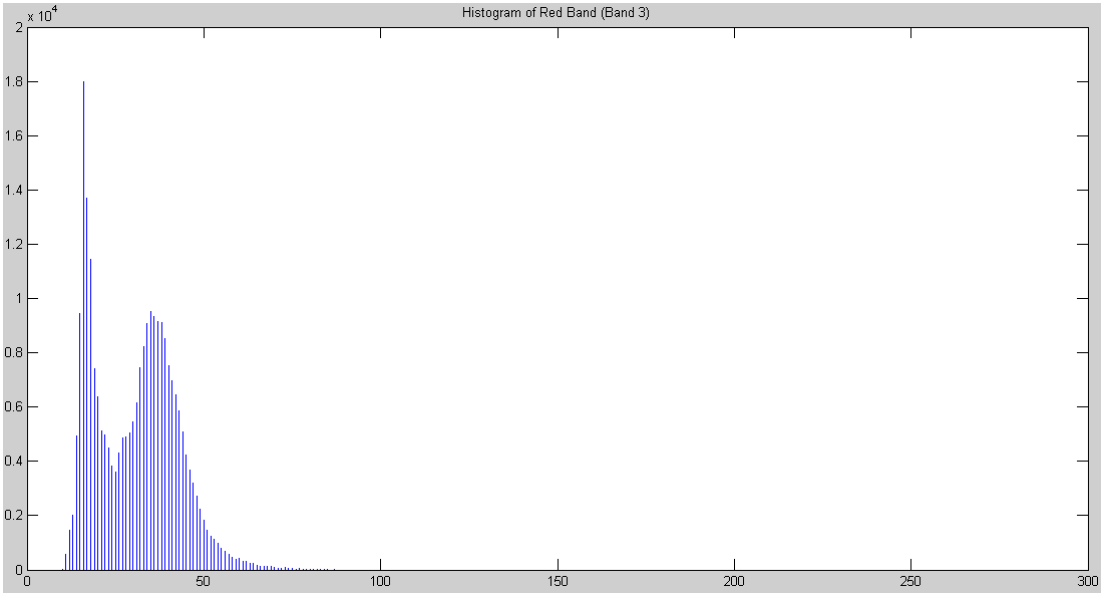

Figure 6.11 Accumulation Class Sampling

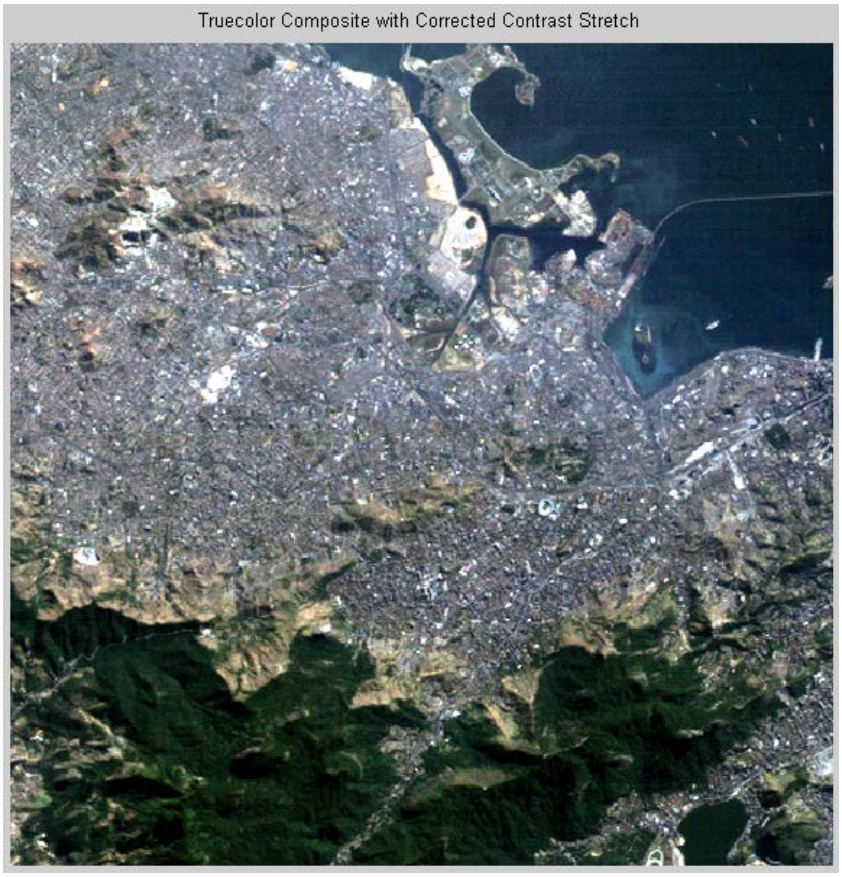

Figure 6.12: Truecolor composite enhancement with a contrast stretch

6.4 DTP-XBUS-2 SoC Performance Analysis

The setup for Single Core DTP-XBUS-2 SoC is as shown in the figure below (Figure 6.13)

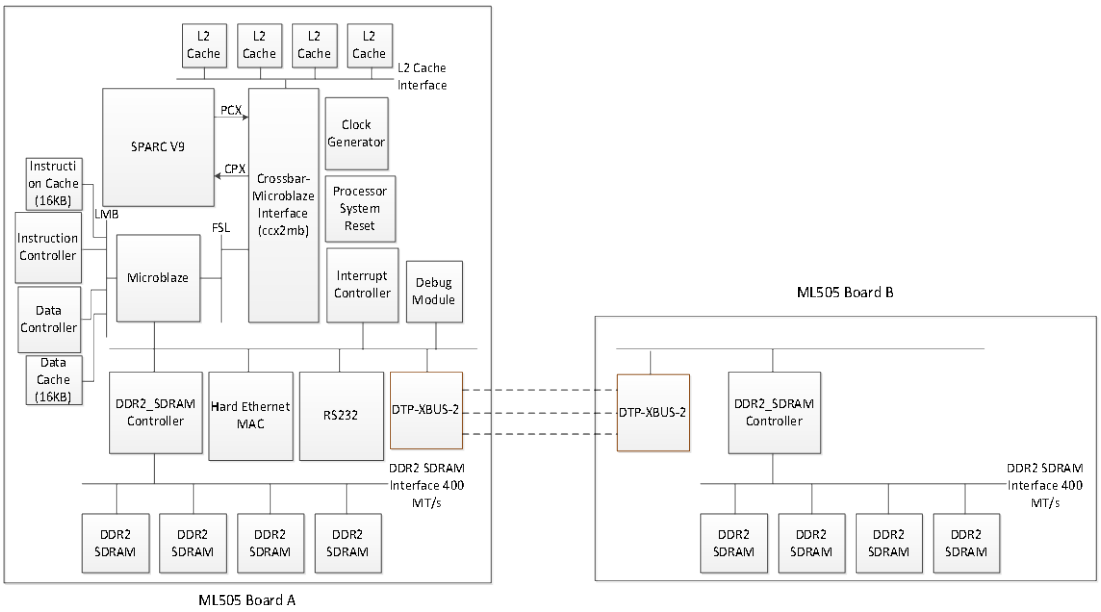

Figure 6.13: Single Core DTP-XBUS-2 SoC setup

The setup for Dual Core DTP-XBUS-2 SoC is as shown in the figure below (Figure 6.14). The CPU performance and memory usage is monitored throughout the test.

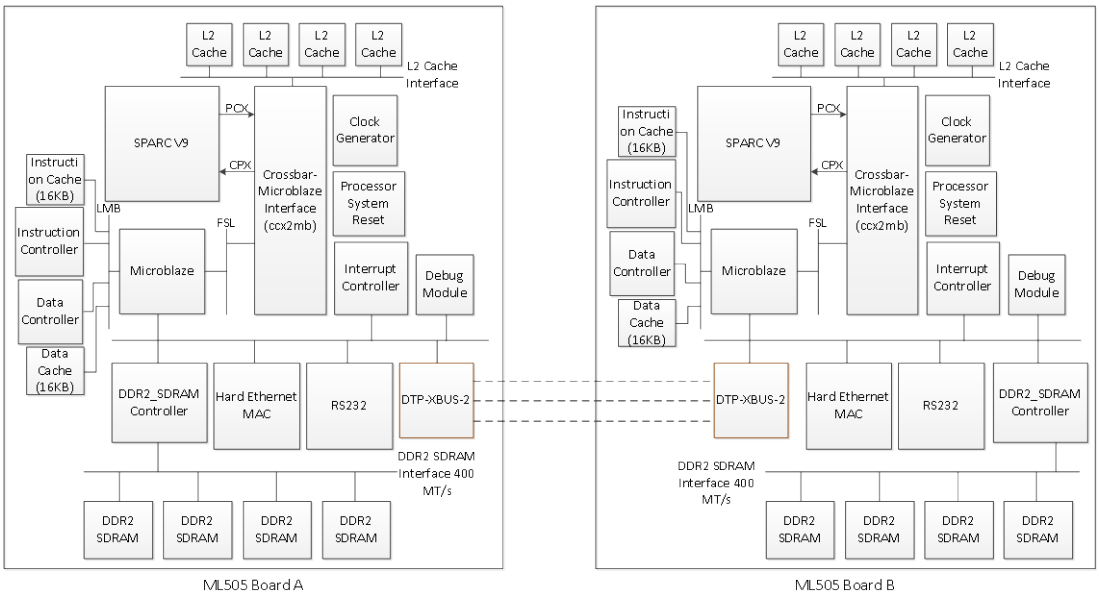

Figure 6.14: Dual Core DTP-XBUS-2 SoC

A standard Xilinx GPU core is integrated externally as shown in the figure below (Figure 6.15)

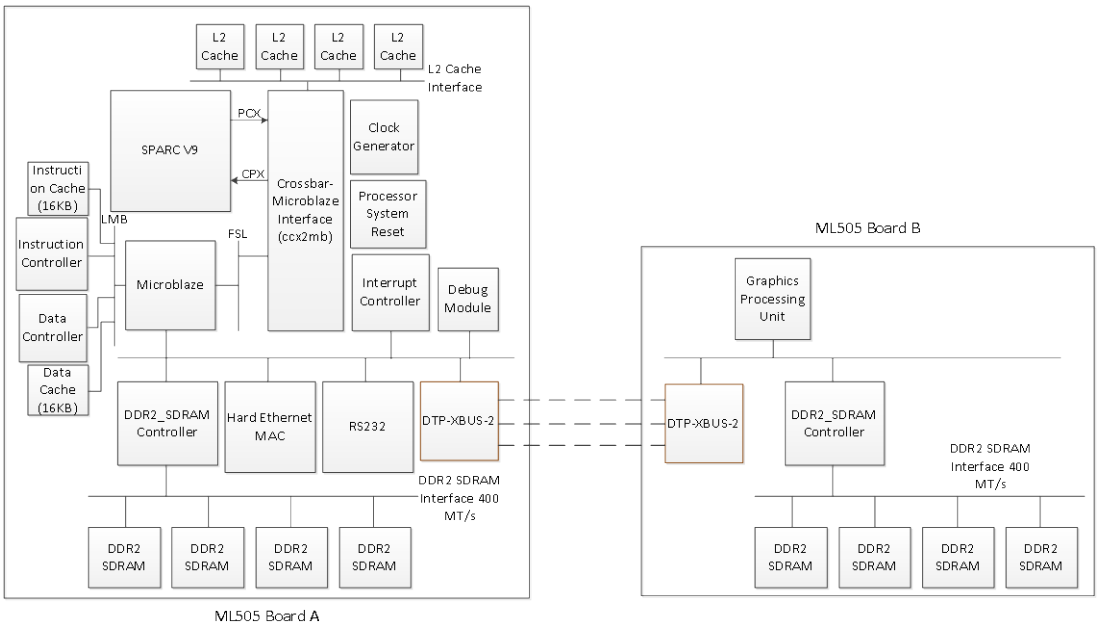

Figure 6.15: DTP-XBUS-2 SoC with External GPU IP core.

The results of computation are presented as follow (Figure 6.16).

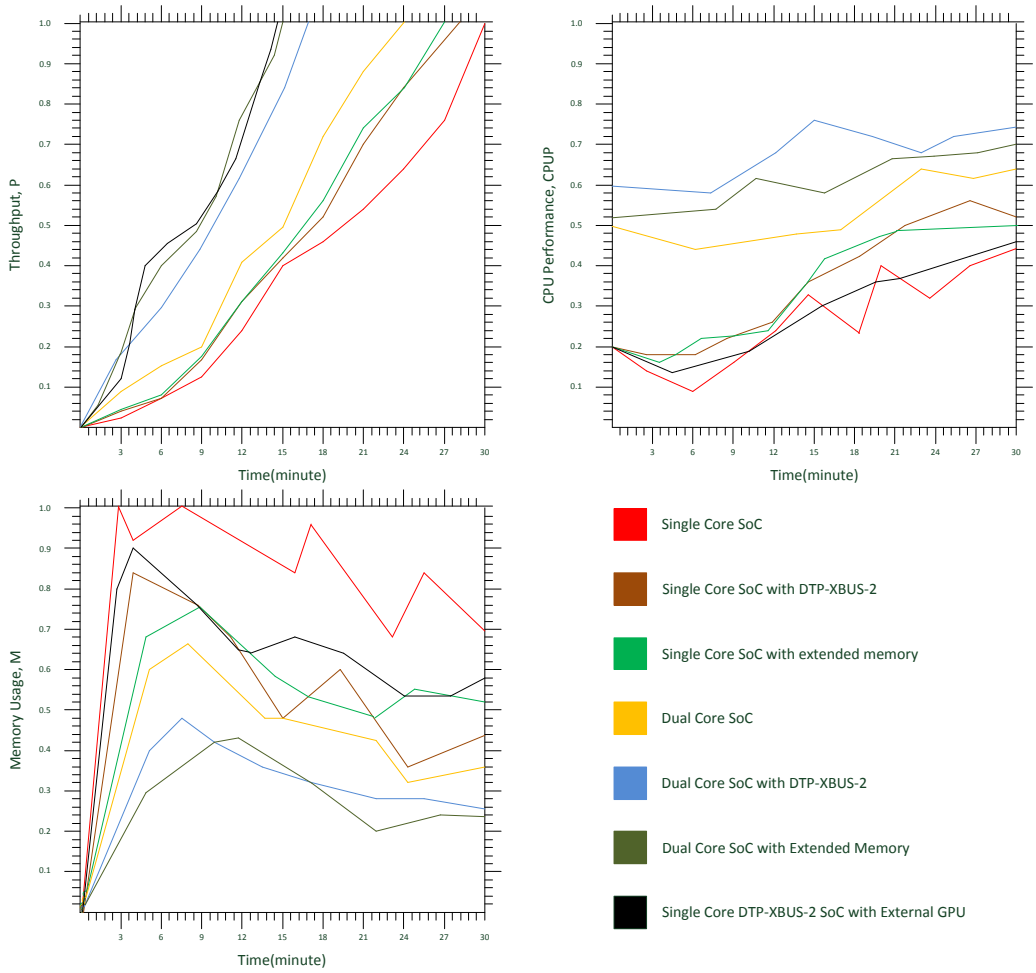

Figure 6.16: Performance Analysis

As observed, the integration of dedicated IP core has a significant influence over the CPU performance. This phenomenon is explained in [39], which is due to the inherent parallelism in the graphics core which makes it suitable for on-demand computation such as image processing. Single core SoC has the poorest performance due to the frequent occurrence of memory bottle-necks as observed. The performance gain in Single core SoC with DTP-XBUS-2 is a direct result of memory resource extension which reduces memory bottle-neck. An interesting phenomenon is observed with Dual Core SoC and Dual Core SoC with DTP-XBUS-2. Dual Core SoC with DTP-XBUS-2 demonstrates performance gain due to the fact that memory allocation is more flexible and uses memory resources available on both systems (ML505 board A and board B). The Dual Core SoC is constraint by the memory resources available on-board (ML505 board A). In general, both systems demonstrate performance gain compared to Single Core SoC due to thread level parallelism.

