uOttawa

L'Université canadienne
Canada's university

FACULTÉ DES ÉTUDES SUPÉRIEURES ET POSTOCTORALES

uOttawa

L'Université canadienne
Canada's university

FACULTY OF GRADUATE AND POSDOCTORAL STUDIES

Pino G. Dicorato

AUTEUR DE LA THÈSE / AUTHOR OF THESIS

M.A.Sc. (Electrical Engineering)

GRADE / DEGREE

School of Information Technology and Engineering

FACULTÉ, ÉCOLE, DÉPARTEMENT / FACULTY, SCHOOL, DEPARTMENT

Hardware Linked-List Using FPGAs and Optical Central Stage Impairments In A Sectored Packet Switch With An Optical Core (Demonstrator)

TITRE DE LA THÈSE / TITLE OF THESIS

T. Hall

DIRECTEUR (DIRECTRICE) DE LA THÈSE / THESIS SUPERVISOR

CO-DIRECTEUR (CO-DIRECTRICE) DE LA THÈSE / THESIS CO-SUPERVISOR

EXAMINATEURS (EXAMINATRICES) DE LA THÈSE / THESIS EXAMINERS

L. MacEachern

V. Groza

Gary W. Slater

Le Doyen de la Faculté des études supérieures et postdoctorales / Dean of the Faculty of Graduate and Postdoctoral Studies

Hardware Linked-List Using FPGAs And Optical Central Stage Impairments In A Sectored Packet Switch With An Optical Core (Demonstrator)

Pino G. Dicorato

A Thesis submitted to the Faculty of Graduate Studies and Postdoctoral Studies in partial fulfillment of requirement for the degree of

Masters of Applied Science

in Electrical Engineering

Ottawa-Carleton Institute for Electrical Engineering
School of Information Technology and Engineering
University of Ottawa
Ottawa, Ontario, Canada

April 2007

Your file Votre référence
ISBN: 978-0-494-49187-4
Our file Notre référence
ISBN: 978-0-494-49187-4

Abstract

Packet switches play a key role in packet-switched networks and are at the very core of the design in routers. The recent decade has enabled photonics to produce high-speed reliable networks; however within these networks the switching aspect remains mostly within the electrical domain. Opto-electrical and Electro-optical conversion is required for packet headers to be identified in order to forward the packets to their appropriate destination link. Convergence to a more optical solution depends on specific packet-switch architectures.

This thesis presents on two aspects of building a "hardware demonstrator" of a CLOS-like optoelectronic packet switch architecture that consists of two stages of electronic islands (called "sectors") surrounding a photonic central stage. The hardware demonstrator is a scaled-down version of the switch and is intended to "demonstrate" the concepts and workings of the architecture.

The first aspect is a detailed description and simulation of a System-on-Chip (SoC) linked-list memory manager implemented in a Field-Programmable Gate Array (FPGA) that is a building block of a common shared-memory switch that forms an electronic sector.

If the core optical central stage is very far from the input and output sectors or if the switch is scaled to higher port counts, then key optical impairments need to be identified. The second part of this thesis explores the impairments for a wavelength selectable switch used in a cross-connect configuration as a possible switching technology and shows optical measurements for each of the identified impairments.

Acknowledgments

I would like to sincerely thank my supervisor Dr. Trevor Hall for his support, patience and most of all his wisdom in the area of photonics and packet-switching. His motivation for bringing photonics back as a strong topic into the university whilst the industry suffered a downturn is very admirable.

I would definitely like to thank both Majid Kalsi and John Montgomery at Altera Corporation for being very good friends and also providing guidance, technical support, and brainstorming sessions on the design. They are both an inspiration to me in terms of true leaders in their realm of engineering and their mentorship only inspires me to continue to coach others who would need similar assistance in the future. I would also like to thank Jean-Luc Berube, VP of Marketing at Altera to introduce me to his team for their support.

I would also like to thank my parents, family and friends for their support and time for helping me through the little things. Behind the scenes my parents gave me more than I can ever hope for in terms of being patient, and helping me with my chores that freed up my time to complete this thesis. For this I will be forever grateful and can not begin to think on how to repay them.

Table of Contents

List of Tables

List of Figures

List of Figures

List of Acronyms

ASM	Algorithmic State Machine
CAS	Column Access Strobe
CMOS	Complementary Metal-Oxide Semiconductor
CPLD	Complex Programmable Logic Device
dB	Decibel
DRAM	Dynamic Random Access Memory
FIFO	First-in First-Out
FPGA	Field Programmable Gate Array
FSM	Finite State Machine
DGD	Differential Group Delay
Hz	Hertz
I/O	Input/Output
IOE	Input Output Element
IEC	International Engineering Consortium
LAB	Logic-Array Block
LE	Logical Element
LUT	Look-Up Table
LVDS	Low-voltage differential signaling
LVTTL	Low-voltage transistor transistor logic
MEM	Micro-Electro-Mechanical Systems
ms	Millisecond
mW	Milliwatt
OEO	Optical-Electrical-Optical
OQS	Output Queued Switch
PDL	Polarization Dependent Loss
PMD	Polarization Mode Dispersion
ps	Picosecond
RAS	Row Access Strobe
RIN	Relative Intensity Noise

List of Acronyms

SRAM	Static Random Access Memory
UI	Unit interval
WSS	Wavelength Selectable Switch
V	Volt
VOSQ	Virtual Output Sector Queue

Chapter 1 Introduction

The growth in packet-switched networks has required the use of different technologies to fuel higher speed for data information transport. Photonic technologies are a key technology to enable both longer reach and higher line rates for data transmission. Within these packet-switched networks, nodes exist that contain routers for guiding packets to their destination. Research has sought in integrating optics within routers that have over the many years been implemented using only electronics [Reference 24]. The use of photonics in place of electronics for its central stage allows a significant reduction in the total power consumption for the packet-switch. The creation of the light path using photonic technology consumes practically no power relative to its electronic counterpart which can consume a lot more power. In cases where the fabric is large (ex. 256 X 256), kilowatts of power is easily consumed. Commercial routers use different optical technologies to meet the line rate speeds for the demands of the network. Client interfaces are referred as lower rate (<2.5 Gbps) speeds and line interfaces are referred as higher rate (>10 Gbps) speeds. What if we can embed optics into the routers and make them completely transparent to the optical network? Unfortunately we are not completely there yet; however, since packet-switches are integrated into every router and their scalability is dependent on the architecture used, then selecting an appropriate architecture can help combine optical technology with the functionality required to forward packets from an input port to an output port within the switch at higher bit rates. Electronics can not be completely avoided due to the header decoding required to determine which output port each packet is to be forwarded too and also the use of memory which is required to store such packets when they contend for the same output ports. A proposed means of achieving such a packet switch is described using a three-stage Clos-like architecture consisting of two buffering stages split into sectors interconnected with a reconfigurable optical central cross-bar stage and using a flexible bandwidth provision (FBP) scheduler. The FBP scheduling method redistributes the internal bandwidth according to the statistics of arriving traffic and reconfigures the optical central switches to set up the paths required for the bandwidth distribution calculation. The configuration time of the photonic core is an important factor in the operation of the switch because the central stage stops transporting packets during this time.

This particular architecture allows realizable packet switches where memory contention is relieved

at the buffering stages, therefore supporting larger port count and higher line rates. It takes advantage of the strengths of the electronics as a memory technology and of photonics as transport technology while accommodating to the incorporation of the slower reconfiguration of optical switching technology.

The buffering stages are themselves sub-switches containing shared memory. Each sector within the buffering stage performs the function of processesing incoming packets in order to buffer them in different virtual output sector queues (VOSQ) within a common dynamic random-access memory (DRAM). A VOSQ is a logical memory block that stores packets destined for a particular output sector. The sector also extracts the required number of packets from each queue and directs them to the corresponding output port connecting to the central crossbar stage (the FBP scheduler determines when the packets need to depart from their queues). Buffering is required since only one packet can depart at an output port at a time; therefore, the switch must store packets while they wait until the output port becomes free.

Most common packet-switches consist of fixed-memory size first-in first-out (FIFO) queues; however, this reduces memory efficiency due to the waste of available space that could be used by other queues that would otherwise grow larger than the specified fixed size (because of higher arrival rates and/or service rates). At the expense of some complexity using pointers, this can be resolved by dynamically allocating shared-memory for all existing queues.

The three-stage packet switch architecture can be expanded and distributed as a larger scale network in a star topology whereby the sectors themselves can be considered as edge nodes in the network and the central stage can be a non-blocking reconfigurable optical switch. Since fiber optics is used to connect the edge nodes, it is not necessary that the core central switch be very close to these edge nodes; rather they can very well be far away, a typical trend experienced in recent system architectures using linecards. This poses problems on how far away the edges nodes can be without experiencing transmission penalties between them.

The following identifies the contributions by the author to the functional blocks needed to realize a hardware demonstrator of this type of packet switch:

1.Overall design of the packet switch hardware demonstrator in collaboration with other members of the research group, the Photonic Technology Laboratory (PTLab)[1], specifically the introduction of a wavelength selective switch as the optical crossbar in the central stage of the packet switch [Reference 26], [Reference 27], & [Reference 28].

2.A System-on-Chip hardware implementation of a linked-list memory manager using both shared off-chip memory and field-programmable gate arrays (FPGAs) to manage the enqueuing and dequeuing of packets for a particular VOSQ. This enables the foundation for building a shared-memory switch and provides for dynamic memory allocation to each VOSQ.

3.The characterization of optical impairments of a micro-electrical mechanical systems (MEMs) based optical wavelength selectable switch to be used in a cross-connect configuration as the central stage. This will help identify the minimum time between successive reconfigurations of the optical central stage required for the FBP scheduler and identify factors impacting signal quality when sectors are positioned far away from other sectors that are connected to the central photonic switch.

1.1 Organization of Document

The thesis is organized into the following chapters:

- Chapter 1: Provides a brief description of packet switching and its role within the network, followed by a description of the architecture of the three-stage sectored packet switch with an embedded optical central stage and the hardware building blocks required. The chapter also provides a description of the photonic technologies currently available that can be used to build the central stage for various port dimensions. Background on the field-programmable gate array System-on-Chip architecture from Altera Corporation is described along with the bus infrastructure internal to the FPGA that is used to connect the many peripherals including off-chip devices. Finally the shared-memory management or linked-list algorithm for creating the digital finite state machines is detailed.

[1] This name was recently changed from Photonic Network Technology Laboratory (PNTLab).

- Chapter 2: Provides a detailed description of the System-on-Chip algorithmic state machine of the linked-list memory manager for the packet switch. The behavioral timing simulation of the test bench is shown using off-chip memory models for both memory technologies.
- Chapter 3: Provides a description of the optical impairments incurred within optical interconnects and details the measurements obtained from a photonic central stage using wavelength-selectable switches. One of the main parameters of interest is the reconfiguration time for the central stage used for the Flexible Bandwidth Provisioning algorithm.
- Chapter 4: Provides the conclusion of the thesis and also lists areas for future work.

There are also the following appendices:

- a list of glossary of terms,
- a list of test and measurement equipment used for measuring the optical impairments,
- a derivation of the inband crosstalk used for the measurement,
- a set of block diagrams detailing the interface signals for the peripherals designed for the linked-list memory manager,
- simulation timing diagrams for the linked-list memory manager during intialization
- the VHDL source code for each of the designed peripherals,
- the VHDL source code used for creating the test bench,
- a list of references.