6.5 Conclusion

The DTP-XBUS-2 SoC is implemented on Xilinx ML505 development board with IBERT core for signal monitoring and analysis. The system stability is demonstrated with the eye diagram on the receiver which shows healthy eye-opening. The performance gain is demonstrated with the computation of statistics for natural images. It clearly observed that DTP-XBUS-2 core allows external IP cores to be seamlessly integrated and unlike SoC, DTP-XBUS-2 has more flexibility in terms of space utilization. The availability of dedicated processing core and extended memory resources explain why DTP-XBUS-2 system generally performs better.

Chapter 7: Conclusion and Future work

7.1 Conclusion

This thesis is divided into two major parts: the first part describes the implementation of the DTP-XBUS-2 (xb2) core and the second part describes the verification methodologies available for xb2. The xb2 core is used with advance control architecture that operates the DTP-XBUS-2 MAC. The function includes configuring the xb2 transceiver and the interface for DTP protocol. The complete system comprises the SPARC V9 processor, an xb2 core, memory subsystems and several other interfaces. The SPARC V9 toolchain is used to generate the memory initialization file for the processor core and early code analysis. The verification of the CPU Subsystem includes the coverage-driven constrained random verification of the xb2 core, the sub-bus system and the memory subsystem. For the verification of the SPARC V9 core, a golden model is implemented using the simICS with a SystemC wrapper around to incorporate the verification environment. Moreover, OVM is used to implement a configurable and reusable verification environment. This thesis includes the co-verification with the programming languages Verilog, C, C++ (SystemC) and PLI. The verification results of the Sub-bus system and the memory system show that both subsystems are implemented correctly. These tests are performed during power-on. Apart from that, performance analysis is performed to benchmark the gain for systems using the DTP-XBUS-2 core.

7.2 Future work

There are several improvements which could be done for the development of the xb2 core. Some possible improvements are listed below.

1) Implementation of load scheduling techniques directly on the DTP-XBUS-2 MAC for more efficient load distribution in a daisy-chain network. The proposed method should dynamically derive loads as the introduction of DTP-XBUS-2 enables the hot-plug/hot-swap of resources and feature sets.
2) Investigation of AGBIST as described in chapter 5. As described, AGBIST does not use LFSR to generate test vectors but uses FSM to generate state dependent test vectors which could be used as method for test vector compression.

3) Introduction of state encryption on the xb2 transceiver. As xb2 core allows system level components to directly communicate with internally integrated IP cores, a concern arises on data privacy. Current state registers have a predictable output on reset. A method is proposed that uses a hybrid of flash technologies that directly embed state information on the state registers.

Furthermore, it could be possible to enhance the throughput of the DTP-XBUS-2 by maximizing the throughput of PLB interfaces of the SPARC V9 core.

Abbreviations A

ASIC	Application Specific Integrated Circuit
ATE	Automated Test Equipment
BFM	Bus Functional Model
BRAM	Block RAM
CISC	Complex Instruction Set Computers
CMT	Chip Multi-Threaded
CPU	Central Processing Unit
CPX	Cache to Processor Crossbar
CRC	Cyclic Redundancy Check
DDR RAM	Double Data Rate RAM
DFT	Design for Testability
DIMM	Dual In-line Memory Module
DRAM	Dynamic RAM
DTE	Data Terminal Equipments
DTP	Data Transfer Protocol
DUV	Design Under Validation
ECC	Error Control Code
FBD	Fully Buffered DIMM
FCS	Frame Check Sequence
FIFO	First in First out
FPGA	Field Programmable Gate Array
FSL	Fast Simplex Link
IC	Integrated Circuit
ILP	Instruction-Level Parallelism
IP	Intellectual Property
JTAG	Joint Test Action Group

LMB	Local Memory Bus
LUT	Lookup Table
MAC	Medium Access Controller
MUX	Multiplexer
NRE	Non-Return Engineering
NOC	Network on Chip
PCI	Peripheral Component Interconnect
PCI-EX	Peripheral Component Interconnect Express
PCX	Processor to Cache Crossbar
PHY	ISO-OSI Physical Layer
PLB	Processor Local Bus
PLI	Programming Language Interface
PPN	Physical Page Number
RAM	Random Access Memory
RISC	Reduced Instruction Set Computers
SiP	System in Package
SiS	System in System
SoC	System on Chip
SLP	System-Level Parallism
TLP	Thread-Level Parallism
TLB	Translation Look-aside Buffers
VLSI	Very Large Scale Integration
XBUS	Extensible Bus

Hardware Implementation B

This appendix presents the details on the hardware implementation of DTP-XBUS-2

Below is the DTP_XBUS_2 top-level implementation

```
entity DTP_XBUS_2 is
  generic
   (
    C_BASEADDR                     : std_logic_vector := X"FFFFFFFF";
    C_HIGHADDR                     : std_logic_vector := X"00000000";
    C_SPLB_CLK_PERIOD_PS           : integer := 10000;
    C_SPLB_AWIDTH                  : integer range 32 to 36 := 32;
    C_SPLB_DWIDTH                  : integer range 32 to 128:= 32;
    C_SPLB_P2P                     : integer := 0;
    C_SPLB_MID_WIDTH               : integer := 1;
    C_SPLB_NUM_MASTERS             : integer := 1;
    C_SPLB_NATIVE_DWIDTH           : integer range 32 to 128:= 32;
    C_SPLB_SMALLEST_MASTER         : integer range 32 to 128:= 32;
    C_SPLB_SUPPORT_BURSTS          : integer range 0 to 1:= 0;
    C_INCLUDE_MDIO                 : integer := 1;
    C_INCLUDE_INTERNAL_LOOPBACK    : integer := 0;
    C_INCLUDE_GLOBAL_BUFFERS       : integer := 0;
    C_DUPLEX                       : integer range 0 to 1:= 1;
    C_TX_PING_PONG                 : integer range 0 to 1:= 0;
    C_RX_PING_PONG                 : integer range 0 to 1:= 0
    );
  port
    (
    PHY_tx_clk      : in std_logic;
    PHY_rx_clk      : in std_logic;
    Xb2_tx_stat     : in std_logic;
    Xb2_rx_dv       : in std_logic;
    Xb2_rx_data     : in std_logic_vector (3 downto 0);
    Xb2_tx_ack      : in std_logic;
    Xb2_rx_ack      : out std_logic;
    Xb2_rx_stat     : out std_logic;
    Xb2_tx_en       : out std_logic;
    Xb2_tx_data     : out std_logic_vector (3 downto 0);
    Xb2_phy_data_ctrl_I    : in  std_logic;
    Xb2_phy_data_ctrl_O    : out std_logic;
    Xb2_phy_data_ctrl_T    : out std_logic;
    Xb2_phy_data_clk: out std_logic;
```

```
IP2INTC_Irpt    : out std_logic;
SPLB_Clk        : in  std_logic;
SPLB_Rst        : in  std_logic;
PLB_ABus        : in  std_logic_vector(0 to C_SPLB_AWIDTH-1);
PLB_UABus       : in  std_logic_vector(0 to 31);
PLB_PAValid     : in  std_logic;
PLB_SAValid     : in  std_logic;
PLB_rdPrim      : in  std_logic;
PLB_wrPrim      : in  std_logic;
PLB_masterID    : in  std_logic_vector(0 to C_SPLB_MID_WIDTH-1);
PLB_abort       : in  std_logic;
PLB_busLock     : in  std_logic;
PLB_RNW         : in  std_logic;
PLB_BE          : in  std_logic_vector(0 to (C_SPLB_DWIDTH/8)-1);
PLB_MSize       : in  std_logic_vector(0 to 1);
PLB_size        : in  std_logic_vector(0 to 3);
PLB_type        : in  std_logic_vector(0 to 2);
PLB_lockErr     : in  std_logic;
PLB_wrDBus      : in  std_logic_vector(0 to C_SPLB_DWIDTH-1);
PLB_wrBurst     : in  std_logic;
PLB_rdBurst     : in  std_logic;
PLB_wrPendReq   : in  std_logic;
PLB_rdPendReq   : in  std_logic;
PLB_wrPendPri   : in  std_logic_vector(0 to 1);
PLB_rdPendPri   : in  std_logic_vector(0 to 1);
PLB_reqPri      : in  std_logic_vector(0 to 1);
PLB_TAttribute  : in  std_logic_vector(0 to 15);
Sl_addrAck      : out std_logic;
Sl_SSize        : out std_logic_vector(0 to 1);
Sl_wait         : out std_logic;
Sl_rearbitrate  : out std_logic;
Sl_wrDAck       : out std_logic;
Sl_wrComp       : out std_logic;
Sl_wrBTerm      : out std_logic;
Sl_rdDBus       : out std_logic_vector(0 to C_SPLB_DWIDTH-1);
Sl_rdWdAddr     : out std_logic_vector(0 to 3);
Sl_rdDAck       : out std_logic;
Sl_rdComp       : out std_logic;
Sl_rdBTerm      : out std_logic;
Sl_MBusy        : out std_logic_vector (0 to C_SPLB_NUM_MASTERS-1);
Sl_MWrErr       : out std_logic_vector (0 to C_SPLB_NUM_MASTERS-1);
Sl_MRdErr       : out std_logic_vector (0 to C_SPLB_NUM_MASTERS-1);
Sl_MIRQ         : out std_logic_vector (0 to C_SPLB_NUM_MASTERS-1)
);
```

```
attribute syn_maxfan                    : integer;
attribute syn_maxfan      of SPLB_Clk   : signal is 10000;
attribute syn_maxfan      of SPLB_Rst   : signal is 10000;
attribute MAX_FANOUT                    : string;
attribute MAX_FANOUT      of SPLB_Clk   : signal is "10000";
attribute MAX_FANOUT      of SPLB_Rst   : signal is "10000";
attribute uselowskewlines               : STRING;
attribute uselowskewlines of PHY_tx_clk : signal is "yes";
attribute uselowskewlines of PHY_rx_clk : signal is "yes";
attribute HDL             :   string;
attribute IMP_NETLIST     :   string;
attribute IP_GROUP        :   string;
attribute IPTYPE          :   string;
attribute STYLE           :   string;
attribute SIGIS           :   string;
attribute ASSIGNMENT      :   string;
attribute ADDRESS         :   string;
attribute PAIR            :   string;
attribute SIGIS       of  SPLB_Clk          :  signal   is  "CLK";
attribute SIGIS       of  SPLB_Rst          :  signal   is  "RST";
attribute SIGIS       of  IP2INTC_Irpt      :  signal   is
                                                "INTR_EDGE_RISING";
attribute ASSIGNMENT  of  C_BASEADDR        :  constant is  "REQUIRE";
attribute ASSIGNMENT  of  C_HIGHADDR        :  constant is  "REQUIRE";
attribute ADDRESS     of  C_BASEADDR        :  constant is  "BASE";
attribute ADDRESS     of  C_HIGHADDR        :  constant is  "HIGH";
attribute PAIR        of  C_BASEADDR        :  constant is  "C_HIGHADDR";
attribute PAIR        of  C_HIGHADDR        :  constant is  "C_BASEADDR";
end XBUS2_MAC;

------------------------------------------------------------------------------
---
-- Architecture
------------------------------------------------------------------------------
---

architecture imp of DTP_XBUS_2 is
constant NODE_MAC : bit_vector := x"00005e00FACE";
signal phy_rx_clk_i    : std_logic;
signal phy_tx_clk_i    : std_logic;
signal phy_rx_data_i   : std_logic_vector(3 downto 0);
signal phy_tx_data_i   : std_logic_vector(3 downto 0);
signal phy_dv_i        : std_logic;
signal phy_rx_er_i     : std_logic;
signal phy_tx_en_i     : std_logic;
```

```
signal Loopback        : std_logic;
signal phy_rx_data_in  : std_logic_vector (3 downto 0);
signal phy_dv_in       : std_logic;
signal phy_rx_data_reg : std_logic_vector(3 downto 0);
signal phy_rx_er_reg   : std_logic;
signal phy_dv_reg      : std_logic;
signal phy_tx_clk_core    : std_logic;
signal phy_rx_clk_core    : std_logic;
component FDRE
  port
   (
    Q  : out std_logic;
    C  : in std_logic;
    CE : in std_logic;
    D  : in std_logic;
    R  : in std_logic
   );
end component;
component BUFG
  port (
     O  : out std_ulogic;
     I : in std_ulogic := '0'
  );
end component;
component BUFGMUX
  port (
     O  : out std_ulogic;
     I0 : in std_ulogic := '0';
     I1 : in std_ulogic := '0';
     S  : in std_ulogic
  );
end component;
component BUF
  port(
    O : out std_ulogic;

    I : in std_ulogic
    );
end component;

  attribute IOB         : string;

begin
   PHY_rst_n   <= not SPLB_Rst ;
   LOOPBACK_GEN: if C_INCLUDE_INTERNAL_LOOPBACK = 1 generate
```

```
   begin
      INCLUDE_BUFG_GEN: if C_INCLUDE_GLOBAL_BUFFERS = 1 generate
      begin
         CLOCK_BUFG_TX: BUFG
           port map (
             O  => phy_tx_clk_core,  --[out]
             I  => PHY_tx_clk        --[in]
           );
      end generate INCLUDE_BUFG_GEN;

      NO_BUFG_GEN: if C_INCLUDE_GLOBAL_BUFFERS = 0 generate
      begin

         phy_tx_clk_core  <= PHY_tx_clk;

      end generate NO_BUFG_GEN;

      CLOCK_MUX: BUFGMUX
        port map (
          O  => phy_rx_clk_core, --[out]
          I0 => PHY_rx_clk,      --[in]
          I1 => phy_tx_clk_core, --[in]
          S  => Loopback         --[in]
        );

      phy_rx_data_in <=  phy_tx_data_i when Loopback = '1' else
                         phy_rx_data_reg;
      phy_dv_in      <=  phy_tx_en_i   when Loopback = '1' else
                         phy_dv_reg;
      phy_rx_er_i    <= '0' when Loopback = '1' else
                         phy_rx_er_reg;
      phy_tx_clk_i <= not(phy_tx_clk_core);
      phy_rx_clk_i <= not(phy_rx_clk_core);
      ------------------------------------------------------------------------
---
      -- Registering RX signal
      ------------------------------------------------------------------------
---
      DV_FF: FDR
        port map (
          Q  => phy_dv_i,             --[out]
          C  => phy_rx_clk_i,         --[in]
          D  => phy_dv_in,            --[in]
          R  => SPLB_Rst);            --[in]
```

```
      ------------------------------------------------------------------
---
      -- Registering RX data input with clock mux output
      ------------------------------------------------------------------
---
      RX_REG_GEN: for i in 3 downto 0 generate
      begin
         RX_FF: FDRE
           port map (
             Q  => phy_rx_data_i(i),   --[out]
             C  => phy_rx_clk_i,       --[in]
             CE => '1',                --[in]
             D  => phy_rx_data_in(i),  --[in]
             R  => SPLB_Rst);          --[in]

      end generate RX_REG_GEN;

   end generate LOOPBACK_GEN;
   NO_LOOPBACK_GEN: if C_INCLUDE_INTERNAL_LOOPBACK = 0 generate
   begin

      INCLUDE_BUFG_GEN: if C_INCLUDE_GLOBAL_BUFFERS = 1 generate
      begin
         CLOCK_BUFG_TX: BUFG
           port map (
             O  => phy_tx_clk_core,  --[out]
             I  => PHY_tx_clk        --[in]
           );
         CLOCK_BUFG_RX: BUFG
           port map (
             O  => phy_rx_clk_core,  --[out]
             I  => PHY_rx_clk        --[in]
           );


      end generate INCLUDE_BUFG_GEN;
      NO_BUFG_GEN: if C_INCLUDE_GLOBAL_BUFFERS = 0 generate
      begin

         phy_tx_clk_core  <= PHY_tx_clk;
         phy_rx_clk_core  <= PHY_rx_clk;

      end generate NO_BUFG_GEN;

      -- Transmit and Receive clocks for core
```

```
   phy_tx_clk_i  <= not(phy_tx_clk_core);
   phy_rx_clk_i  <= not(phy_rx_clk_core);

   -- TX/RX internal signals
   phy_rx_data_i <= phy_rx_data_reg;
   phy_rx_er_i   <= phy_rx_er_reg;
   phy_dv_i      <= phy_dv_reg;

end generate NO_LOOPBACK_GEN;

IOFFS_GEN: for i in 3 downto 0 generate
attribute IOB of RX_FF_I : label is "true";
attribute IOB of TX_FF_I : label is "true";
begin
   RX_FF_I: FDRE
      port map (
         Q  => phy_rx_data_reg(i), --[out]
         C  => phy_rx_clk_core,    --[in]
         CE => '1',                --[in]
         D  => Xb2_rx_data(i),     --[in]
         R  => SPLB_Rst);          --[in]

   TX_FF_I: FDRE
      port map (
         Q  => Xb2_tx_data(i),     --[out]
         C  => phy_tx_clk_core,    --[in]
         CE => '1',                --[in]
         D  => phy_tx_data_i(i),   --[in]
         R  => SPLB_Rst);          --[in]

 end generate IOFFS_GEN;

IOFFS_GEN2: if(true) generate
   attribute IOB of DVD_FF : label is "true";
   attribute IOB of RER_FF : label is "true";
   attribute IOB of TEN_FF : label is "true";
   begin
      DVD_FF: FDRE
        port map (
          Q  => phy_dv_reg,      --[out]
          C  => phy_rx_clk_core, --[in]
          CE => '1',             --[in]
          D  => Xb2_rx_dv,          --[in]
          R  => SPLB_Rst);       --[in]
```

```
      RER_FF: FDRE
        port map (
          Q  => phy_rx_er_reg,   --[out]
          C  => phy_rx_clk_core, --[in]
          CE => '1',             --[in]
          D  => Xb2_rx_ack,       --[in]
          R  => SPLB_Rst);       --[in]

      TEN_FF: FDRE
        port map (
          Q  => Xb2_tx_en,       --[out]
          C  => phy_tx_clk_core, --[in]
          CE => '1',             --[in]
          D  => PHY_tx_en_i,     --[in]
          R  => SPLB_Rst);       --[in]

end generate IOFFS_GEN2;

xb2MAC : entity XBUS2_MAC
  generic map
     (
     C_DUPLEX               => C_DUPLEX,
     C_RX_PING_PONG         => C_RX_PING_PONG,
     C_TX_PING_PONG         => C_TX_PING_PONG,
     C_INCLUDE_MDIO         => C_INCLUDE_MDIO,
     NODE_MAC               => NODE_MAC,
     C_BASEADDR             => C_BASEADDR,
     C_HIGHADDR             => C_HIGHADDR,
     C_SPLB_AWIDTH          => C_SPLB_AWIDTH,
     C_SPLB_DWIDTH          => C_SPLB_DWIDTH,
     C_SPLB_P2P             => C_SPLB_P2P,
     C_SPLB_MID_WIDTH       => C_SPLB_MID_WIDTH,
     C_SPLB_NUM_MASTERS     => C_SPLB_NUM_MASTERS,
     C_SPLB_SUPPORT_BURSTS  => C_SPLB_SUPPORT_BURSTS,
     C_SPLB_SMALLEST_MASTER => C_SPLB_SMALLEST_MASTER,
     C_SPLB_CLK_PERIOD_PS   => C_SPLB_CLK_PERIOD_PS,
     C_SPLB_NATIVE_DWIDTH   => C_SPLB_NATIVE_DWIDTH
     )

  port map
     (
     SPLB_Clk       => SPLB_Clk,
     SPLB_Rst       => SPLB_Rst,
     PLB_ABus       => PLB_ABus,
     PLB_UABus      => PLB_UABus,
```

```
PLB_PAValid    => PLB_PAValid,
PLB_SAValid    => PLB_SAValid,
PLB_rdPrim     => PLB_rdPrim,
PLB_wrPrim     => PLB_wrPrim,
PLB_masterID   => PLB_masterID,
PLB_abort      => PLB_abort,
PLB_busLock    => PLB_busLock,
PLB_RNW        => PLB_RNW,
PLB_BE         => PLB_BE,
PLB_MSize      => PLB_MSize,
PLB_size       => PLB_size,
PLB_type       => PLB_type,
PLB_lockErr    => PLB_lockErr,
PLB_wrDBus     => PLB_wrDBus,
PLB_wrBurst    => PLB_wrBurst,
PLB_rdBurst    => PLB_rdBurst,
PLB_wrPendReq  => PLB_wrPendReq,
PLB_rdPendReq  => PLB_rdPendReq,
PLB_wrPendPri  => PLB_wrPendPri,
PLB_rdPendPri  => PLB_rdPendPri,
PLB_reqPri     => PLB_reqPri,
PLB_TAttribute => PLB_TAttribute,
Sl_addrAck     => Sl_addrAck,
Sl_SSize       => Sl_SSize,
Sl_wait        => Sl_wait,
Sl_rearbitrate => Sl_rearbitrate,
Sl_wrDAck      => Sl_wrDAck,
Sl_wrComp      => Sl_wrComp,
Sl_wrBTerm     => Sl_wrBTerm,
Sl_rdDBus      => Sl_rdDBus,
Sl_rdWdAddr    => Sl_rdWdAddr,
Sl_rdDAck      => Sl_rdDAck,
Sl_rdComp      => Sl_rdComp,
Sl_rdBTerm     => Sl_rdBTerm,
Sl_MBusy       => Sl_MBusy,
Sl_MWrErr      => Sl_MWrErr,
Sl_MRdErr      => Sl_MRdErr,
Sl_MIRQ        => Sl_MIRQ,
IP2INTC_Irpt   => IP2INTC_Irpt,
PHY_tx_clk     => phy_tx_clk_i,
PHY_rx_clk     => phy_rx_clk_i,
Xb2_tx_stat    => PHY_crs,
Xb2_rx_dv      => phy_dv_i,
Xb2_rx_data    => phy_rx_data_i,
Xb2_tx_ack     => PHY_col,
```

```
        PHY_rx_ack      => phy_rx_er_i,
        Xb2_tx_en       => PHY_tx_en_i,
        Xb2_tx_data     => PHY_tx_data_i,
        Xb2_phy_data_ctrl_I     => PHY_MDIO_I,
        Xb2_phy_data_ctrl_O     => PHY_MDIO_O,
        Xb2_phy_data_ctrl_T     => PHY_MDIO_T,
        Xb2_phy_data_clk        => PHY_MDC,
        Loopback        => Loopback
        );
end imp;
```

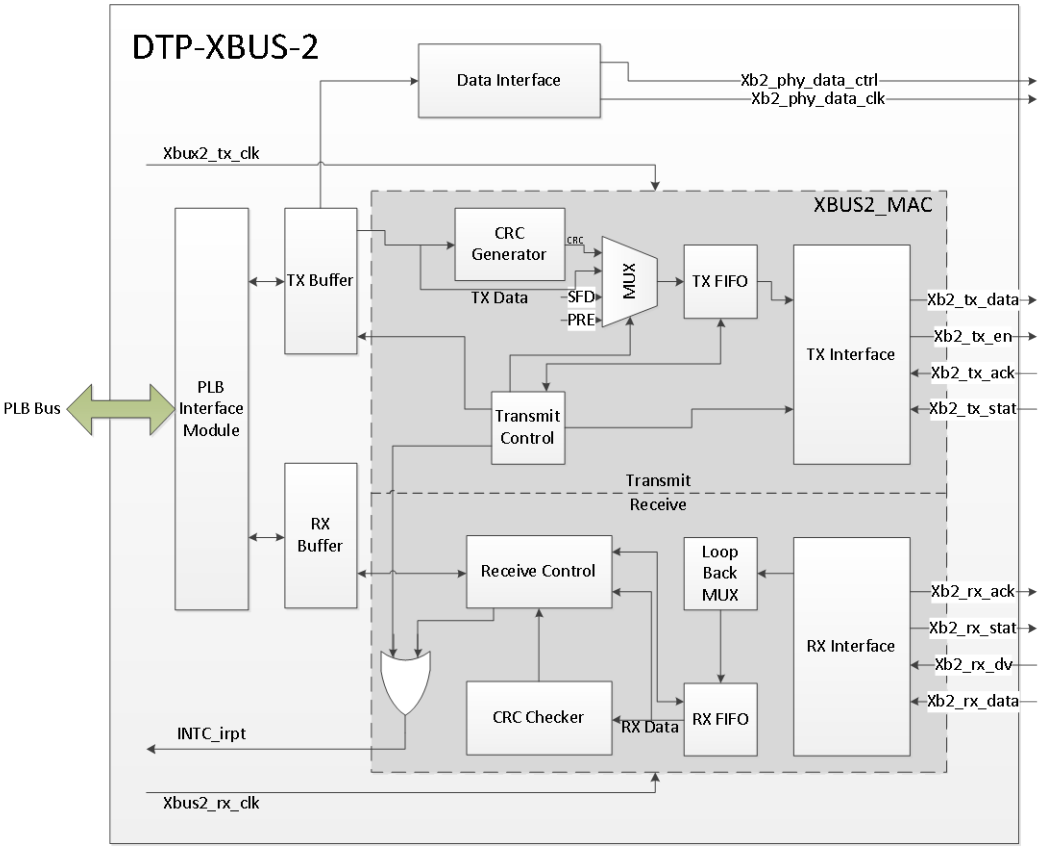

Figure B.1: DTP-XBUS-2 Top-level illustration

The implementation of all components in the figure B.1 will be shown in the following sections.

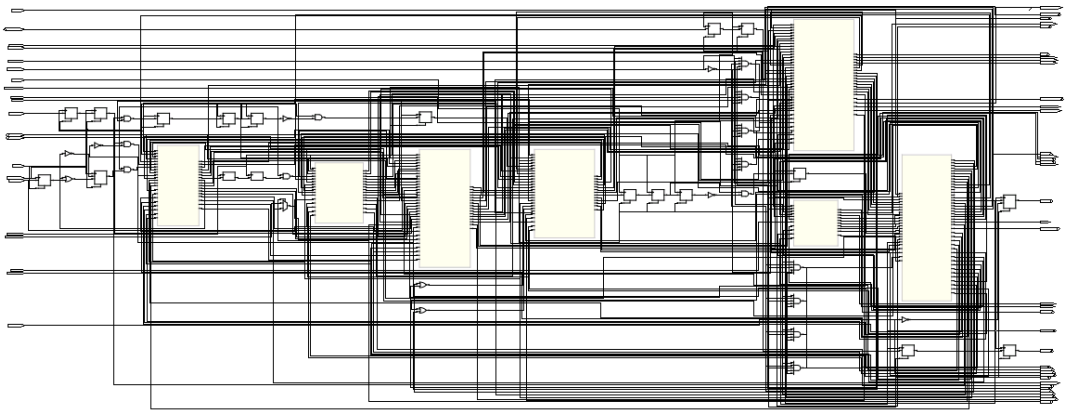

Figure B.2: Synthesized DTP-XBUS-2.

Figure B.2 shows the complete implementation of the DTP-XBUS-2.

The transmission interface is as shown in the figures (Figure B.3 and Figure B.4) below.