Chapter 2 Background

In the previous chapter, a description of the various topics to be detailed in the thesis was highlighted; specifically, the type of packet switch that is being designed was summarized and how this architecture can be sought to be a large network with large links between the core central switching fabric. In this chapter, we will review packet switches, and its role within the network, while also providing a brief description of the common types of packet switching network topologies. The main architecture of the packet switch and its implementation within hardware will be the primary focus for the remaining chapters of this thesis. We shall be referring the hardware details as *"the hardware demonstrator"*. Since the primary focus for this thesis is tailored towards both the circuit architecture for the memory management within the packet switch, and the details surrounding the optical central stage, the concepts of the memory management technique used and a description of the potential optical technologies to be used within the central crossbar will be highlighted. The scheduling algorithm for the packet switch will be briefly reviewed since it plays a role on how the pathways are constructed within the switch and how the packets are cleared from the memory buffers.

2.1 Packet Switching

Since the early 1970's packet switching has been created to influence the computing age in transporting information from one computer to another computer. The packet switching network was originally set in place to mirror the existing circuit switched network used to accommodate voice traffic. Since original circuit-switched networks could not accommodate digital information in the form of packets - a series of pre-defined length of bits describing data - the packet switch network would take its place to provide a connection path between a source and destination and allow that digital information to be transmitted across the network. These packets therefore contain control information such as the destination address to indicate the final destination of the packet in the network; however, it is the responsibility of the packet switch or higher level routers to physically create the logical pathways the packet must traverse and direct it through them. Depending on the type of data being transported across the network; for example, video, or

electronic images, the payload of the packet will be larger than the specified minimum size and will be segmented into a series of packets each containing sequence labels to properly identify the order of the data for reassembly at the destination. The packet switch can be placed in different locations within a network which can form either a mesh configuration, or a star configuration. Figure 1 and Figure 2 shows a topological diagram of how the packet switch is distributed and connected to create both a mesh network and a star network, respectively. The Local Exchange Carriers (LEC) connect to a switch to which they will transfer digital information in the form of packets across to the destination LEC through the pathways generated by each switch. The packet switch here is depicted at a higher level within the communication network, but can also be expanded further to other areas in the network which use switching such as backplanes interconnecting linecards.

The International Standard Organization (ISO) created a seven-layer architecture called the Open System Interconnection (OSI) reference model. This standard OSI model creates a protocol stack to which simplifies the process of system providers to build equipment that can communicate with one another. The OSI reference model has 7 layers to which the 3 bottom layers are focused on transmission protocols and switching across the network; whereas, the others pertain to the data protocols; that is, what we do with the data once it gets to its destination. There are seven layers that are labeled as: physical, data link, network, transport, session, presentation and finally application. The primary area within the model that the packet switch plays a role and obtains its core directives is within the Network Layer and the Data Link Layer; typically referred to as Layer 2 and Layer 3. Detailed description of these layers and its exact functions can be found in [Reference 10]. One item that should be emphasized however is that the Network layer does have other responsibilities such as quality of service than the routing and switching of the data, but it is the Data Link layer that will map these packets onto the physical circuit pathways.

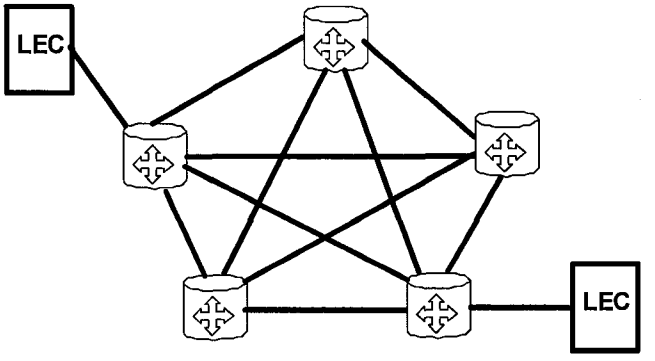


Figure 1. Packet Switches used in a MESH network topology

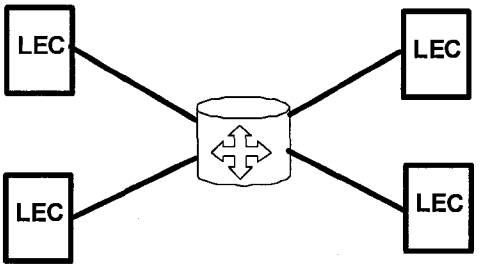


Figure 2. Packet Switches used in a STAR network topology

The physical connections to the packet switch using today's technologies can be realized in many ways using optical transceivers, radio frequency transceivers, or direct low level digital transmission links. Although the details of these physical connections are beyond the scope of this thesis, it represents an important part of the hardware architecture interface of the hardware demonstrator and will be discussed in the chapter on optical interconnects used for the central stage.

2.2 Packet Switch Architecture

Packet switches are a critical element within a packet-switched network and a view of how this is designed is important in order to ensure the most idealistic interaction between the source and the destination. It is not surprising to learn that any impairments within the packet switch will directly translate into poor performance within the network. A packet switch performing poorly will

introduce packet delays which may exceed the specifications for the transmission system excluding the use of transmission control protocols. The main design goal for a packet switch is to allow a packet to be received and directed to an output port gated for the direction towards its destination with no delay. Unfortunately this is not the case, due to the dependencies on line rate, packet size, and the load to the switch. For optical transmission systems, line rate plays an important role as the serial rates here can span beyond 10 Gb/s, unlike its RF (wireless) counterpart. Some of the more common issues in generic packet switches can be itemized as packet contention, packet sequencing, packet forwarding, and finally memory buffer access times. They can be summarized as follows:

- *Packet contention* occurs when there are two or more inputs ports that are requesting to transmit to the same output port of the switch.
- *Packet sequencing* occurs when the memory word size of the packet switch is not the same size of the packet and these packets are fragmented and stored according to the word size. When these packet fragments are transmitted to the output sector, they could potentially arrive at different times and may be buffered in a different order to when they were originally fragmented.
- *Packet forwarding* is identified as a scheduling delay that results from the computation in determining which output port each packet is to be transmitted too and also the appropriate bandwidth required to transmit these packets across from an input port to an output port. Normally this requires the header processing of the data for each input port, knowledge of the type of traffic statistics that exists and memory utilization.
- *Memory buffer access time* results from physically storing or accessing the contents to and from a memory location, typically random-access memory, either on-chip or off-chip. The access time becomes more of an issue when the line-rate increases since memory access times are generally less than the line-rate in an optical transmission system.

The many different architectures studied within the literature try to optimize these parameters. All of the possible types will not be detailed in this thesis; however, a review of some of the different types of switches of main importance are captured in [Reference 11]. The idealistic switch is the Output Queued Switch (OQS) which contains one buffer for each output port and has the ability to direct all packets to their destined output ports. Although physically unrealistic, all other architectures have strongly sought to achieve the level of it's performance. The proposed conditions of a strictly non-blocking configuration of a switching cross-fabric was derived by Charles Clos in 1953 [Reference 12]. This paper identifies that a connection from an idle input port

to an idle output port can be achieved with less than N^2 crosspoints with the use of 3 switching stages, which is important for the scalability of the packet switch being considered in this design. Figure 3 depicts a general setup of a strictly non-blocking 3-stage CLOS packet switch. The architecture consists of an input stage with memory buffering, and an output stage with memory buffering and a central stage with no memory buffering. The advantages of this architecture are that it exploits high connectivity by increasing the amount of interstage switching connections to an output sector to adapt to increases in traffic destined for the same output port and that it facilitates scalability to larger port counts. However, the architecture does have a disadvantage, since it requires specialized scheduling algorithms in order to configure the individual switches to properly configure the number of flow paths. A flow path represents a physical connection path between an input port to an output port through the central crossbar. There could be many depending on how many paths are created within the central crossbar. Previous architectures have generally sought the use of electronic high-speed switches in the central stage. Previous packet switches generally use very fast scheduling algorithms that allow the central stage fabric to configure every timeslot (in the order of a memory access time) which results in the dissipation of a large amount of electrical power. To solve the issues of high power consumption within the central stage, ease the scheduling bottlenecks across the input and output nodes at increased line rates, and achieve a performance near that of an Output Queued Switch, an architecture denoted as the Sectored Packet Switch with An Optical Core; otherwise known as the Flexible Bandwidth Provision Packet Switch depicted in Figure 4 has been created. The details of the derivation of the architecture can be found in [Reference 11]. Although a brief overview will be provided of the main elements of the architecture for the reader in order to highlight the elements of primary focus for *the hardware demonstrator*, which is tasked by Dr. Trevor Hall's team to develop in order to prove out concepts that have only been modelled via computer simulations.

The hardware demonstrator is a scaled down version of a 64 X 64 packet switch described in [Reference 11]. It contains 8 input ports and 8 output ports and is sectored with 4 input or output ports per sector. Each sector of this sectored packet switch are common memory switches (also known as centralized shared memory switches), and are interconnected by a switching fabric that emulates a stack-up of L x L switches. These interstage switching elements are fast configurable photonic switching elements. However, for the purpose of *the hardware demonstrator,* a slower

photonic technology was used. The crossbar switches will be an optical interconnect emulating a stack-up of four 2 X 2 switches. Each sector will contain 4 input ports and 4 output ports. As a result each connection from the output ports of each sector are connected directly to the 8 ports of the central stage switch. Similarly the 8 ports at the input of the output sectors are also connected to the central stage optical switch.

The use of CLOS architecture allows scalability using low-port count switches and growing the interstage to accommodate large connections to each common memory switch. The common memory switches; otherwise known as the input sectors, are optical-to-electrical-to-optical (O-E-O) conversion switches that take an optical serial data stream from a desired input port, converts it to the electronics domain through an optical receiver, processes its header and buffers it in shared-memory according to their desired destination sector. The packet is then retrieved from memory, converted back to the optical domain through an optical transmitter and transported across the photonic central stage to an output sector. The process of conversion is repeated here again and the packet is finally transported optically to the destination link. The common memory switches contain the most important functionality since the majority of the packet processing and memory management occurs in this section of the packet switch. The memory management responsible for maintaining the packets into buffers as described previously is one of the main topics of this thesis. Specifically, the memory within the input sector is divided into logical first-in-first-out queues, one for each output sector that exists for the packet switch; otherwise known as Virtual Output Sector Queues (VOSQ). Figure 4 shows each input sector for the packet switch connected to the central stage crossbar. Each input sector and output sector uses shared-memory of which is segregated into queues. On the input side, there is one queue for each existing output sector (2 of them in this case), and on the output side, there is one queue for each output line (4 of them in this case). The block diagrams for the sectors are sketched this way to represent the read / write accesses to single-port memory; that is, memory to which a read or a write cannot occur simultaneously.

One notion that may need to be clarified is the definition of switching. A packet is said to be switched when the packet within the input sector is received, processed for its destination sector, and then stored in the appropriate VOSQ memory location. The central stage in this architecture

allows for the packets to be transported and copied in the output sectors memory quickly even when the line rates are increased. The use of optics in the central stage provides variable bandwidth connections required for different types of traffic loads to each output sector. The configuration of this central stage occurs on a scheduled basis.