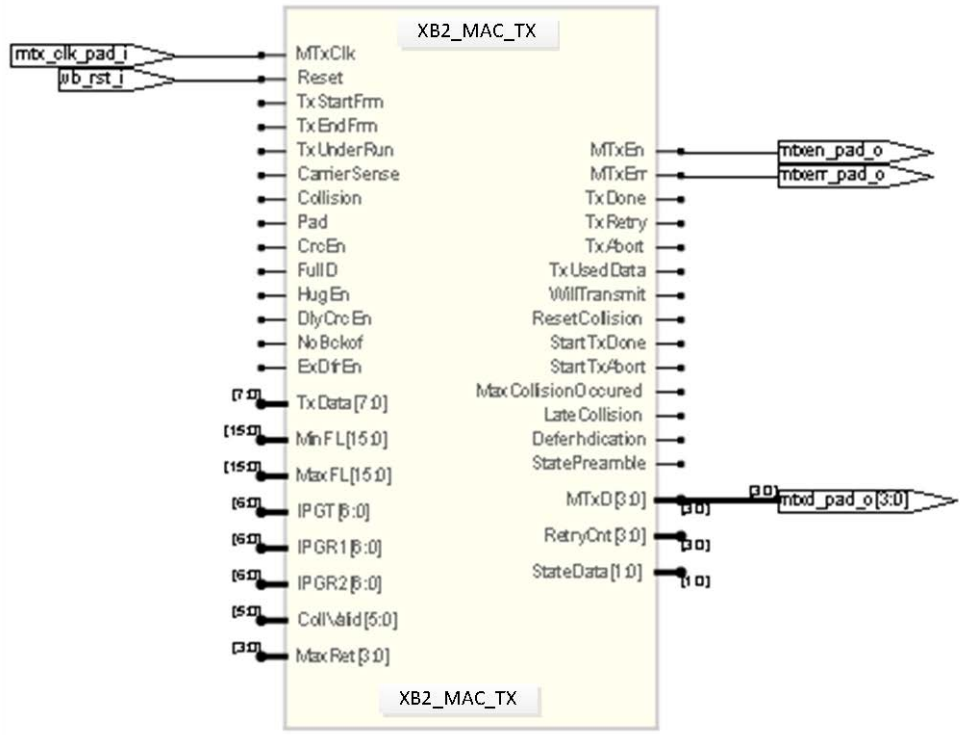

Figure B.3: DTP-XBUS-2 Data Transmitter implementation

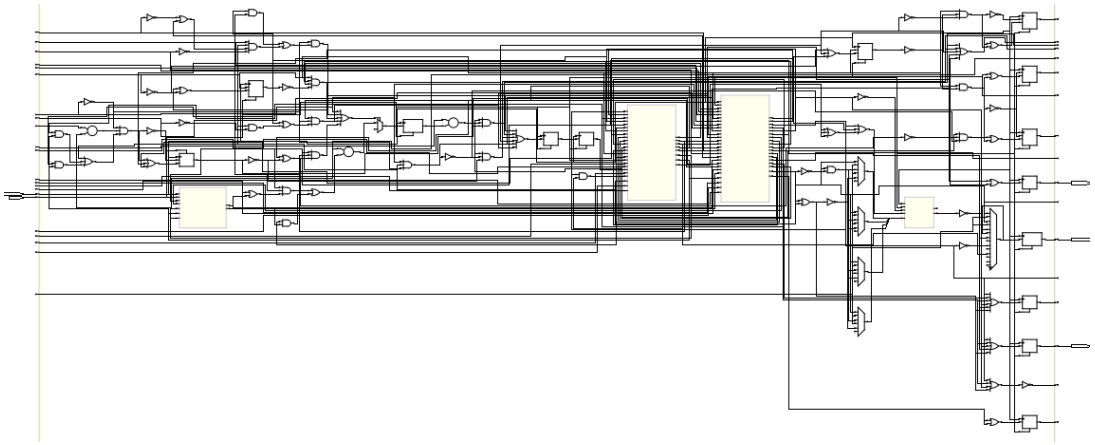

Figure B.4: Synthesized DTP-XBUS-2 Data Transmitter implementation

DTP-XBUS-2 receiver implementation (Figure B.5 and Figure B.6)

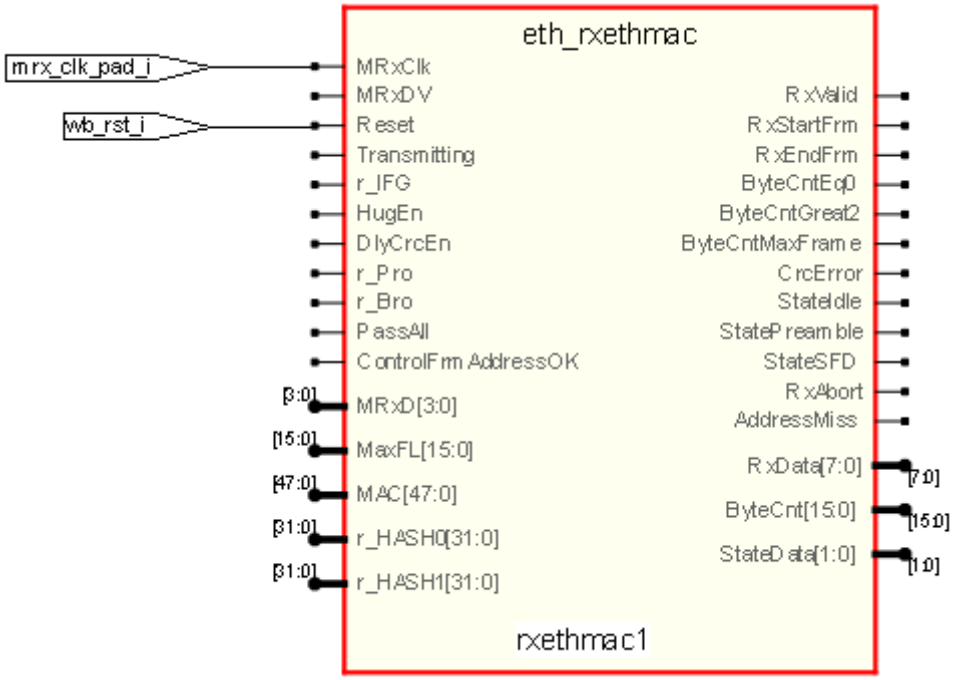

Figure B.5: DTP-XBUS-2 Receiver implementation

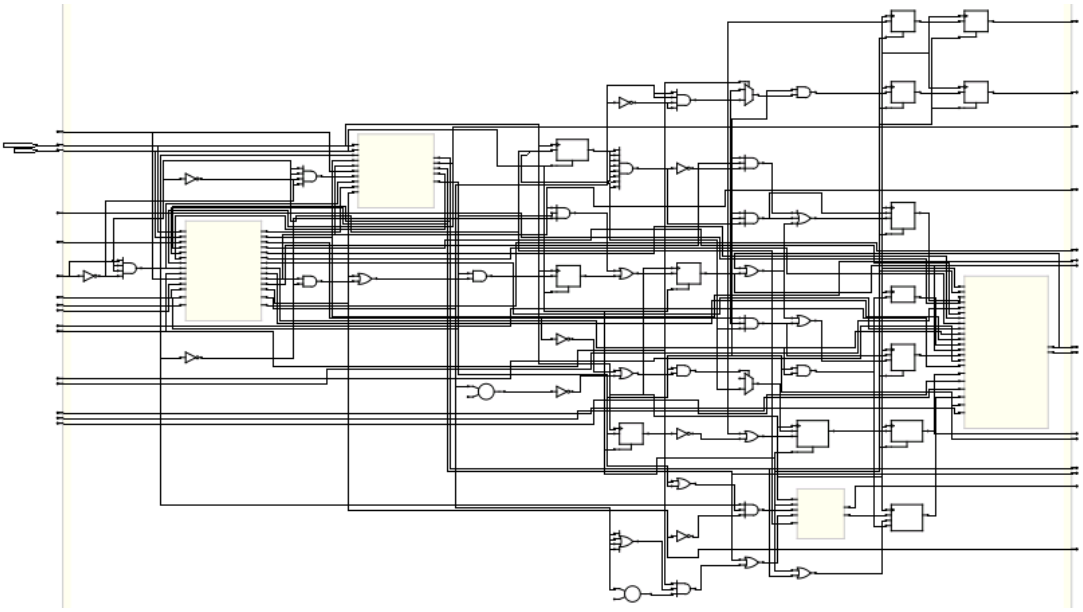

Figure B.6: Synthesized DTP-XBUS-2 receiver.

DTP-XBUS-2 CRC implementation (Figure B.7 and Figure B.8)

Figure B.7: DTP-XBUS-2 CRC

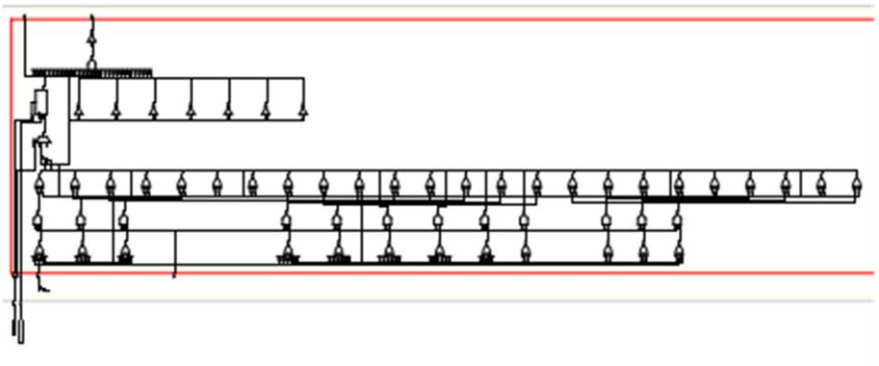

Figure B.8: Synthesized DTP-XBUS-2 CRC

DTP-XBUS-2 Transmit Control Implementation (Figure B.9 and Figure B.10)

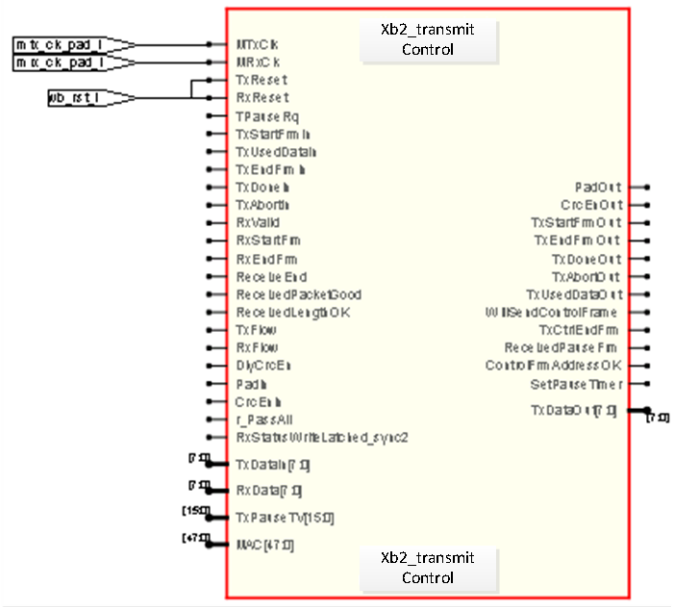

Figure B.9: DTP-XBUS-2 Transmit control

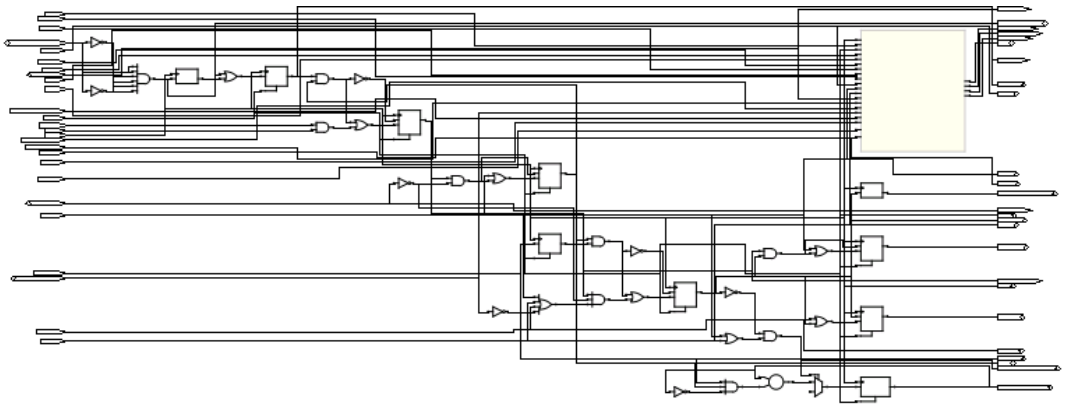

Figure B.10: Synthesized DTP-XBUS-2 Transmit Control

DTP-XBUS-2 Receive Control Implementation (Figure B.11 and Figure B.12)

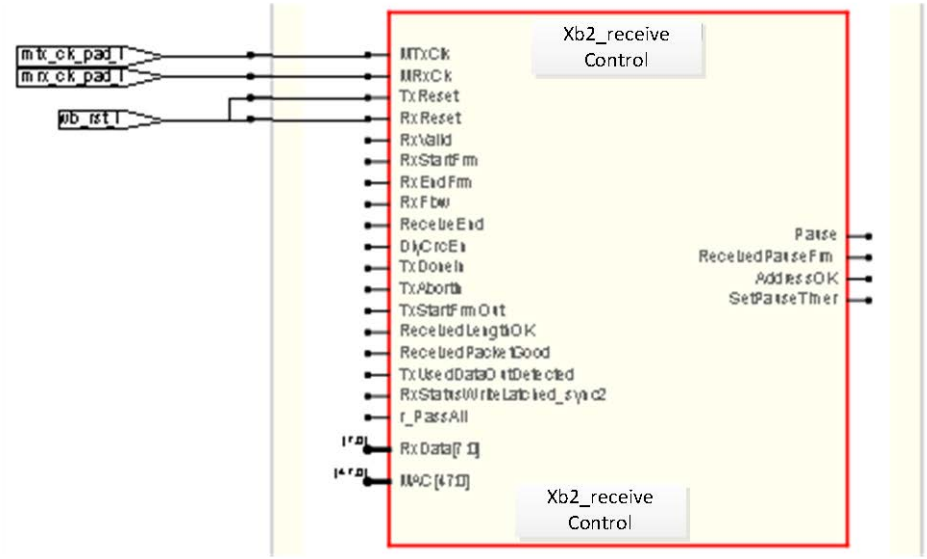

Figure B.11: DTP-XBUS-2 Receive control

Figure B.12: Synthesized DTP-XBUS-2 Receive Control

DTP-XBUS-2 CRC Checker implementation (Figure B.13 and Figure B.14)