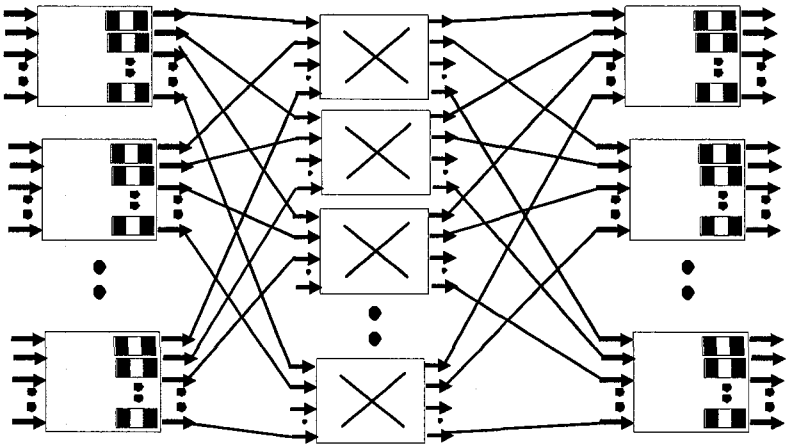


Figure 3. 3-stage CLOS architecture switch

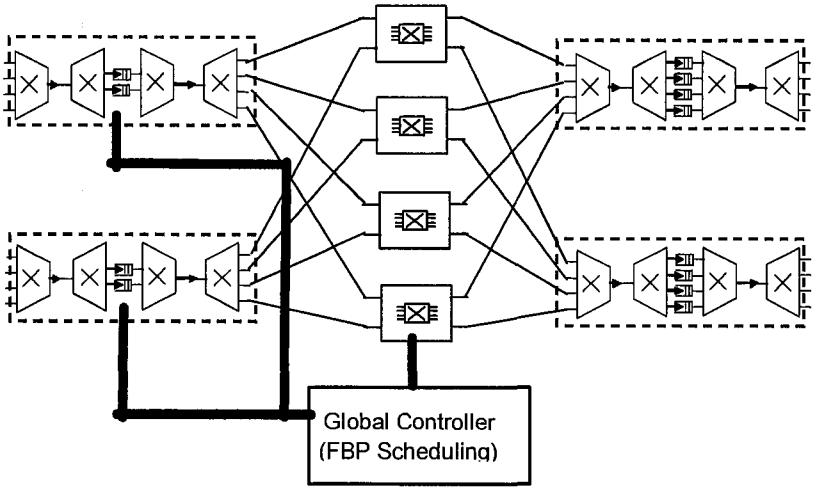


Figure 4. High-Level block diagram of an 8x8 sectored packet switch
The CLOS-like architecture is seen with an interstage crossbar using a stack-up of 2x2 switches

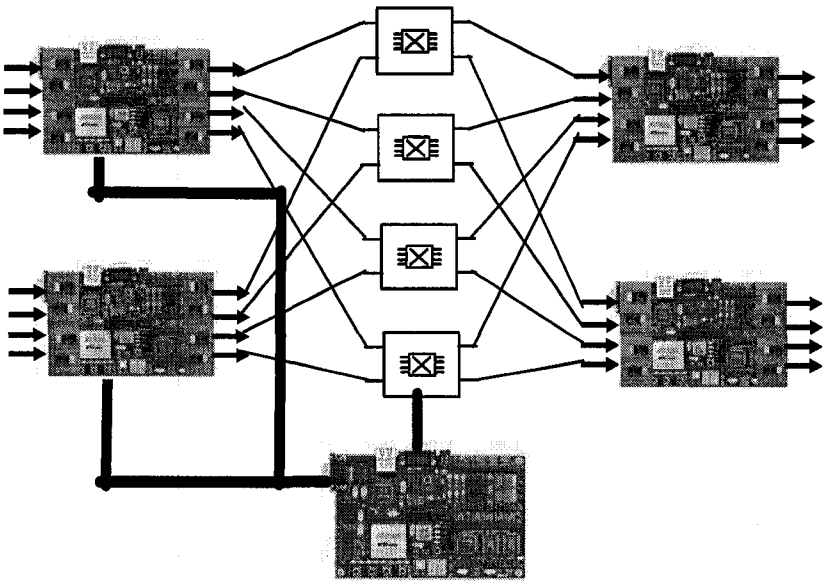


Figure 5. 8x8 Sectored Packet Switch Using FPGAs and 2x2 switches

Figure 5 depicts the hardware interconnection using Altera Stratix I FPGA development boards. The development boards allow for the integration of the optical-to-electrical conversion blocks, header recognition, memory interface, and arbitration logic required to process packets and prepare them for transport to their destination links.

All packet switch architectures rely on very sophisticated scheduling algorithms in order to keep the total memory queue lengths low and the delay for each flow as low as possible. This packet switch is no different. In order to provide the dynamic bandwidth allocation between the sector pairs, the central stage is also managed by a centralized control unit, named the *Global Controller*. The Global Controller is the main intelligence for the architecture and allows the scheduling algorithm to be processed and configures the core central switch fabric on a periodic basis. The algorithm used in this architecture is called Flexible Bandwidth Provision (FBP). Reference 11 captures in detail the FBP algorithm; however for simplicity for the reader a brief description is provided since it does impact the length of the VOSQs in the input sector by providing adequate bandwidth to service those queues that are being filled at a higher rate.

The FBP algorithm relies on the calculation of a traffic matrix whose elements correspond to the measured/estimated traffic arrivals or requested bandwidth between two sectors. From the traffic

matrix, an inter-sector service rate matrix is constructed whose integer entries specify the number of paths to be set up between sector pairs. By varying the number within the service matrix, FBP is achieved and the number of packets to be taken from each queue is defined. The configuration of the central stage is found by solving an edge-coloring problem on a bipartite graph. Each permutation defines the state of a crossbar switch. The FBP algorithm is executed on a periodic basis, referred to as the inter-configuration period [Reference 11], and the resulting schedule is a function of the traffic demands, which for the purpose of the *"hardware demonstrator"* are defined as the length of the Virtual Output Sector Queues.

2.3 Optical Technologies Used for Central Stage Crossbar

The primary architecture as previously discussed details the mid-stage central crossbar as an optical entity requiring in some cases large external port count switches in order to connect with each individual sector. The CLOS type architecture helps reduce the requirement for large density switches in the central stage; provided a reasonable amount of speed-up. Generally a high speed-up factor would increase the port count required to interconnect with the sectors. Speed-up is the ratio between the total bandwidth at the output of a sector connecting to the central stage crossbar relative to the total bandwidth at the input of the same sector; or can also be sought as the ratio of the number of ports at the output of a sector divided by the number of ports at the input of a sector assuming in this case that the line rate of all ports are equivalent. The interconnect size requires optical technologies that can scale appropriately. Typically a fast reconfiguration time for the optical crossbar is required so that it becomes negligible relative to the inter-configuration period. If the reconfiguration time is larger than the inter-configuration period, packet delay penalties will be incurred. The issue with optical switches to be used in the central stage is the trade-offs related to the size of the switch and the different technologies that can be used. Figure 6 depicts a map of the port density for optical switches, and the appropriate technology available for realizing those densities. What is shown is that large port counts can be available only at low switch speeds and high switching speeds can be made available only at low port counts. In this figure the technologies used for low port count switches are directional couplers made with lithium niobate which intrinsically have a fast response; however require a large amount of voltage and power to provide

the phase change needed for switching, and therefore smaller structures are more feasible than larger ones. A company called Civcom has patented what they refer to as the Solid Free Space (SFS) technology which is a combination of liquid crystal and free-space optical switches capable of exhibiting very fast switching speeds (~400 ns); however, are available only in low port counts [Reference 6]. Other very fast response switches which can scale to larger port counts is using phased-array technology or a stack-up of semiconductor optical amplifier (SOA) type structures. As the switching rate becomes slower by orders of magnitude the options become a bit more limited and the use of liquid crystal and liquid crystal phased-arrays can be selected to achieve the port count density. When slow response times are acceptable, both low port counts and large port counts are feasible through the use of micro-electro mechanical systems (MEMS) or thermo-optic type switches. Both have response times in the milliseconds due to their mechanical and thermal properties. Specific types of architectures (such as broadcast & select) using these types of technologies and their crosstalk trade-offs are addressed in [Reference 13]. In a subsequent chapter, the optical properties; specifically the reconfiguration time of the MEMS based technology that will be used for *the hardware demonstrator* is measured.

Since this packet switch architecture uses an optical interconnect as the central stage, it becomes important to explore the different technologies regarding their reconfiguration times relative to the inter-configuration period. The work presented in Reference 11 does not explore the effects of reconfiguration times as it is assumed to be instantaneous. Unfortunately the other technologies were not available during the course of the project period to allow for a comparative study against the inter-configuration period; however, it does remain as one of the critical parameters affecting the overall transmission delay.

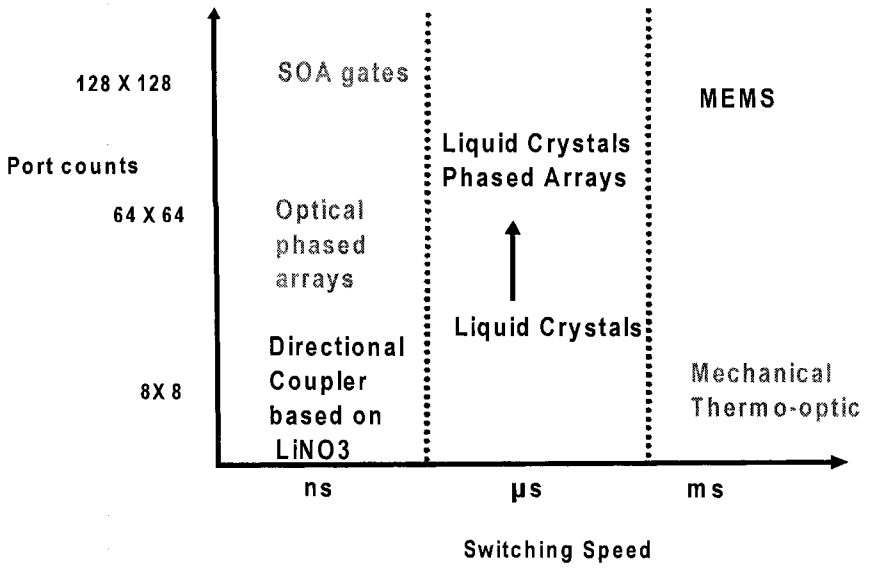


Figure 6. Port Count versus Switching Speed Optical Technology Map

2.4 FPGA Technology Within The Packet Switch

The packet switch as described allows the transfer of packets to and from input ports and output ports. It doesn't matter whether it is an electrical or optical serial input data stream, both require a common processing platform executed within the electrical domain to decipher information from the bitstream and direct the data to its destination. Figure 7 depicts a generic block diagram to how the data for an input port is received.