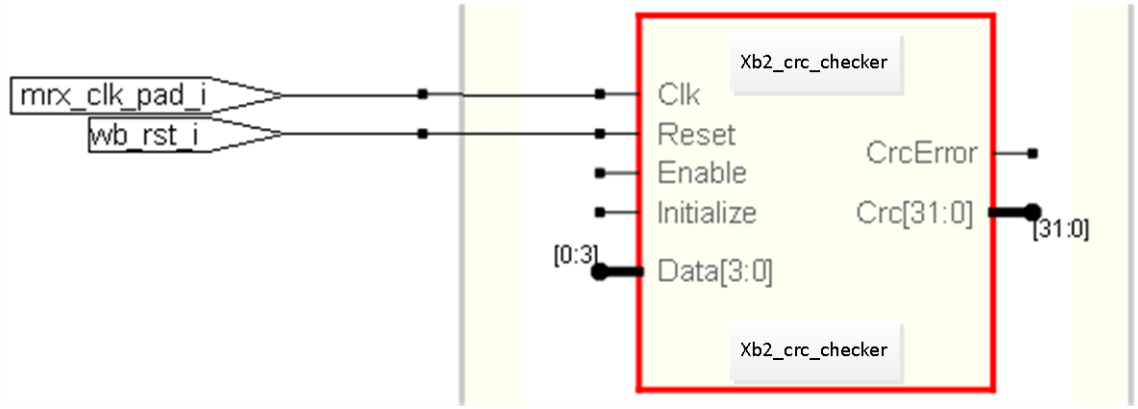

Figure B.13: DTP-XBUS-2 CRC Checker

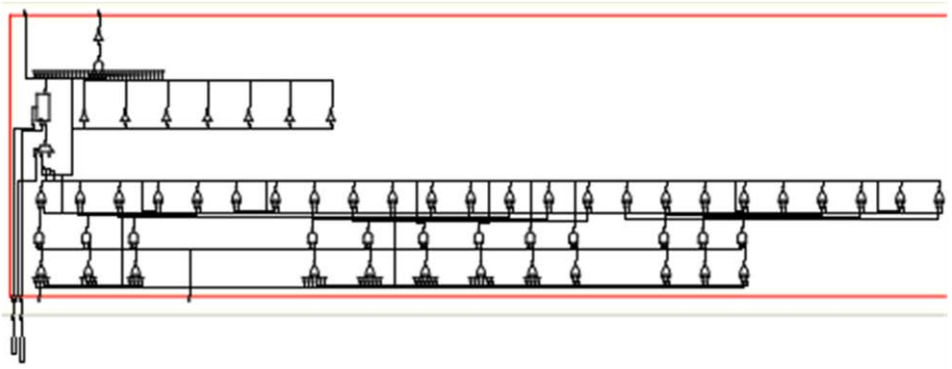

Figure B.14: Synthesized DTP-XBUS-2 CRC Checker

DTP-XBUS-2 Data Interface implementation (Figure B.15 and Figure B.16)

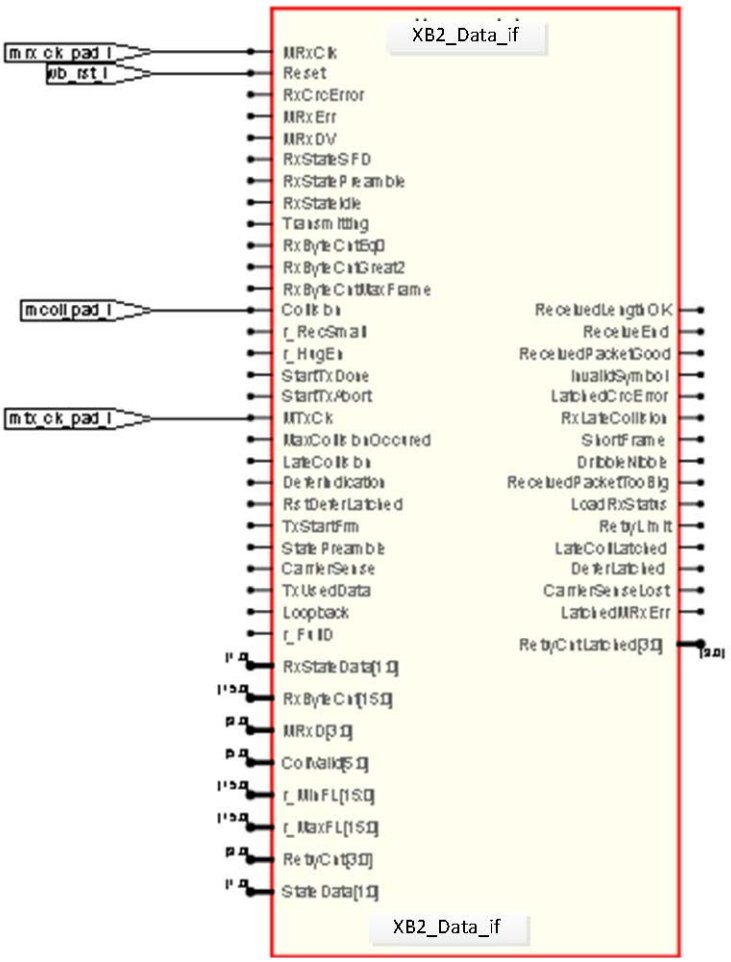

Figure B.15: DTP-XBUS-2 Data Interface

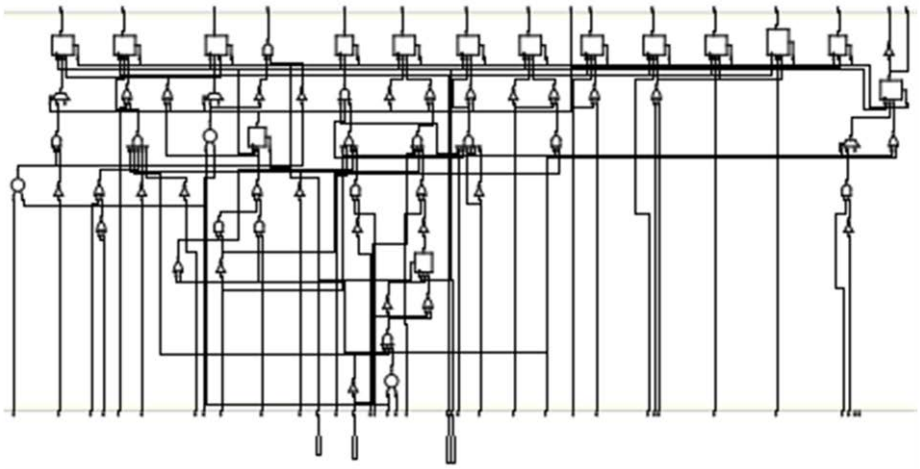

Figure B.16: Synthesized DTP-XBUS-2 Data Interface

DTP-XBUS-2 FPGA implementation (Figure B.17)

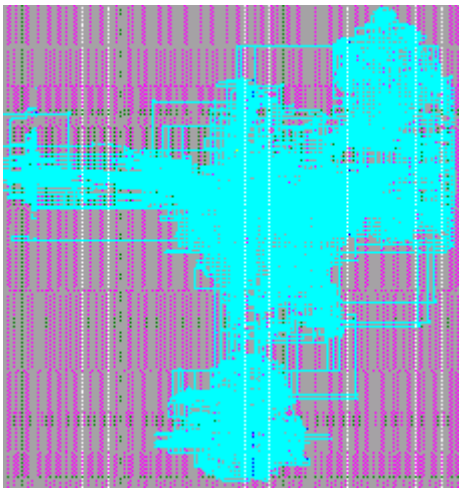

Figure B.17: DTP-XBUS-2 implemented in ML505 Virtex-5 FPGA

Clock Strip Analysis C

This section will describe the setup for clock strip analysis.

In this analysis, the components are as listed below:

a) Air box (Figure C.1)

b) Four Copper Nets (Figure C.2, Figure C.3, Figure C.4 and Figure C.5)