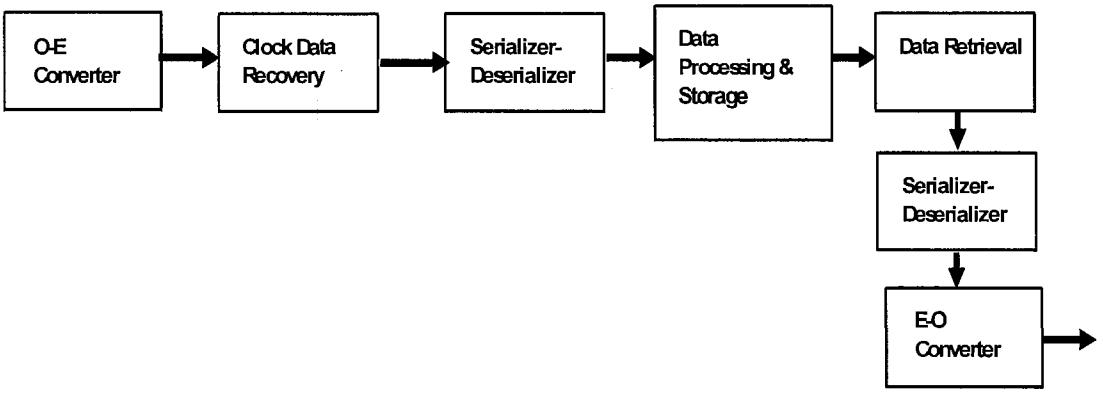


Figure 7. Common Packet Receiver Functional Block

The block can be broken to several functional units:

a) Optical-to-Electrical (O-E) Conversion: This section strictly converts an optical signal to an electrical signal and receives the data onto an electrical high-speed circuit.

b) Clock and Data Recovery: This section strictly is responsible for using the incoming serial stream of data and phase shifting a local oscillator to represent the exact frequency of the incoming data signal. This frequency is then the recovered clock used to time out the data bit stream at the rate to which it was transmitted. This functionality is achieved through the use of a digital phased-locked loop. Most commonly used digital phased-locked loops are phase-frequency detectors. [Reference 19] depicts an example of a linear phase-frequency detector.

c) Serial-Deserializer: This section is used to transform the data bits from a serial stream to a parallel stream, most commonly used to interface with a 32 bit or 64 bit wide bus. The opposite is also true in that the parallel stream can also be converted back to a serial stream.

d) Data Processor & Data Retrieval: These sections are used to perform bit header recognition and they incorporate the decision logic to store full size packets into memory segmented in terms of the destination sectors, or output ports. The retrieval section removes the entries from memory for hand-off to the transmitting section.

e) Electrical-to-Optical (E-O) Conversion: This section strictly converts an electrical signal to an optical signal and transmits these data onto the optical pathway.

Current silicon-based complementary metal-oxide semiconductor (CMOS) technology provides many of the integrated circuits to construct the path as shown by the figure; however, a central processing unit (CPU) is generally used to perform the processing of packets and provide the memory interface for storage and retrieval. The issue with integrated circuits is that they can

consume a large amount of real estate on a printed circuit board and power, and require good termination matching between them to optimize performance. To alleviate this, current state of the art field-programmable gate arrays (FPGAs) are equipped with the required building blocks to create these blocks on one silicon chip. This chapter will review the FPGA architecture to be used for the particular digital building blocks of interest for the design; primarily the data storage and retrieval logic.

2.4.1 System on Chip Architecture

Since the 1980s the digital world introduced two family of devices, the first being the programmable logic device (PLD) and second the FPGA. The differences between them were architectural. The first device contained a number of gates arranged to form sum-of-product functionality; compared to the second which contained much more embedded logical look-up tables (LUTs), internal fast memories, processor cores (micro- & digital signal processors), and a data bus for improved interconnects between the functionality and the logical blocks. The architecture for the FPGA which contains all the embedded functionality including memory is usually referred to as System on Programmable Chip (SOPC), or System on Chip (SoC). These may be used interchangeably throughout the document. For the design of the common memory switch, the SoC approach is the primary architecture used in the choice of the FPGA for the design. For the realization of the designs within the thesis, the Altera family of FPGAs based on their SoC architecture was used.

The FPGA architecture consists of logic-array blocks (LABs) that contain logical elements (LEs). The elements are what is used to generate the digital logical functions. The LABs are surrounded by an interconnect matrix that transports and provides connectivity to other area's within the FPGA such as multiplier blocks implemented within the digital-signal processor, or internal memory blocks. These interconnects are mapped to the input-output elements (IOE) and external pads of the chip. Since FPGA output signals can be connected to any type of peripheral with different protocols, the IOE blocks are designed to contain a variety of I/O standards such as low-voltage transistor logic (LVTTL), low-voltage differential signaling (LVDS), and other high-speed standards which are configurable once the digital design is complete. The IOE blocks also contain

tristate buffers to support bidirectionality, such as the case with asynchronous peripherals. The details of each LE, LAB and IOE are discussed in [Reference 15]. Figure 8 depicts the LE functional block within the Altera Stratix FPGA family. The diagram is taken courtesy of Altera within [Reference 15]. The main section of interest in this diagram is the LUT which is designed with the ability to create logic functions for 4 input data variables, while having the output synchronized to a clock via a connection to a flip-flop (either D, T, JK, or SR). It has simple bit-wise arithmetic properties through both the carry-chain and synchronous data bit loading blocks. Creating a system requires the interaction amongst these LABs as they provide the generation of custom logic as well as the other elements such as the processor core, memory and signal processor core through a large interconnect switching fabric. Since clocks and internal circuits must pay close attention to fanout, these switching interconnects allow distribution to any location within the chip.

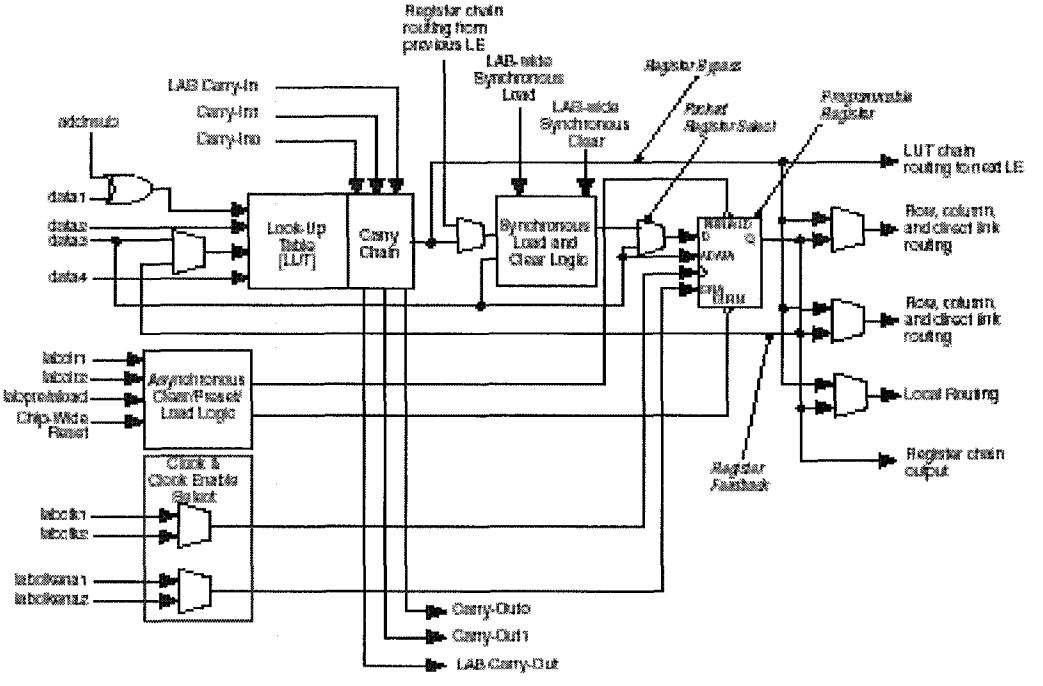


Figure 8. Logical Element Block within Altera Stratix I FPGA

2.4.2 Avalon Bus - The Switching Interconnect Fabric

One of the key design entities within the FPGA is the proprietary interconnect technology. The switching interconnect fabric used within the Altera Stratix family FPGA is called the Avalon Bus. It is a proprietary designed interface which allows any user defined master peripherals (derived by LABs) to be interconnected with other circuits within the chip and or other slave peripherals (ex. Ethernet ports, serial ports etc....). Some of the key features of the Avalon Bus of interest include the ability to:

1. Decode address locations - the switching fabric can be connected to any port as long as an address is provided.
2. Multiplex Data Paths - the switching fabric has the ability to imbed multiplexer circuits when data is being transferred to and from one slave to a particular master, or vice versa.
3. Provide Clock Domain Independence - the switching fabric is agnostic to whether different peripherals operating at different clock frequencies are connected. The fabric will create circuitry required to allow data to be traversed and registered at the correct frequency.
4. Provide Wait-State Logic - the switching fabric allows for control signals to be activated to provide suspension of an event while a slave peripheral requires additional clock cycles to complete a particular task.
5. Provide Pipelining Connections - the switch fabric also can provide additional registers in order to increase the clock frequency between a given master or a slave pair; specifically in the case when many master ports share the same slave peripheral. It should be highlighted that the pipelining introduces additional latency due to the additional registers and logic functions required.
6. Provide Dynamic Bus Sizing and Native Address Alignment - the switch fabric has the ability to pair two peripherals that do not share the same data widths and align addresses accordingly.
7. Arbitrate for Multiple Masters - the switch fabric creates an arbiter per slave peripheral for the occurrence that more than one master shares a common slave peripheral and requires access at the same time. Slave-side arbitration provides benefits since the entire bus continues to allow access to other master peripherals that need to access other peripherals.
8. Provide Burst support - the switch fabric contains logic to allow for burst transmission, a series of transfers between a master and a slave peripheral of a certain length of bits. This allows continued interaction with a specific slave until the series of bits are completely transferred without a master having to make an additional request to the slave.

9. Multiplex Multiple Interrupts - the switch fabric similarly to the Data Multiplexing has the logic to multiplex many interrupt (IRQs) from many slave peripherals to an identical master. Albeit, within the design of the digital memory management scheme proposed in this document, this is not required as this pertains mainly with the use of embedded central processing units within the FPGA; however, it is worth highlighting.

The key design intent of the Avalon Switch fabric allows generic control signals, such as *data*, *address*, *write* and *read* to be specified at the interface so that any peripheral connecting to the fabric can communicate with each other and both become aware when they are transmitting or when they are receiving information. The Avalon Switch fabric also provides other control signals such as *waitrequest*, so that slave peripherals requiring additional time to complete a logical event, denies other master peripherals from getting access to that slave until the event is completed. In all cases, the slave peripheral is responsible for asserting the *waitrequest* signal, while a master peripheral probes the Avalon Switch fabric to determine when it becomes active. More information on each of these functions and the details on the interface signals required for connecting to the Avalon Switch fabric is highlighted in [Reference 16] and [Reference 17]. These signals are what will be used in the generation of the memory management SoC logic within subsequent chapters of this thesis.

2.4.3 Memory technologies used in the linked-list design

Within each sector of the packet switch exists a memory location to which each incoming packet is stored. The input sector of the switch is segregated into first-in first-out (FIFO) virtual output sector queues - one queue for every output sector, and the output sectors are segregated into first-in first out (FIFO) output queues - one queue for each output port [Reference 11]. Before describing the implementation of the shared memory for managing queues, it is beneficial to highlight the types of memories which will be used for the design.