c) FR4 epoxy

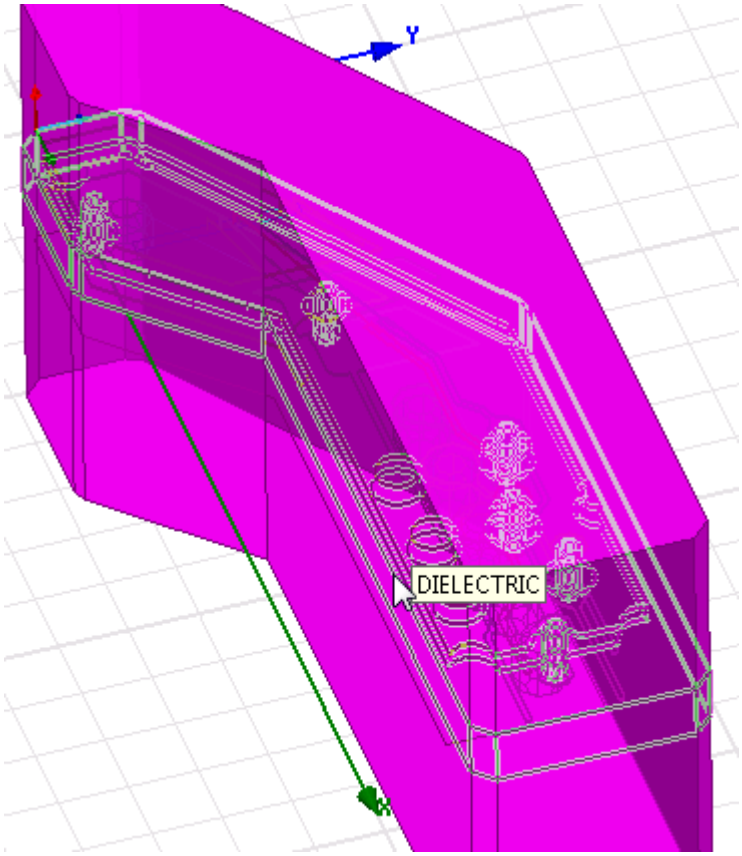

Figure C.1: Air-box setup

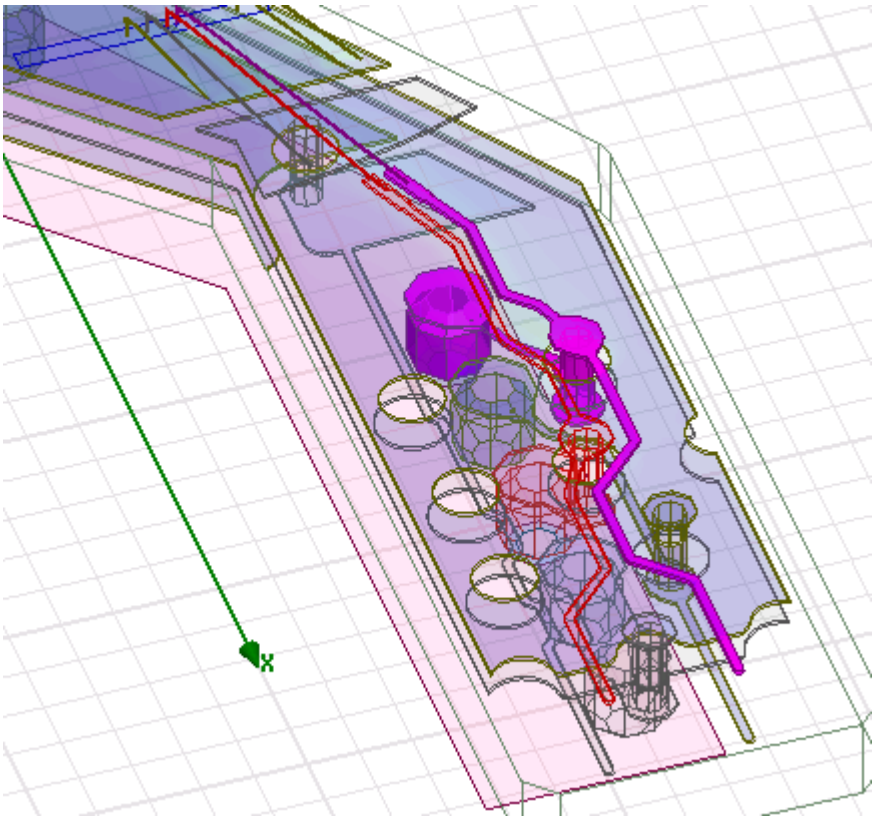

Figure C.2: Copper Net179 setup

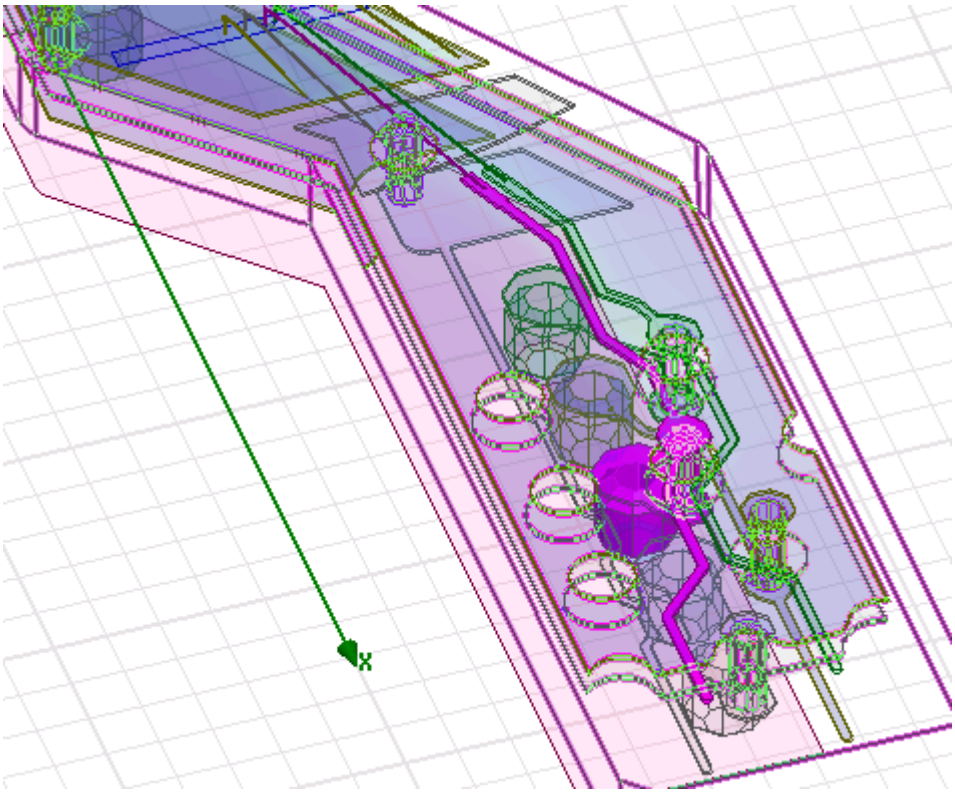

Figure C.3: Copper Net178 setup

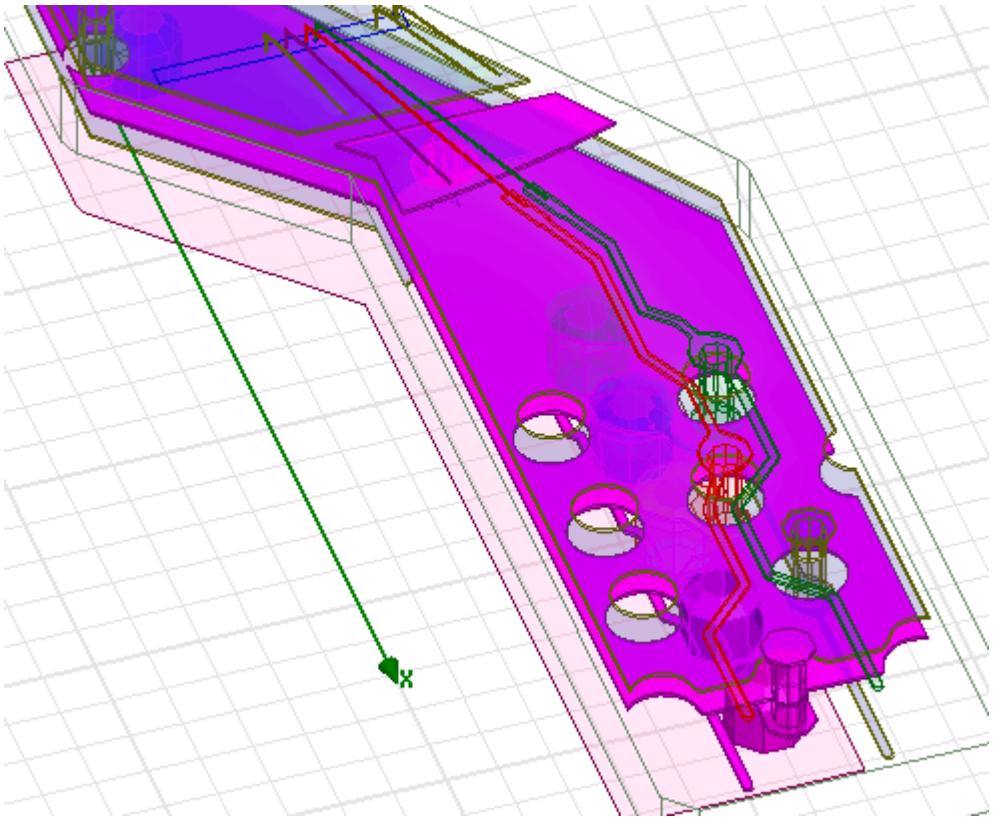

Figure C.4: Copper plane 2 Setup

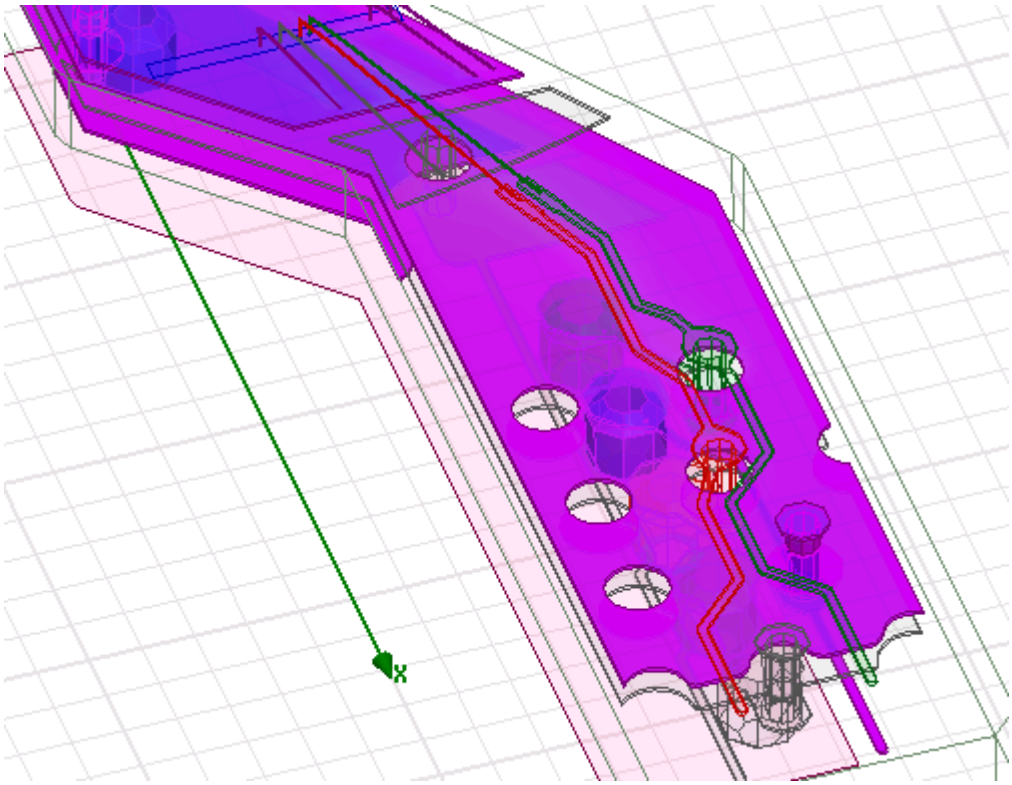

Figure C.5: Copper plane 1 setup

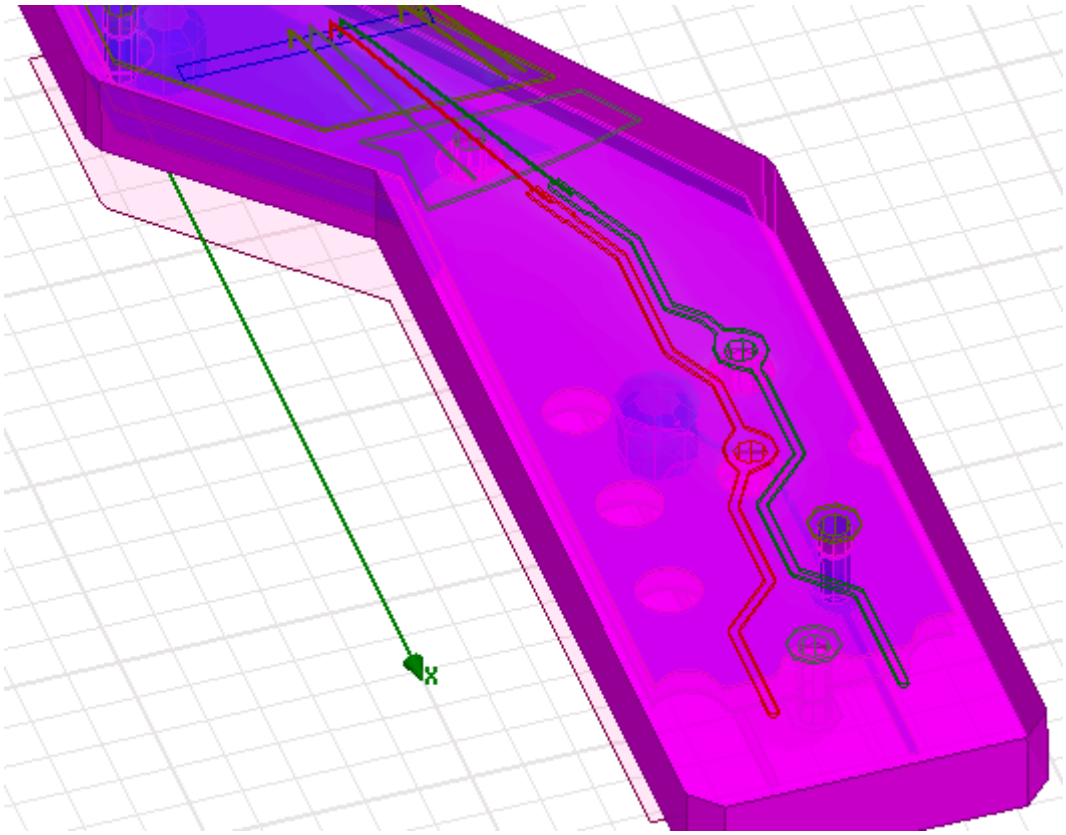

Figure C.6: FR4 Epoxy setup

Similarly, the clock strip analysis for package connectors contain the following components

a) Vacuum box (Figure C.7)

b) Modified Epoxy (Figure C.8)

c) FR4 Epoxy (Figure C.9 and Figure C.10)

d) Copper connectors and layers (Figure C.11, Figure C.12 and Figure C.13)

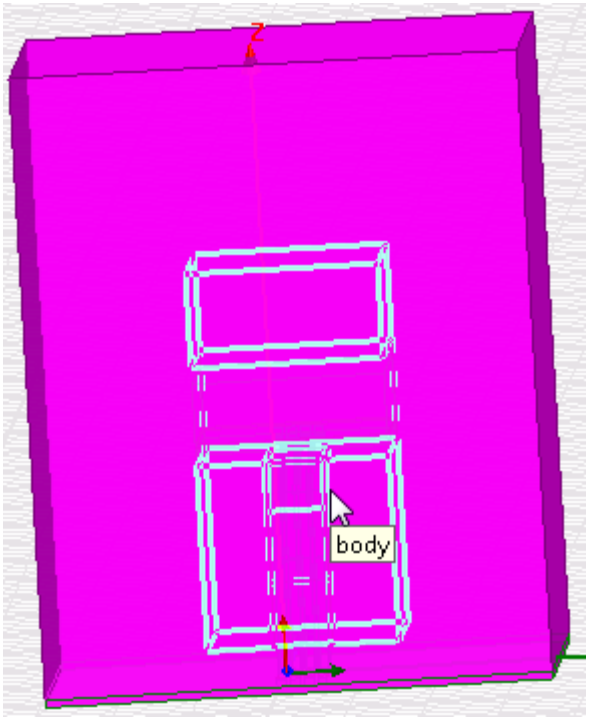

Figure C.7: Vacuum box setup

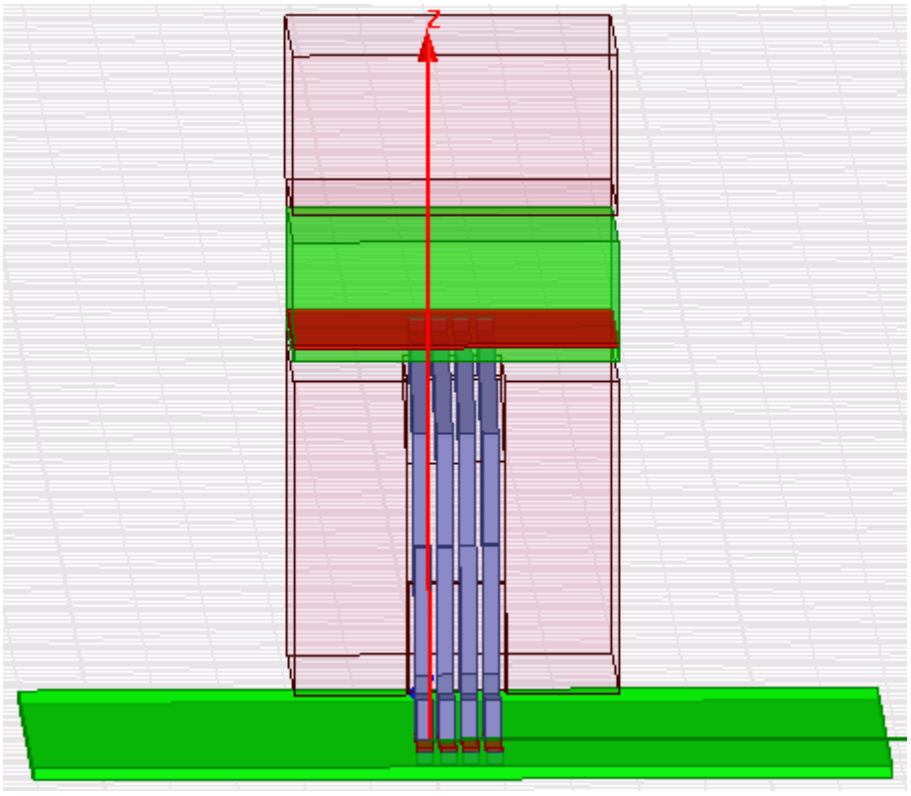

Figure C.8: Modified Epoxy

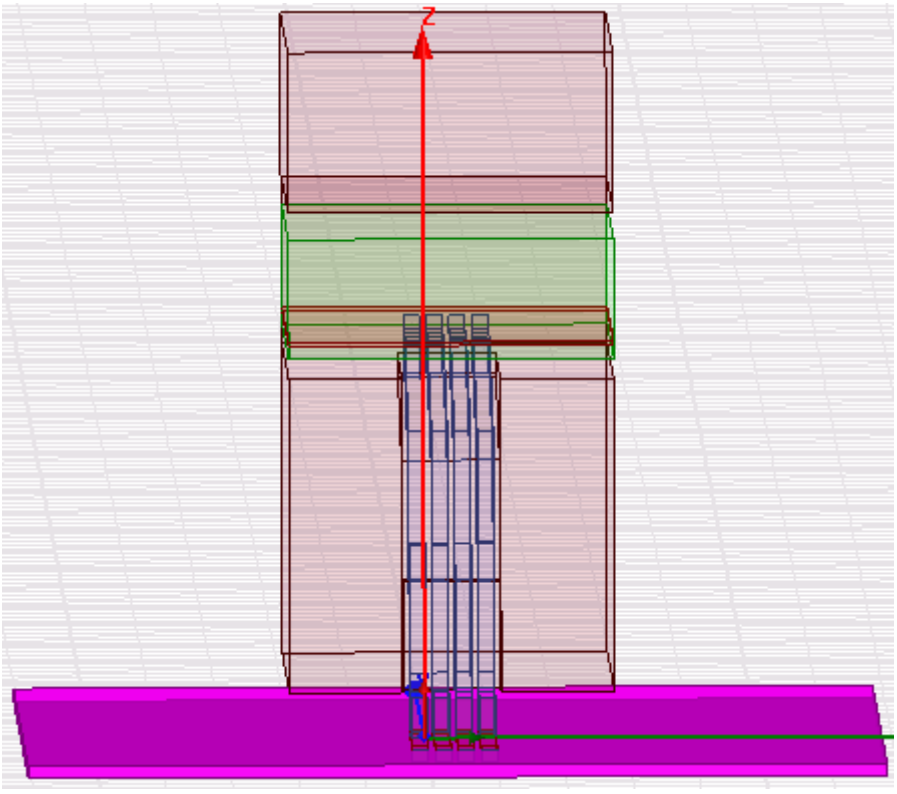

Figure C.9: Board 1 FR4 Epoxy

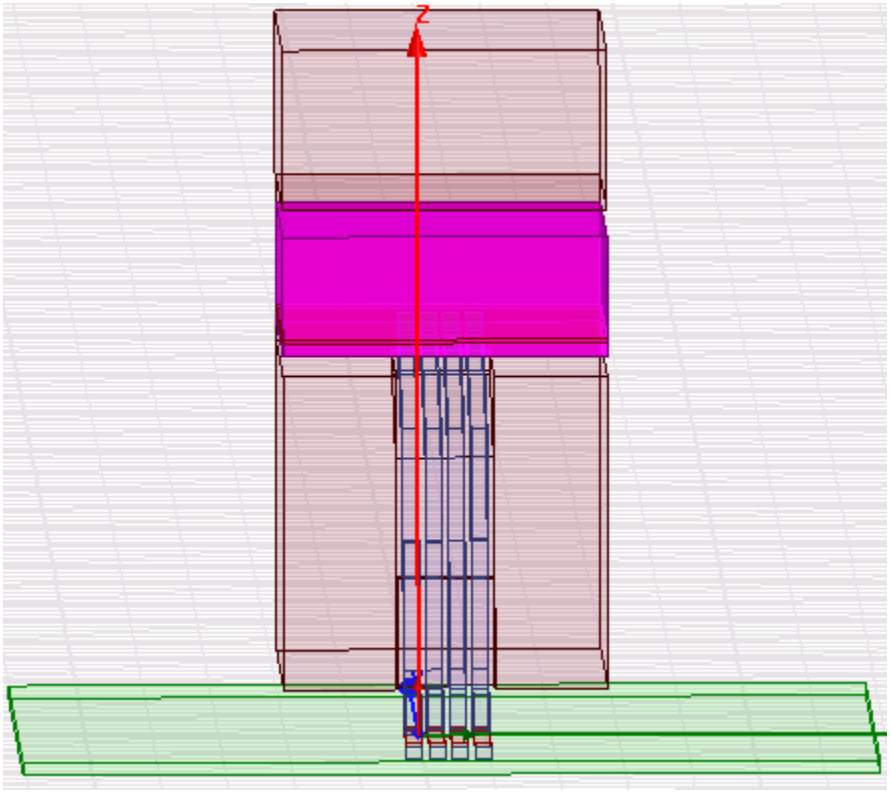

Figure C.10: Board 2 FR4 Epoxy

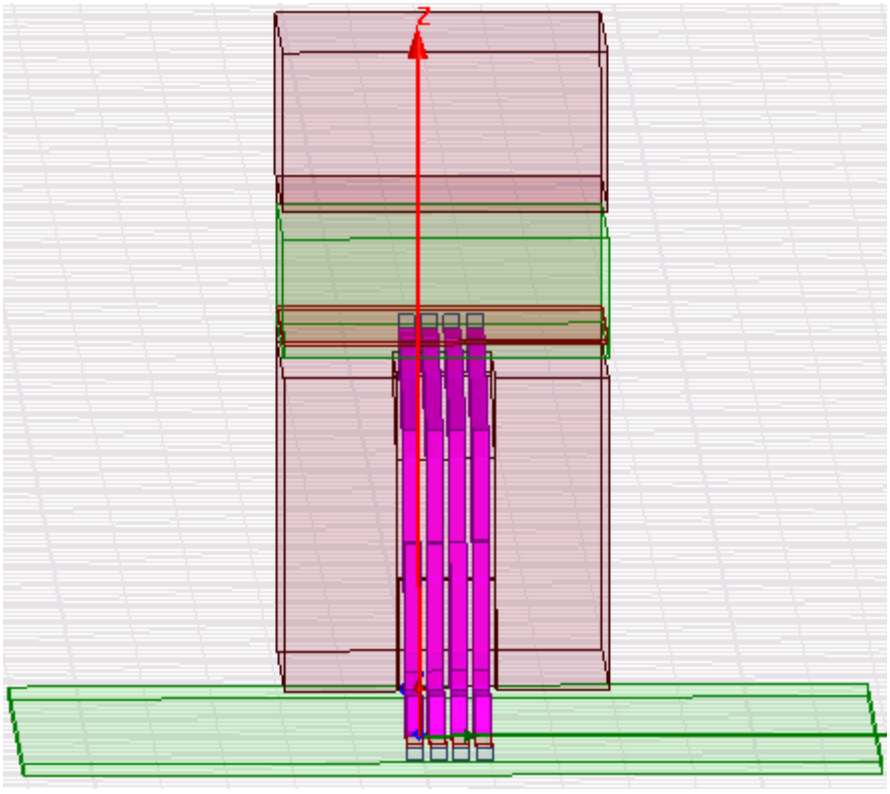

Figure C.11: Copper connectors setup

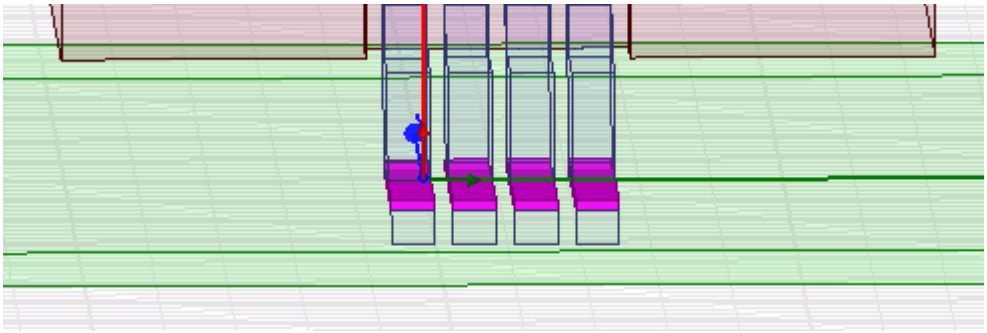

Figure C.12: Copper pads setup

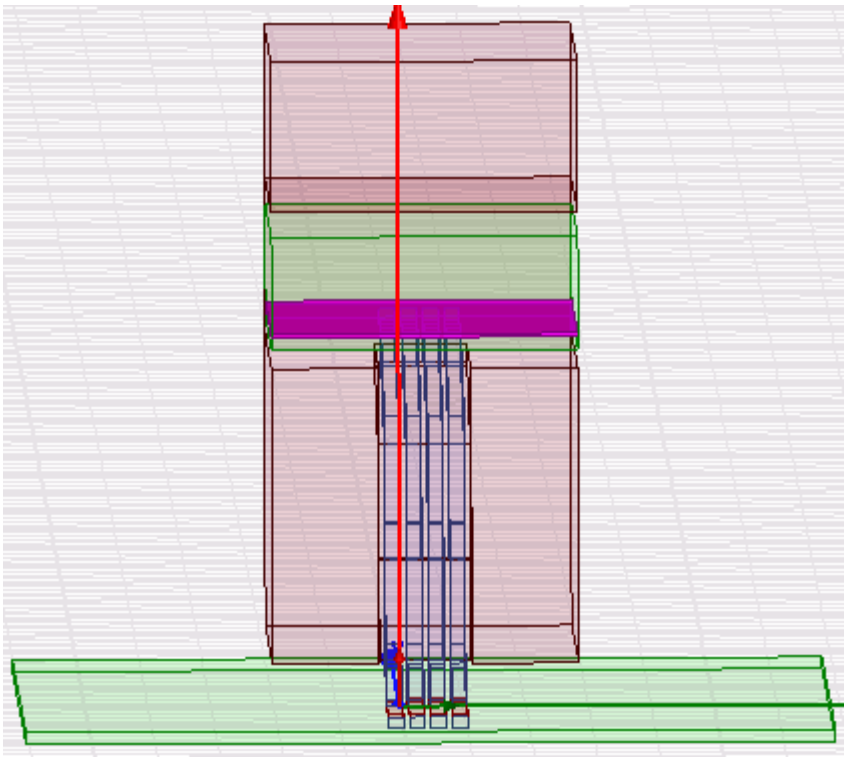

Figure C.13: Ground plane setup

Linker Script D

```
/*************** LinkerScript to set-up the Memory-map ****************/
/*--------startup .ld--------*/
/ *
* Setup the memory map of the Code .
* stack grows down from high memory .
* ------------------------------------------------------------------------
* The .text section - contains instructions
* The .data section - contains static initialized data
* The .rdata section - contains static constant data
* The .bss section - contains uninitialized data
* The .ctor section - contains addresses of global constructors
* The .dtor section - contains addresses of global destructors
* The .stabs section - part of  the debug symbol table
* The .stabstr section - part of  the debug symbol table
* ------------------------------------------------------------------------
* The memory map look like this :
* +--------------------+ <- S t a r t of ROM
* | Interrupt Tabl e |
* +--------------------+ <- 0 x100
* |. t e x t |
* |_ s t e x t |
* |* . t e x t |
* |_ e t e x t |
* +--------------------+ <- i n i t i a l i z e d d a t a goe s h e r e
* |. d a t a |
* |_ s d a t a |
* |* . d a t a |
* |_ s d a t a |
* +--------------------+ <- t h e c t o r and d t o r l i s t s a r e f o r
* |. r d a t a | C++ s u p p o r t ( i f r e q u i e d )
* |* . r d a t a |
* | |
* +--------------------+ <- S t a r t of RAM
* | | s t a r t of bs s , c l e a r e d by c r t 0
* |. b s s | s t a r t of heap
* |_ _ b s s _ s t a r t |
* | _end |
* +--------------------+
* . .
* . .
* . .
* |_ _ s t a c k |
* +--------------------+ <- t o p of s t a c k
* /
STACKSIZE = 0x100 ;
OFFSET = 0x0 ;
/ * The n e x t l i n e i n t h e s c r i p t g i v e s a v a l u e t o t h e l i n k e r symbol _ _ s t a c k . * /
PROVIDE ( _ _ s t a c k = ADDR( . b s s ) + SIZEOF ( . b s s ) + STACKSIZE + OFFSET) ;
PROVIDE ( _ _ c o p y _ s t a r t = _ c o p y _ s t a r t ) ;
PROVIDE ( __copy_end = _copy_end ) ;
PROVIDE ( __copy_adr = _copy_adr ) ;

MEMORY
{
        rom ( rx ) : ORIGIN = 0 x00000000 , LENGTH = 0 x000f0000
        ram ( rwx ) : ORIGIN = 0 xf0000000 , LENGTH = 0 x000f0000
}
SECTIONS
{
        . t e x t 0 x100 :
        {
                _ s t e x t = . ;
                * ( . t e x t )
                _ e t e x t = . ;
        } > rom
/ *
All initialized data sections go in t e RAM.
* /
        . d a t a : {
                _ c o p y _ s t a r t = . ;
```

```
                _sdata = .;
                *(.data)
                _edata = .;
        } > rom
        .rdata :
        {
                *(.rdata)
                _copy_end = .;
                __CTOR_LIST__ = .;
                LONG( (__CTOR_END__ - __CTOR_LIST__) / 4 - 2)
                *(.ctors)
                LONG( 0 )
                __CTOR_END__ = .;
                __DTOR_LIST__ = .;
                LONG( (__DTOR_END__ - __DTOR_LIST__) / 4 - 2)
                *(.dtors)
                LONG( 0 )
                __DTOR_END__ = .;
        } > rom

        .bss (NOLOAD) :
        {
                _copy_adr = .;
                . = ( SIZEOF (.data) + SIZEOF (.rdata) );
                __bss_start = .;
                *(.bss)
                *(COMMON)
                end = ALIGN(0x2);

                _end = ALIGN(0x2);
        } > ram

        .stab 0 (NOLOAD) :
        {
                [.stab]
        }
        .stabstr 0 (NOLOAD) :
        {
                [.stabstr]
        }
}
```