The FPGA typically contains internal memory that has fast access times compared to external memory since there are additional propagation delays incurred by going off-chip. Generally, the I/O pads and the path length to the off-chip device contribute to a large part of the delay. There are several variants of off-chip memory technologies but the common types of memory technologies available are both the static random access memory (SRAM) and synchronous dynamic random

access memory (SDRAM) both of which memories retain information only while power is activated. These two technologies are offered on the Altera Stratix I Development Board provided to the university project and will be used as the basis for packet storage in the packet switch demonstrator.

For SRAM, cross-coupled transistors are used to store either a logical "1", or a "0" - a bit of information. Typically address lines and data lines are independent within the SRAM architecture and not multiplexed. Drawbacks of the SRAM are that it generally consumes much space when large amounts of data are to be stored and that it consumes a lot of power when clock speeds increase and when many of them are used to store large amounts of data. Comparatively, SDRAMs are created using a one-transistor cell with a capacitor as seen in Figure 9. The transistor consists of a word line, bit line and a capacitor. The storage takes place by providing a voltage level (a logical "1") to the word line and then providing either a logical "1" or logical "0" to the bit line to either charge or discharge the capacitor to the desired level and have a bit of information stored or retrieved. Sensing amplifiers are used to detect small voltage differences on the bit line in order to determine the value stored by the internal capacitor. Due to the effects of memory loss, otherwise known as capacitive leakage, the memory cell is required to be charged periodically (refreshed) to retain the contents stored within the capacitor.

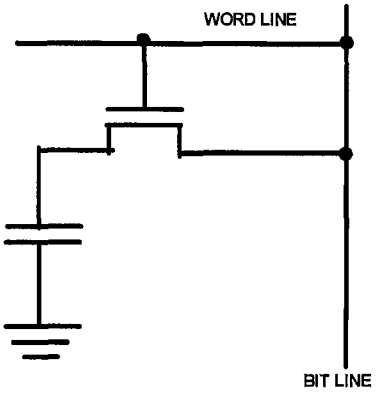


Figure 9. SDRAM Transistor Cell

Unlike the SRAM, SDRAMs have larger densities. Transistors are configured into multiple two dimensional grids, each one replicated into multiple banks so that the memory can contain a large

storage capacity without an increase in the number of address lines required to access the data bits. The address lines permit access to any given row and any given column. Each cell extracted from the grid represents a 32 bit cell.

Due to the limitations of the existing internal memory within the FPGA for use as cache memory as it will be shown later within the input sector, cache memory will not be used as part of the memory management design.

2.4.4 Linked-List structure and operation

There are two options to share a memory. The first requires the memory to be divided into equally spaced blocks, one for each queue, and the second is to create a variable size linked-list for each queue. The second approach will be chosen for the implementation in hardware as it provides dynamic memory boundaries for each queue while taking advantage of the entire memory and minimizing the unused space. The former approach using equally divided memory space would tend to under utilize the memory in the case where some queues grow disproportionally longer than others; however, this approach would provide an advantage in improving store/retrieval times since there is no overhead associated with controlling pointers relative to the linked-list case.

As previously highlighted, to implement the linked-list VOSQ, both off-chip memories will be used. There are a couple of design options to implement the linked-list using the off-chip memory:

1. Keeping a list of available *idle* address locations for each packet and the *head, tail* and *next* pointers for each linked-list are stored within SRAM; while the data is stored within SDRAM.
2. Keeping a list of available *idle* address locations for each packet stored within SRAM; while the *head, tail* and *next* pointers for each linked-list are stored within SDRAM along with the data.

SRAMs read and write times are generally much faster than that of SDRAMs since the are implemented using a different internal architecture and the SRAM interface is asynchronous. For the linked-list operation, the main design objective is to provide the SDRAM address on the bus as quickly as possible so that the limitation on access times are imposed by the SDRAM. Unfortunately in using linked-lists there is no known method to avoid the use of pointers, and

therefore option #1 is chosen since it would provide minimal requests to the SDRAM except for the time the data is required to be stored or retrieved.

The following subsection will describe the shared-memory criteria, and a detailed description of the implementation of the linked-list. The next chapter will be focused on the detailed finite state machine design for the *"the hardware demonstrator"* of this implementation.

General linear linked-lists have been implemented in software using data structures as the ones described in [Reference 18]; however, in such cases, the memory compilers automatically allocate and deallocate contents for the data structure. Details are obscure on how the compilers utilize the memory as this is microprocessor dependent. This form of memory access using microprocessors would lead to a poorer performance for higher speed and higher port count packet switches. It is for this reason that the direct memory access approach by using a dedicated state machine peripheral for interfacing with the ports of the packet switch, a dedicated bus, and memory controllers will be used for the shared-memory design in this thesis. Note that the term *list* and *queue* will be used interchangeably throughout this document. They both refer to the packets being stored within shared-memory analogous in operation to a first-in first-out buffer, but in dispersed memory locations.

The overhead requirements for a linked-list queue are the following:

a) Block of memory containing both the *head* and *tail* pointer addresses
b) Block of memory containing the *next* pointer address.
c) Block of memory containing the *idle* addresses

The *head* and *tail* pointers represent SRAM addresses that identify the first and last elements of the queue, which in turn identify the locations where packets will be written too (when a packet is appended to the tail of the queue) or read from (when a packet is removed from the queue), respectively. The *next* pointer represents a SRAM address that references the location of the next data packet in the queue. The idle pointer represents a SRAM address that references the next location in SRAM that in turn contains the SDRAM address that an incoming packet is allowed to occupy in memory for storage.

Figure 10 depicts a simple linked-list structure for two queues. The figure identifies the head, tail,

and next pointer, the packet data and finally the null pointer which symbolizes the end of the queue. The null pointer in hardware is identified by a series of hexidecimal "FFFF" values.

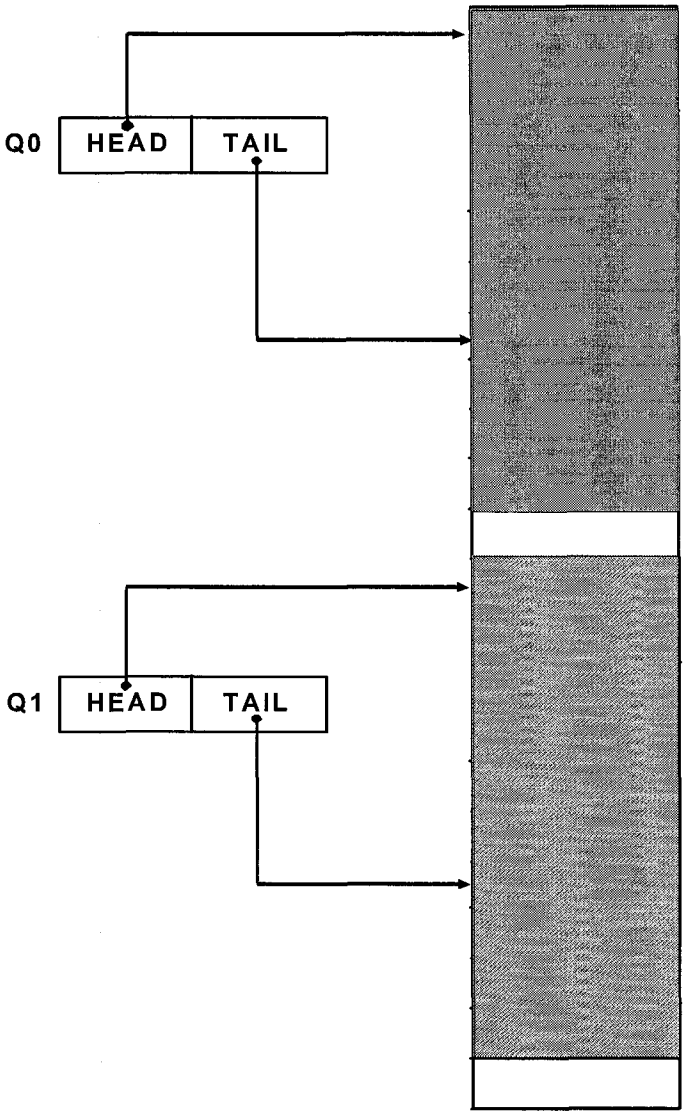


Figure 10. Simple Linked-List structure with two VOSQs.

Figure 11 depicts how the SRAM is divided to store all these overhead pointers. The data bus width, "*W*", is divided into two parts: the first is the address location in SDRAM where the actual data packet within the linked-list resides, and the second is the *next* pointer address. The remaining segments of memory capture the *idle* addresses and the *head* and *tail* addresses. The location in

SRAM of the *head* and *tail* addresses are fixed since these locations do not need to be dynamic; rather, the contents need only to be updated. Increasing the number of VOSQs within a sector requires more space assigned to hold other *head* and *tail* pointers as there is one set for each queue. The allowed shared memory size is taken to be *"W/2"* since for every *'idle'* address location needed, an equivalent number of *'next'* pointer addresses are needed as well in order to keep track of packets that pertain to a particular VOSQ. The SRAM address length, Q, is typically more than W/2; therefore allowing all the latter segments to be captured in SRAM. If this is not the case, then additional SRAM memory would be required to extend the datawidth, "W", to accommodate for the pointer overhead. Note that the total *'idle'* pointer addresses available is less than the entire address size of the SDRAM because of the difference in their densities. More SRAM would also be needed if one wanted to increase the size of the *'idle'* addresses available to store more packets in SDRAM and take advantage of its architected density.

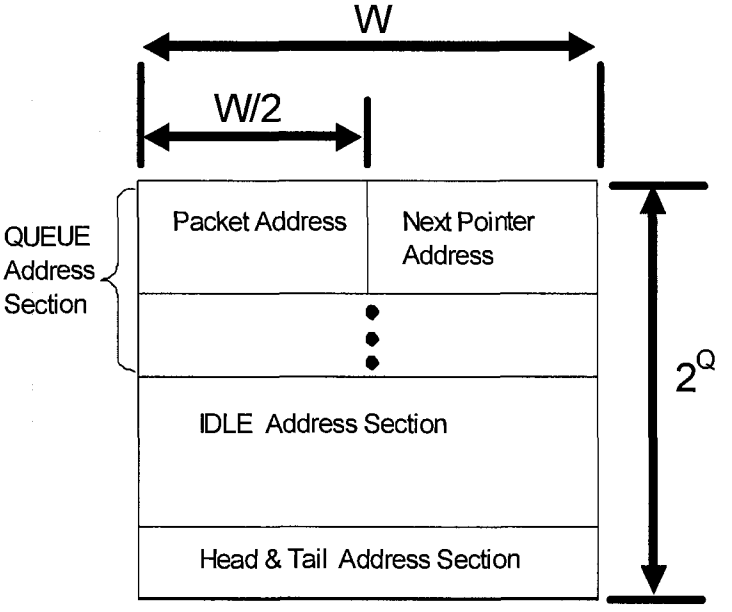


Figure 11. Linked-List SRAM Memory Map.
Packet Address is a SDRAM address location of a stored packet, only W/2 bits wide, Next Pointer Address is a SRAM address that identifies the location of the next packet in the list within the queue section, only W/2 bits wide, Idle address is the SRAM address pointer that references where the unused SDRAM address locations and the head/tail addresses are SRAM addresses whose contents point to the start and end of the linked-list.

The number of VOSQ head and tail addresses that can be created is given by the amount of memory

space that remains; that is $\{2^Q - 2^{(W/2)}\} / 2$. Note that dividing by 2 is needed in order to allocate room for both the head and tail addresses.