Startup Script E

```
/*--------Startup Script to include explicit initialization code--------*/
/*--------startup.S--------*/
.extern __stack
.extern __copy_start
.extern __copy_end
.extern __copy_adr
/* Core jumps here at start and reset */
_stext:
/* Stack initialization */
|. movhi r1, hi(__stack)
|. ori r1, r1, lo(__stack)
_mem_data_copy:
|. movhi r3, hi(__copy_adr)
|. ori r3, r3, lo(__copy_adr)
|. movhi r4, hi(__copy_start)
|. ori r4, r4, lo(__copy_start)
|. movhi r5, hi(__copy_end)
|. ori r5, r5, lo(__copy_end)
|. sub r5, r5, r4
|. sfeqi r5, 0
|. bf _jump_main
|. nop
_mem_data_loop:
|. lwz r6, 0(r4)
|. sw 0(r3), r6
|. addi r3, r3, 4
|. addi r4, r4, 4
|. addi r5, r5, -4
|. sfgtsi r5, 0
|. bf _mem_data_loop
|. nop
/* Jump to Main */
_jump_main:
|. movhi r2, hi(_main)
|. ori r2, r2, lo(_main)

|. jr r2
|. nop
```

```
int simICS_ init (const char * config_file, const char * image _ file , void * class _ ptr,unsigned long int (* upr ) (
        void * class_ ptr , unsigned long int addr , unsigned long int mask ) ,void (* upw) ( void * class _ ptr ,
        unsigned long int addr , unsigned long int mask , unsigned long int wdata ) ,
        void (* upcpustatus ) ( void * class_ptr , void * cpu_statusPtr ) ) ;

/ * *** libtoplevel.c *** * /
config.ext.write_up_cpustatus = upcpustatus
struct config
{
        struct
        { / * External linkage for SystemC * /
                void * class_ptr ;
                unsigned long int (* read_up ) ( void * class_ptr,
                uns igned long int addr , unsigned long int mask ) ;
                void (* write_up ) ( void * class_ptr, uns igned long int addr ,
                uns igned long int mask , uns igned long int wdata ) ;
                void (* write_up_cpustatus ) ( void * class_ptr, void * cpu_statusPtr ) ;
        } e x t ;
        struct ext_access_cpu_status
        {
                void (* write_cpustatus_up) ( void *) ;
        } ;

        extern struct ext_access_cpu_statuscpu status_up ;
/ * *** sim-c o n f i g . c *** * /
        a.struct ext_access_cpu_statuscpustatus_up ;
        b.config.e x t.write_up_cpustatus = NULL;
/ * *** g e n e r i c . c *** * /
        static void ext_write_cpustatus(void * cpu_statusPtr )
        {
                config.ext.write_up_cpustatus(config.ext.class_ptr , cpu_statusPtr );
        }
/ * b . In g e n e r i c _ s e c _ s t a r t ( ) . * /
        cpustatus_up.write_cpustatus_up=ext_write_cpustatus;
/ * *** e x e c u t e . c *** * /
        cpustatus_up.write_cpustatus_up (&cpu_state) ;

/ * ******* e x e cgen . c ****** * /
        case 0 x11 :
                i f ( ( insn & 0xfc000000 ) == 0x44000000 ) {
                {
                        uorreg_ta;
                        / * Number o f ope rands : 1 * /
                        a = ( insn >> 11) & 0x1f ;
                        #define SET_PARAM0(val) cpu_state.reg [ a ] = v a l
                        #define PARAM0 cpu_state.reg [ a ]
                        { / * " l _ j r " * /
                                cpu_state.pc_delay = PARAM0;
                                next_delay_insn = 1 ;
                                if ( config.sim.profile )
                                        fprintf(runtime.sim.fprof , "-%08l lX %"PRIxADDR" \ n ", runtime .
                                sim.cycles , cpu_state.pc_delay) ;
                        }
                        #undef SET_PARAM
                        #undef PARAM0
                        if ( do_stats )
                        {
                                c u r r e n t ->i n s n _ i n d e x = 104; / * " l . j r " * /
                                a n a l y s i s ( c u r r e n t ) ;
                        }
                }
        }
        e l s e
         {
/ * I n v a l i d i n s n * /
                {
                        l _ i n v a l i d ( ) ;
                        i f ( d o _ s t a t s )
                        {
                                c u r r e n t ->i n s n _ i n d e x = -1; / * "???" * /
```

```
                                        analysis(current);
                                }
                        }
                }
                break;
L. JALR:
                case 0x12:
                        if ((insn & 0xfc000000) == 0x48000000)
                        {
                                {
                                        uorreg_ta;
                                        /* Number of operands: 1 */
                                        a = (insn >> 11) & 0x1f;
                                        #define SET_PARAM0(val) cpu_state.reg[a] = val
                                        #define PARAM0 cpu_state.reg[a]
                                        {/* "l_jalr" */
                                                cpu_state.pc_delay = PARAM0;
                                                setsim_reg (LINK_REGNO, cpu_state.pc + 8);
                                                next_delay_insn = 1;
                                        }

                                        #undef SET_PARAM
                                        #undef PARAM0
                                        if (do_stats)
                                        {
                                                current->insn_index = 105; /* "l.jalr" */
                                                analysis (current);
                                        }
                                }
                        }
                        else
                        {
/* Invalid insn */
                                {
                                        l_invalid();
                                        if(do_stats){
                                                current->insn_index = -1; /* "???" */
                                                analysis(current);
                                        }
                                }
                        }
                        break;

/* ********** exe cgen.c ********** */
                case 0x30:
                        if ((insn & 0xfc000000) == 0xc0000000)
                        {
                                {
                                        uorreg_ta, b, c;
                                        /* Number of operands: 3 */
                                        a = (insn >> 16) & 0x1f;
                                        #define SET_PARAM0(val) cpu_state.reg[a] = val
                                        #define PARAM0 cpu_state.reg[a]
                                        b = (insn >> 11) & 0x1f;
                                        #define PARAM1 cpu_state.reg[b]
                                        c = (insn >> 0) & 0x7ff;
                                        c |= ((insn >> 21) & 0x1f) << 11;
                                        #define PARAM2 c
                                        {/* "l_mtspr" */
                                                uint16_t regno = PARAM0 + PARAM2;
                                                uorreg_tvalue = PARAM1;
                                                if (cpu_state.sprs[SPR_SR] & SPR_SR_SM)
                                                        mtspr (regno, value);
                                                else
                                                {
                                                        PRINTF ("WARNING: trying to write SPR while
                                                SR[SUPV] is cleared.\n");
                                                        sim_done();
                                                }
                                        }
#undef SET_PARAM
#undef PARAM0
#undef PARAM1
#undef PARAM2
                                        if(do_stats){
```

```
                                        current->insn_index = 139; /* "l. mt spr " */
                                        analysis(current);
                                }
                        }
                }
                else
                {
/*Invalidinsn*/
                        {
                                l_invalid();
                                if(do_stats)
                                {
                                        current->insn_index = -1; /* "???" */
                                        analysis(current);
                                }
                        }
                }
                break;
```

Bibliography

[1] Akira Hatanaka and Nader Bagherzadeh. “Scheduling Techniques for Multi-Core Architectures”. In Proceedings of the 6th International Conference on Information Technology: New Generations, Las Vegas, USA, April 2009.

[2] Rantala, V., Lehtonen, T., Liljeberg, P., Plosila, J., "Multi network interface architectures for fault tolerant Network-on-Chip", *Signals, Circuits and Systems, 2009. ISSCS 2009. International Symposium on*, On page(s): 1 - 4, Volume: Issue: , 9-10 July 2009

[3] S. Bataineh, T.-Y. Hsiung, and T. G. Robertazzi. “Closed form solutions for bus and tree networks of processors load sharing a divisible job”. *IEEE Trans. Comput.*, 43(10):1184–1196, Oct. 1994.

[4] Veeravalli B. And Wong Han Min. “Scheduling divisible loads on heterogeneous linear daisy chain networks with arbitrary processor release time”. Parallel and Distributed Systems, IEEE Transactions, 15(3) 273-288, March 2004.

[5] J. Sohn and T.G. Robertazzi. “A Multi-Job Load Sharing Strategy for Divisible Jobs on Bus Network”. CEAS Technical Report 665, State University of New York at Stony Brook, Apr. 1993.

[6] Sham, M.X. ; Chen, Y.C. ; Leung, L.W. ; Lin, J.R. ; Chung, T. “Challenges and Opportunities in System-in-Package (SiP) Business”. Electronic Packaging Technology, 2006. ICEPT’06. 7th International Conference.

[7] Suganya, K. ; Nagarajan, V. “Efficient run-time task allocation in reconfigurable multiprocessor System-on-Chip with Network-on-Chip”. Computer, Communication and Electrical Technology (ICCCET), 2011 International Conference.

[8] Kyeong Keol Ryu, Mooney V.J.I.I.I. “Automated bus generation for multiprocessor SoC Design”. Design, Automation and Test in Europe Conference and Exhibition, 2003.

[9] Kyeong Keol Ryu, Eung Shin, Mooney V.J. “A comparison of five different multiprocessor SoC bus architectures”. Digital Systems, Design, 2001. Proceedings. Euromicro Symposium.

[10] Meng-Chou Chang, Feipei Lai. “Efficient exploitation of instruction-level parallelism for superscalar processors by the conjugate register file scheme”. IEEE Transactions on Computers, 45(3) 278-293, March 1996.

[11] Soo Ho Chang, Soo Dong Kim, “Reuse-based Methodology in Developing System-on-Chip (SoC)”. Software Engineering Research, Management and Applications, 2006.

Fourth International Conference.

[12] Beaumont S.P. “The SoC Challenge”. Electronics and Communication Engineering Journal, 13(6) 234-235, Dec 2001.

[13] Pizhou Ye, Chaodong Ling. “A RISC CPU IP Core”. Anti-counterfeiting, Security and Identification, 2008, ASID 2008, 2nd International Conference, 356-359, 20-23 Aug, 2008.

[14] You-Sung Chang, Bong-Il Park, In-Cheol Park, Chong-Min Kyung. “Customization of a CISC processor core for low-power applications”. Computer Design, 1999. (ICCD ‘99) International Conference, 152-157, 10 Oct 1999 to 13 Oct 1999.

[15] Zhao Junxia, Zhen Botao, Liu Hongjun. “A median filter FPGA with Harvard Architecture”. International Conference on Information Science and Technology (ICIST), 2011. 50-51, 26-28 March 2011.

[16] Thimbleby H. “Modes, WYSIWYG and the von Neumann bottleneck”. IEE Colloquium on Formal Methods and Human Computer Interaction. 4/1 – 4/5, 22 Feb 1988.

[17] Mayer-Lindenberg F. “High-Level FPGA Programming through Mapping Process Networks to FPGA Resources”. International Conference on Reconfigurable Computing and FPGAs, 2009 ReConFig ’09. 302-307, 9-11 Dec. 2009.

[18] Enright Jerger N., Li-Shiuan Peh, Lipasti M. “Circuit-Switched Coherence”. Second ACM/IEEE International Symposium on Networks-on-Chip, 2008. NoCS 2008. 193-202, 7-10 April 2008.

[19] Cournier A., Dubois S., Villain V. “A snap-stabilizing point-to-point communication protocol in message-switched networks”. IEEE International Symposium on Parallel & Distributed Processing, 2009. IPDPS 2009. 1-11, 23-29 May 2009

[20] Ito T., Sampei S., Morinaga N. “A wireless packet transmission with adaptive processing gain and transmitter power control scheme for circuit-switched and packet-switched modes integrated DS/CDMA systems”. 1999 IEEE 49th Vehicular Technology Conference. (3) 2039-2043, 16 May 1999 to 20 May 1999.

[21] Yaobin Wang, Hong An, Jie Yan, Qi Li, Wenting Han, Li Wang, Gu Liu. “Investigation of Factors Impacting Thread-Level Parallelism from Desktop, Multimedia and HPC Applications”. FCST Fourth International Conference on Frontier of Computer Science and Technology, 2009, 27-32, 17-19 Dec 2009.

[22] Hofmann R., Drerup B. “Next generation CoreConnect processor local bus architecture”. 15th Annual IEEE international ASIC/SOC Conference, 2002, 221-225, 25-28 Sept. 2002.

[23] Lei Li, Siyuan Liu, Mingyu Chen, Jianping Fan. “Grid Memory Service Architecture for High Performance Computing”. GCC’08 Seventh International Conference on Grid

and Coorperative Computing, 2008. 22-27, 24-26 Oct. 2008.

[24] Shrivastav A., Tomar G.S, Singh A.K. “Performance Comparison of AMBA Bus-Based System-on-Chip Communication Protocol”. 2011 International Conference on Communication Systems and Network Technologies (CSNT), 449-454, 3-5 June 2011.

[25] Kuo J.B, Liao H.J. “A BiCMOS tristate buffer for high-speed microprocessor VLSI”. Fourth Annual IEEE International ASIC Conference and Exhibit, 1991 Proceedings. P10- 5/1-4. 23 Sep 1991 to 27 Sep 1991.

[26] August N. “A Robust and Efficient Pre-Silicon Validation Environment for Mixed-Signal Circuits on Intel’s Test Chips”. ISQED 2008. 9th International Symposium on Quality Electronic Design, 2008. 423-428, 17-19 March 2008.

[27] Hudec J. “Processor Functional Test Generation – Some Results with Using of Genetic Algorithms”. 2011 2nd Eastern European Regional Conference on the Engineering of Computer Based Systems (ECBS-EERC), 159-160, 5-6 Sept. 2011.

[28] Topaloglu R.O., Orailoglu A. “A DFT approach for diagnosis and process variation-aware structural test of thermometer coded current steering DACs”. Proceedings 42nd Design Automation Conference, 2005. 851-856, 13-17 June 2005.

[29] Mollick E. “Establishing Moore’s Law”. IEEE Annals of the History of Computing, (3) 28 62-75, July-Sept. 2006.

[30] Chia-Yuan Hsieh, Chuen-De Wang, Kun-You Lin, Tzong-Lin Wu. “A Power Bus with Multiple via Ground Surface Perturbation Lattice for Broadband Noise Isolation: Modelling and Application in RF-SiP”. IEEE Transactions on Advanced Packaging, 33(3) 582-591, Aug 2010.

[31] Goetz M. “System on chip design methodology applied to system in package architecture”. Electronic Components and Technology Conference, 2002. Proceedings. 52nd. 254-258, August 2002.

[32] Ten Xu, Zhang Linchang. “A method for analyzing and modelling electromagnetic radiation property due to common-mode current in clock strip on PCB”. Asia-Pacific Conference on Environmental Electromagnetics, 2000. CEEM 2000 Proceedings. 111-115, 2000.

[33] Asghari T.A. “PCB Thermal Via Optimization using Design of Experiments”. The tenth Intersociety Conference on Thermal and Thermomechanical Phenomena in Electronics Systems, 2006. ITHERM ’06. 224-228, May 30 2006 to June 2 2006.

[34] Eun Ju Hwang, Woo Kim, Young Hwan Kim. “Impact of process variation on timing characteristics of MTCMOS flip-flops for low-power mobile multimedia applications”. Proceedings of the 2009 12th International Symposium on Integrated Circuits, ISIC’09. 332-335, 14-16 Dec 2009.

[35] Hirano K., Nakatani S., Handa H., Takehara H., “A new composite substrate with

high thermal conductivity for low power modules". IEMT/IMC Symposium, 2nd 1998, 321-326, 15-17 Apr 1998.

[36] Constantinides K., Mutlu O., Austin T. "Online design bug detection: RTL analysis, flexible mechanisms, and evaluation". 41st IEEE/ACM International Symposium on Microarchitecture, 2008. MICRO-41. 282-293, 8-12 Nov. 2008.

[37] Polian I., Miller C., Engelke P., Nopper T., Czutro A., Becker B. "Benchmarking Academic DFT Tools on the OpenSparc Microprocessor". IEEE International Test Conference ITC 2008. p1. 28-30 Oct. 2008.

[38] De Caro D., Mazzocca N., Napoli E., Saggese G.P., Strollo A.G.M. "Test pattern generator for hybrid testing of combinational circuits". The 8th IEEE International Conference on Electronics, Circuits and Systems, 2001. 745-748(2). 02 Sep 2001 to 05 Sep 2001.

[39] Fan Wu, Chun-han Chen, Narang H., "An Efficient Acceleration of Symmetric Key Cryptography Using General Purpose Graphics Processing Unit". 2010 Fourth International Conference on Emerging Security Information Systems and Technologies (SECURWARE). 228-233. 18-25 July 2010