Let *"b"* represent the number of VOSQs within a given common memory switch (sector), let *"L"* represent the number of packets in a VOSQ, let "2Q" represent the total number of 'idle' address locations retrieved from SRAM memory used for packet storage, then in order to implement the linked-list with pointer assignments stored in SRAM and data packets stored in SDRAM, the following shared-memory criterion must be maintained.

$$\sum_b L_b \leq 2^Q$$

This criterion emphasizes that as long as there are 'idle' pointers (i.e. pointing to an empty space in SDRAM) available in SRAM, packets for any VOSQ will be able to dynamically occupy a memory location in SDRAM without exceeding the allowed shared-memory size.

Considering only one VOSQ linked-list, Figure 12 and Figure 13 depict the sequential operation of the linked-list for the write and read operations, respectively. For simplicity of the description, the head and tail addresses have been conceptually separated from the SRAM in order to attempt to identify the operations more clearly. We will introduce an *empty head* and an *empty tail* address pointer to represent the beginning and the end of the *idle* addresses within SRAM. The circuit level details on the read and write operations are captured by the algorithmic state machine in the next chapter. For the purpose of the explanation, assume that Q = 18, W = 32: H_0, T_0 are the *head* and *tail* addresses for the VOSQ, and H_e and T_e are the *head* and *tail* addresses for the *idle* addresses. Both are actual SRAM addresses since all of the pointers reference other areas within SRAM. In order for the linked-list algorithm to operate, the SRAM *'idle'* address region and the *'queue'* address region must be initialized with valid address data. Due to the byte-addressing architecture used by the Avalon Bus within the FPGA, the data within the *'idle'* address region are address values offset by 4 bytes; whereas, the *'queue'* address region are address values offset by an integer multiple of 4 bytes, since the packet data width is typically larger and will occupy more memory. The data values within the SRAM in each region are filled with the use of simple counters

preloaded with the correct address.

The write operation can then be summarized as follows:

1. The tail pointer, To, points to the location of the last packet appended to the linked-list queue. Here the last packet is stored at SDRAM address location "0x0A" and therefore "0x0A" is stored in the most-significant byte of the SRAM word in the queue section. The least-significant byte of the SRAM word in the queue section is "FFFF" and indicates the end of the linked-list queue. In this example, only one packet is in the queue.
2. The head idle pointer, He, (an allowable SRAM address location containing the idle SDRAM address for packet storage) is obtained from within SRAM and written to the least-significant byte position of the tail pointer, To. As shown in Figure 11, this becomes the "next" pointer address and is the location of the next packet that will be appended to the list.
3. The tail pointer, To, is updated with the head idle pointer, He. Doing so increases the queue length by one packet. The tail pointer now contains the SRAM location of the last packet in the queue and the packet can be written to the SDRAM address ("0x0B") that was retrieved from the head idle pointer location. The operation of the actual packet being written to the SDRAM is not shown within the diagram. The "next" pointer (i.e. the most significant byte) at this location is again identified as "FFFF" to show the end of the list. In this example 2 packets are now stored in the queue.
4. Finally the head idle pointer, He, is updated with the next available idle SRAM location.

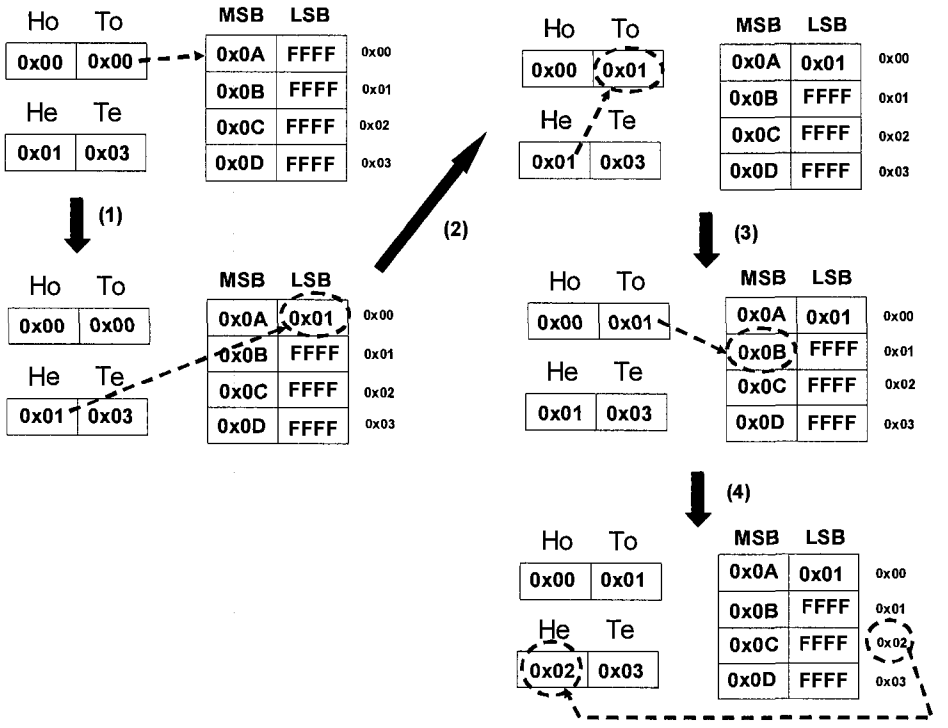


Figure 12. Flow diagram of the linked-list write operation.
He, Te, Ho, To, represent the pointers to idle head and tail address, and VOSQ head and tail address respectively within SRAM.

The read operation can then be summarized as follows:

1. The head pointer, Ho, points to the SRAM location of the first stored packet of the linked-list queue. Here the first packet is at SDRAM address location "0x0A" and this address value is stored in the most-significant byte of the SRAM word in the queue section. The least significant byte field represents the "next" SRAM address of the following packet in the queue. The end of a list as described previously has this least significant byte field terminated in "FFFF". Although not directly shown within the diagram, this most significant byte address is placed onto the SDRAM address bus, and the actual packet is then read from memory. Note that in this example there are 3 packets in the list.
2. The head pointer, Ho, is taken and this value is written directly in the idle tail pointer, Te location in SRAM. Doing this will replenish the list of idle addresses for the write operations so that this address can be re-used at a later time.
3. Replace (write) and update the *head* pointer, H_0, of the queue with the *next* pointer from the current *head* address, H_0 location from SRAM. Doing so reduces the queue length by one packet.

4. In order for the SRAM to be cleared of any stray addresses that could be wrongly addressed, the most significant byte and least significant byte at the extracted SRAM *head* address, H_0 location is overwritten with "FFFF".

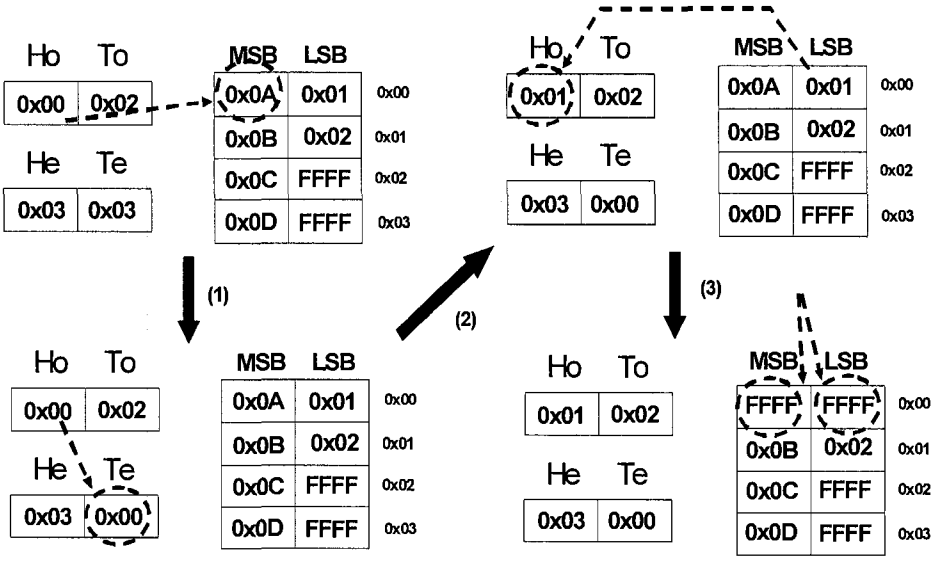


Figure 13. Flow diagram of the linked-list read operation.
He, Te, Ho, To, represent the pointers to idle head and tail address, and VOSQ head and tail address respectively within SRAM

The empty head and tail pointers are located in a different section within SRAM and are updated with the address values extracted from the 'idle' address section of the SRAM. The 'idle' address section is a contiguous block of memory that in normal operation will contain random SRAM memory addresses depending on which queues get filled or emptied. This makes the queues dynamic in terms of having packets scattered anywhere in memory; however at initialization the 'idle' address section contains a sequential list of address values. Two additional pointers are needed to keep track of this *'idle'* address section so that the list of addresses is shortened in the case when packets are written to, and extended when packets are read from memory. In both cases, incrementing the pointers is required. When either of the pointers reaches the maximum, they are allowed to wrap-around to begin at the top of the section.

The methodology described here is important to the realization of the common memory switch - the input sector - of the packet switch. The linked-list is key in managing packets in their respective queues. To realize this linked-list methodology, a customized Finite State Machine (FSM) will be

required to access each memory device, update pointers that reference each packet in one VOSQ and finally keep track of packets in their random location in memory. The details of the digital design onto the FPGA for creating, and updating the linked-list will be discussed in the next chapter.

Chapter 3 Linked-List Hardware Design

In the previous chapter, a description of the Clos-like packet switch with an optical core was presented along with the technologies for implementing the optical crossbar. The previous chapter also described the off-chip memory technologies and the algorithm description (see Section 2.4.4) that was used for implementing the shared memory management scheme (linked-list). In this chapter, we will continue to expand the concepts of the linked-list scheme established in the previous chapter by detailing the Finite State Machines required to create a System-on-Chip (SoC) design. A brief overview of the finite state machine choice used and the design details of the peripherals needed to create the memory manager that will be integrated into the input sector of *"the hardware demonstrator"* will be discussed. Finally, the simulation results showing the operation of the linked-list and the measurement of the memory access times will be presented.

3.1 Finite State Machine Design Methodology

The design of each digital peripheral within this chapter is based on synchronous sequential finite state machine (FSM) networks. The thesis will not describe the principles of FSM's as it is assumed that the reader understands some of the principles of digital design. The designs herein will adopt and implement a synchronous sequential network using a Mealy model [Reference 20]; that is, all output signals are a function of both the present state and the input signals. Each FSM in subsequent sections will be described and explained using Algorithmic State Machines (ASM). Theoretical details and explanations of FSMs and ASMs can be reviewed in [Reference 20]. Figure 14 depicts an example of how the ASM will be created for each peripheral. The figure shows a flow diagram and a table displaying the *output signals* of the state machine and their values for each state. The *state box* represents the current state of the output signals. The *decision box* highlights a direction taken by the state machine and is governed by the value of the input signals.

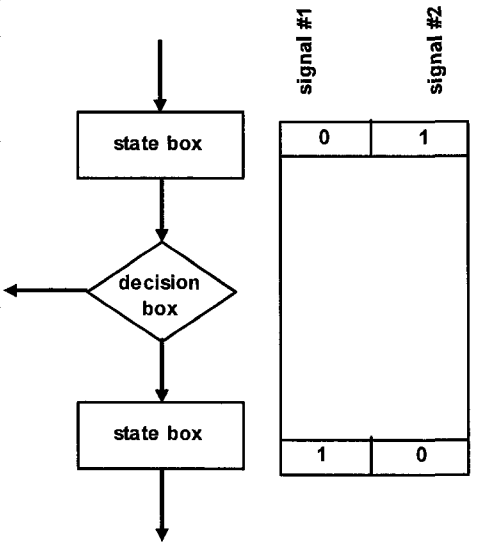


Figure 14. Algorithmic State Machine Example

3.2 Input Sector Design Overview

The input sector is one of the key entities for the packet switch. It provides both an external interface to the incoming packet stream and the central crossbar. The goal of the input sector is to receive the packets, decode the header to obtain the destination address, and temporarily store them in First-In First-Out (FIFO) elastic buffers, one for each virtual output sector queue (VOSQ). Figure 15 shows the block diagram of the input sector and specifically depicts the location of the linked-list memory manager. Two sets of elastic buffers are needed to connect to the link-list memory manager. These are required since the receivers, transmitters and header recognition sections and the linked-list manager section can operate at two different clock frequencies and therefore are asynchronous to one another. The elastic buffers, both the prestore and prefetch FIFOs, allow both the asynchronous circuit domains (header recognition, service matrix execution, and linked-list manager) to be interconnected with one another. Once the packets are stored into each of these elastic buffers, they are then taken and stored into their appropriate virtual output sector queue using the linked-list memory manager. The criteria for the operation of the linked-list manager is based on having the prestore FIFOs not empty and the prefetch FIFO not full. Once the storage operation is completed and packets are filled into the prefetch FIFO by the linked-list

manager, the service matrix provided by the global controller is decoded and packets are now taken and directed to each of the output ports connected to the crossbar. The Service Matrix Execution Block is responsible for fetching the correct number packets from each of the respective prefetch FIFOs and physically loading the transmitters. The details of the state machine and logic functions for the design of each element within the input sector are beyond the scope of this thesis.

For the purpose of the design of *"the hardware demonstrator"*, it has been chosen for simplicity to transmit the clock in parallel with the data. In order for each sector to synchronize to the transmitted clock, a phased-lock loop (PLL) is used. The received clock is then used as the transmitter clock at the output of the input sector. The transmitted data is clocked at a line rate of 10 Mbps; whereas, the internal clock frequency for the linked-list memory manager is at a clock frequency of 50 MHz. The 10 Mbps line rate was based on a design decision for the '*hardware demonstrator*' [Reference 26] & [Reference 27] while the 50 MHz frequency is the maximum reference clock frequency that is located on the hardware evaluation board. At higher line rates, the clock is extracted from the incoming data using a clock-data recovery circuit.

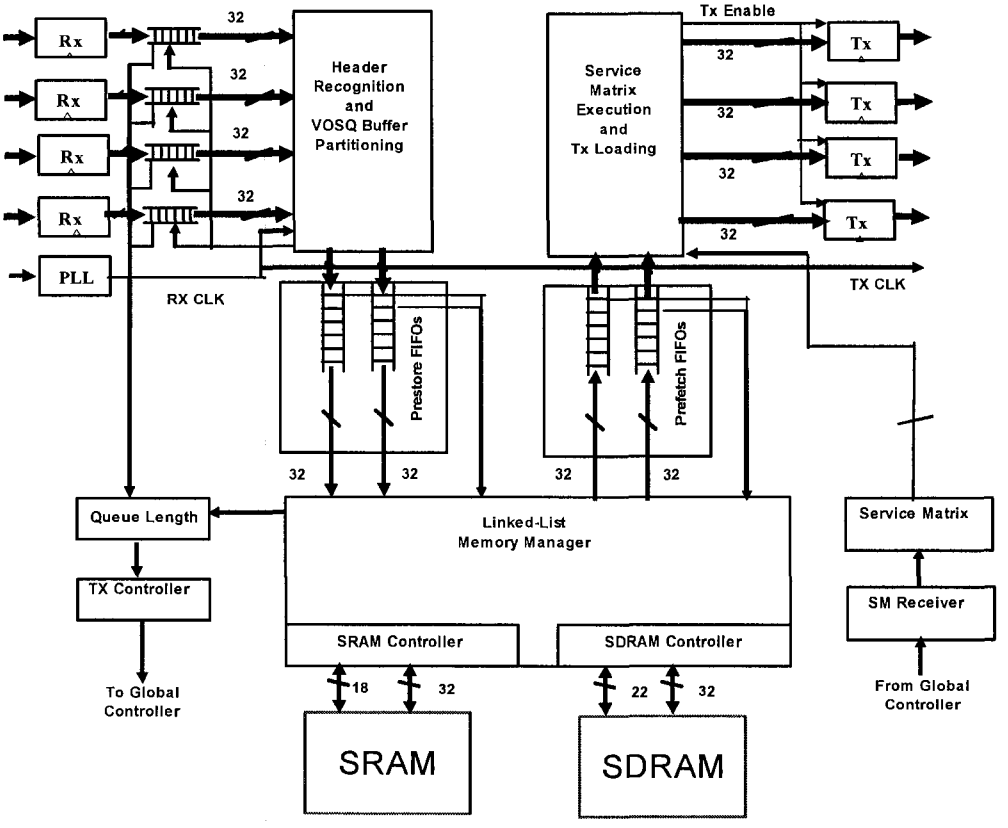


Figure 15. Top-Level Block Diagram of the Input Sector.
Communication to and from Global controller includes traffic matrix and service matrix information, respectively.

3.3 Linked-List Memory Manager Design

The linked-list memory manager is a digital finite state machine that dynamically creates and manages a list of packets for one VOSQ using a common shared memory. The shared memory for this design is Synchronous Dynamic Random Access Memory (SDRAM); chosen as a result of the hardware provided on an Altera^{TM} FPGA evaluation board. For this type of memory (SDRAM), only one read and write operation can occur at any given time. In order to identify the subsequent packets within the same list, pointers are used. Multiple sets of pointers identifying *idle* address locations, and *occupied* address locations for each VOSQ are stored in SRAM.

The algorithmic operation of the linked-list was provided in previous sections; however, this section will detail the System-on-Chip state machine required to realize such an operation. All state

machines will be based on the positive edge of a clock. Along with managing the pointers for a list of packets for a specific queue, the linked-list memory manager will also provide external control signals to calculate the length of the queue while the list changes in size. As highlighted in previous chapters SRAM is also located on the evaluation board but is only used to store pointers for the shared-memory due to its faster access times and low density. Using SDRAM for storing and retrieving pointers would significantly increase the total linked-list access time.

Figure 16 depicts the block diagram of the linked-list memory manager designed within an Altera^{TM} Stratix FPGA using the Avalon Bus. The Avalon Bus provides a bus interconnection between any master and slave peripheral provided that the interconnecting signals match the format of the bus interface. The *Master Input Logic* (one instance of this block is required for each VOSQ) controls the flow of information to and from the shared memory by obtaining either a queue pointer address or an idle address from the *Idle Address Manager* and updates the packet list for both read and write scenarios. The *Master Input Logic* then loads the *DRAM Address Generator* with the retrieved address value and generates as many consecutive 32-bit address locations needed to store an entire packet, which is dictated by the *Packet Fragment Counter*. Since the memory data width only supports a maximum of 32 bits, a *Packet Fragment Counter* logic block will assert a signal to indicate the completion of an entire packet to the *Master Input Logic* and the *DRAM Address Generator* blocks. Control signals are also connected between the *Master Input Logic* and the *DRAM Address Generator* in order to enable or disable the generator during the time packet data is being stored or retrieved from memory. The *Free Space Pointer* block is connected to the *Idle Address Manager* so that both the SRAM read and write pointers are updated during the same time. It is these pointers which determine the address within SRAM to which a valid starting packet data SDRAM address can be either provided too or sent from the *Master Input Logic* block.

When multiple master peripherals are connected to a slave peripheral, such as the case between the *Master Input Logic* and the *Idle Address Manager*, the design of the Avalon Bus^{TM} is such that it will create slave arbitration logic. The benefits of slave arbitration logic compared to bus arbitration are evident since for slave arbitration, the bus itself is not busy when other masters want to request information from other slave peripherals; only the one slave will be arbitrated, while for bus arbitration the entire bus is occupied with one master and the other master is left to wait. An

example of this would be when both *Master Input Logic* blocks, representing both VOSQs, wants to store or retrieve data to/from the SRAM or SDRAM simultaneously. For the purpose of this document, the arbitration between two *Master Input Logic* blocks will not be discussed; rather the focus will be on the design of the linked-list logic instead using a single block

The linked-list memory manager also contains controllers for interfacing with external memory. The SRAM controller slave peripheral to the bus will be described; however, the SDRAM controller slave peripheral will not be described in detail in this chapter because it is not trivial since the state machine must pay close attention to the correct timing based on the specific control signals such as Column Address Strobe (CAS), Row Address Strobe (RAS), and most importantly, the internal capacitance refresh that occurs on a periodic basis. For this reason the controller was taken from the Altera^{TM} System-On-Chip (SoC) library suite.

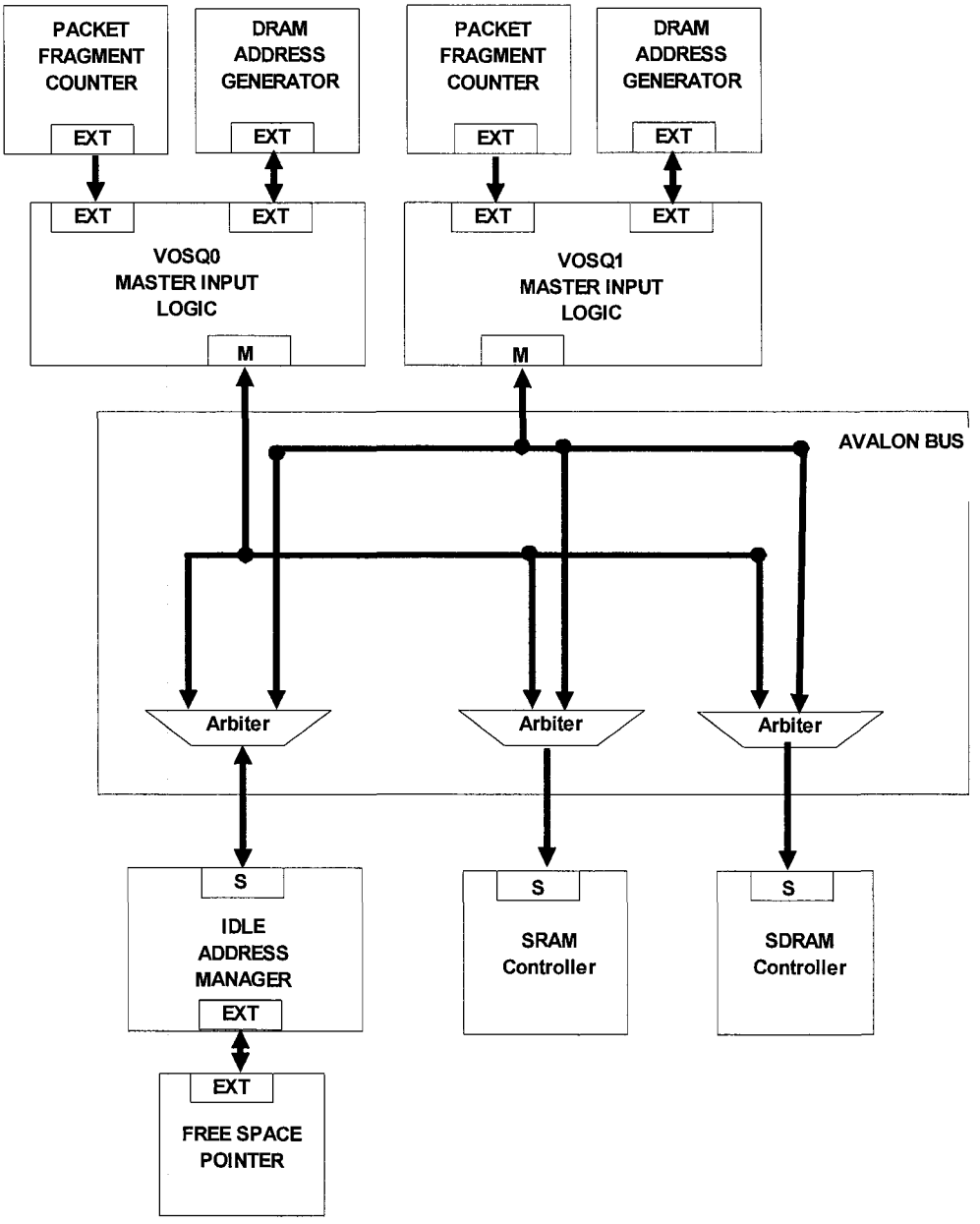


Figure 16. Linked-List Memory Manager SoC using the Altera Avalon Bus

The following sections will describe the details of each logic block that creates the linked-list memory manager to perform the required operations.

3.3.1 VOSQ Master Input Logic Peripheral

The *Master Input Logic* peripheral is designed to be the main interface to all other slave peripherals connected to the Altera™ Avalon Bus. It communicates with the *Idle Address Manager*, to extract address locations required for writing/reading packets into/from SDRAM. Packets are read from the Prestore FIFOs as long as they are not empty, and written to the Prefetch FIFOs as long as they are not full. This provides a constant stream of packet data flowing in and out of the queues directly to their desired central stage ports. The *Master Input Logic* peripheral monitors these signals after each read or write operation. Arbitration between reading from the prestore FIFO and writing to the prefetch FIFO is fair; that is, provided that the latter conditions are satisfied, a write will always occur followed by a read consecutively. Unfortunately, writing to the VOSQ and reading from the VOSQ do not take the same amount of time which is a direct result of the access time being different for the reading and writing operation on external memory.

In Figure 16, the external (EXT) signals connecting to the *Master Input Logic* peripheral represent signals that are not directly connected to the Avalon Bus™. The primary use of the Avalon Bus is to allow the connectivity to the off-chip memory via their respective memory controllers to the *Master Input Logic* blocks.

The algorithmic state machine for the *Master Input Logic* is separated into different functional operating modes:

1. The initialization mode of the 'queue' and 'idle' sections of the off-chip SRAM, explained in the previous chapter and the off-chip SDRAM with NULL values (i.e. values of hexadecimal values of 'FFFF').
2. The normal mode which stores and retrieves packets from SDRAM at an address location given by the Idle Address Manager.

In either of these functional sections, the logical block communicates with the *Idle Address Manager* which allows the *Free Space Pointer* block to update specific address pointers.

The state machine for the initialization is depicted in both Figure 17 and Figure 18. The initialization begins when the RESET is de-asserted. In the startup condition, both the SRAM and SDRAM must contain the appropriate contents as described in the previous chapter in order for the state machine to access the correct locations in memory for creating the linked-list for a particular

queue. The output signals interfacing the *Initialization Address Generator and Data Filler* allow the appropriate SRAM and SDRAM memory addresses and values to be returned. When the initialization is completed and the logical block returns to perform its normal operation of reading and writing packets to off-chip memory, a condition can arise when a significant amount of packets being written to the prefetch FIFO causes it to become full. This scenario can arise since the prefetch FIFOs memory size is significantly less than the size of the off-chip memory. If the prestore FIFO becomes full and the off-chip memory is not full, then the *Master Input Logic* state machine will continue to write to the off-chip memory until it also gets full or depletes the prestore FIFO. If the prestore FIFO is not emptied under these conditions, the *Free Space Pointer* block will reach its maximum idle address pointer value. The pointer value represents the last free address location returned to the *Idle Address Generator*. When subsequent packets under this scenario are written to off-chip memory the same pointer (address location) will be used, forcing previously stored packet data in that location to be overwritten others will be considered dropped.

By instantiating multiple *Master Input Logic* blocks; that is, implementing multiple VOSQs a problem can arise since each block will execute the same initialization states. In order to avoid this duplication, the logic block must also be designed with a bypass condition when the block is reset. To achieve this, both *MEM_INIT_COMPL* and *MASTER_ID* input signals are used (the interface signals for the Master Input Logic are labeled in Appendix D). If a given *Master Input Logic* block has a MASTER_ID asserted, this starts the process of initializing the SRAM to contain all the necessary SDRAM address values that are allowed to be written to, along with the locations of the head and tail SRAM pointer address values. When the main *Master Input Logic* (MASTER_ID = '1') is completed, the MASTER_MEM_INIT_DONE signal is asserted. Other *Master Input Logic* blocks (i.e. other VOSQs) are informed to bypass initialization and execute the normal operation for checking the prestore FIFO and prefetch FIFO full and empty status signals by connecting the MASTER_MEM_INIT_DONE signal directly to the MEM_INIT_COMPL signal on the other VOSQ Master Input Logic blocks and configuring the MASTER_ID signal on the other VOSQ block to be active low.

Figure 19 depicts the algorithmic state machine for storing the head and tail pointers. The interaction between the *Master Input Logic* and the SRAM occurs through the Avalon Bus and the

SRAM controller (slave peripheral). Since access to the SRAM is asynchronous, a finite amount of delay is required to access the off-chip memory data bus and hence the Master Input Logic is suspended in the current state until the delay is completed. A *Programmable Delay Timer* that contains the exact time delay is enabled by the *Master Input Logic* and an output signal from the delay timer is asserted when the delay has been reached. This operation also occurs when accessing the SDRAM.

The main sequence flow for the *Master Input Logic* state machine for reading and writing packets into external off-chip memory commences at the INIT state where the conditional check for the arbitration and the FIFO status signals takes place. Once the packets have been read from memory or written to in memory, and the necessary pointers are updated, the state machine returns back to the INIT state to start the sequence all over again. Figure 20 and Figure 21 depict the state flow for writing the first packet upon startup or reset. Figure 23 depicts the state flow for writing packets from the prestore FIFO into SDRAM and updating the tail and next pointers for the linked-list within SRAM. Figure 22 depicts the state flow for reading packets from the SDRAM into the prefetch FIFO and updating the head and next pointers for the linked-list within SRAM. Both require interfacing the *Idle Address Generator* which in turn sends the correct control information to the *Free Space Pointer* logic block to update both read and write offset pointers.

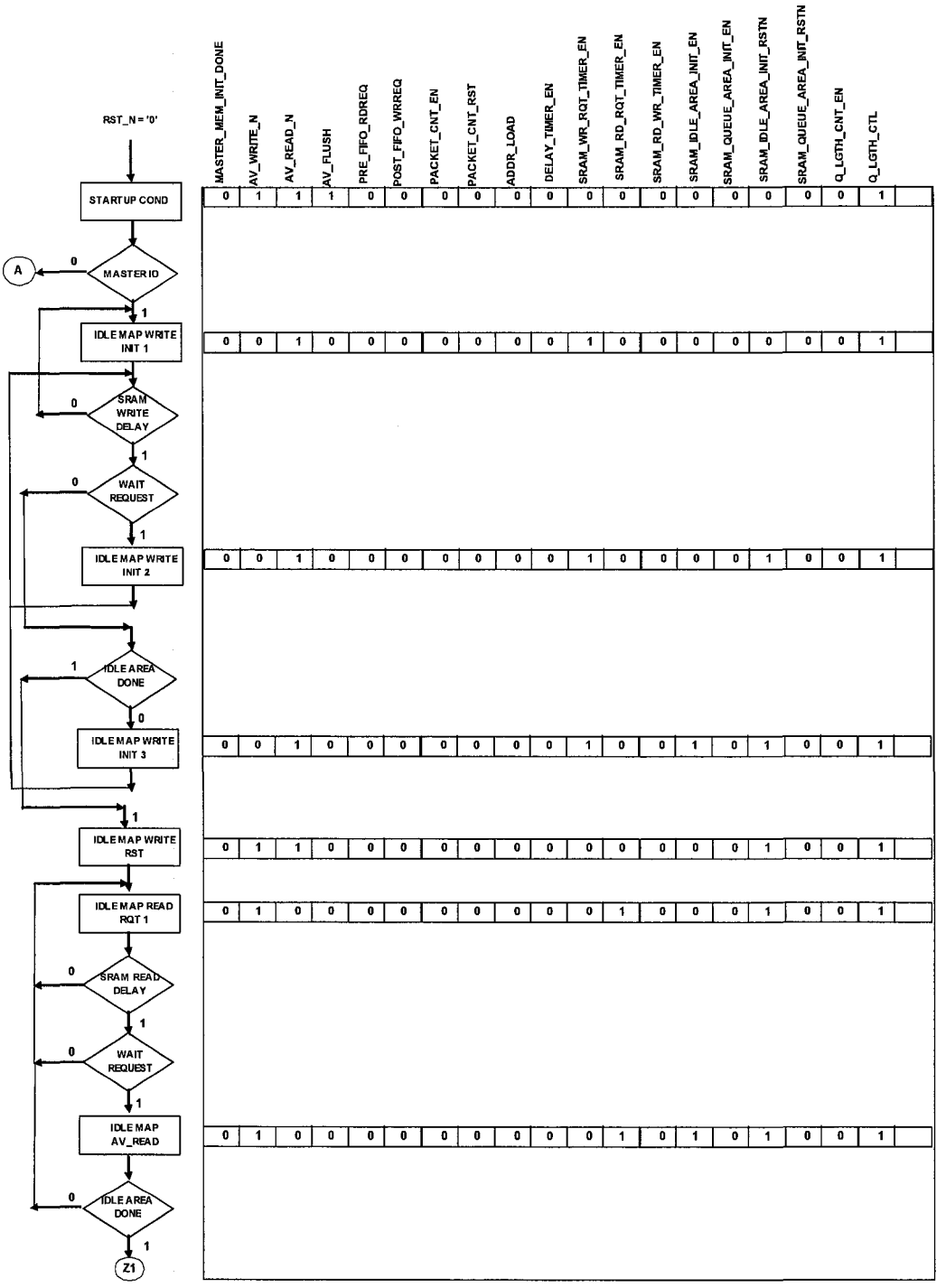


Figure 17. Algorithmic State Machine for the Master Input Logic initialization (part 1). This initialization is for both the idle and queue area

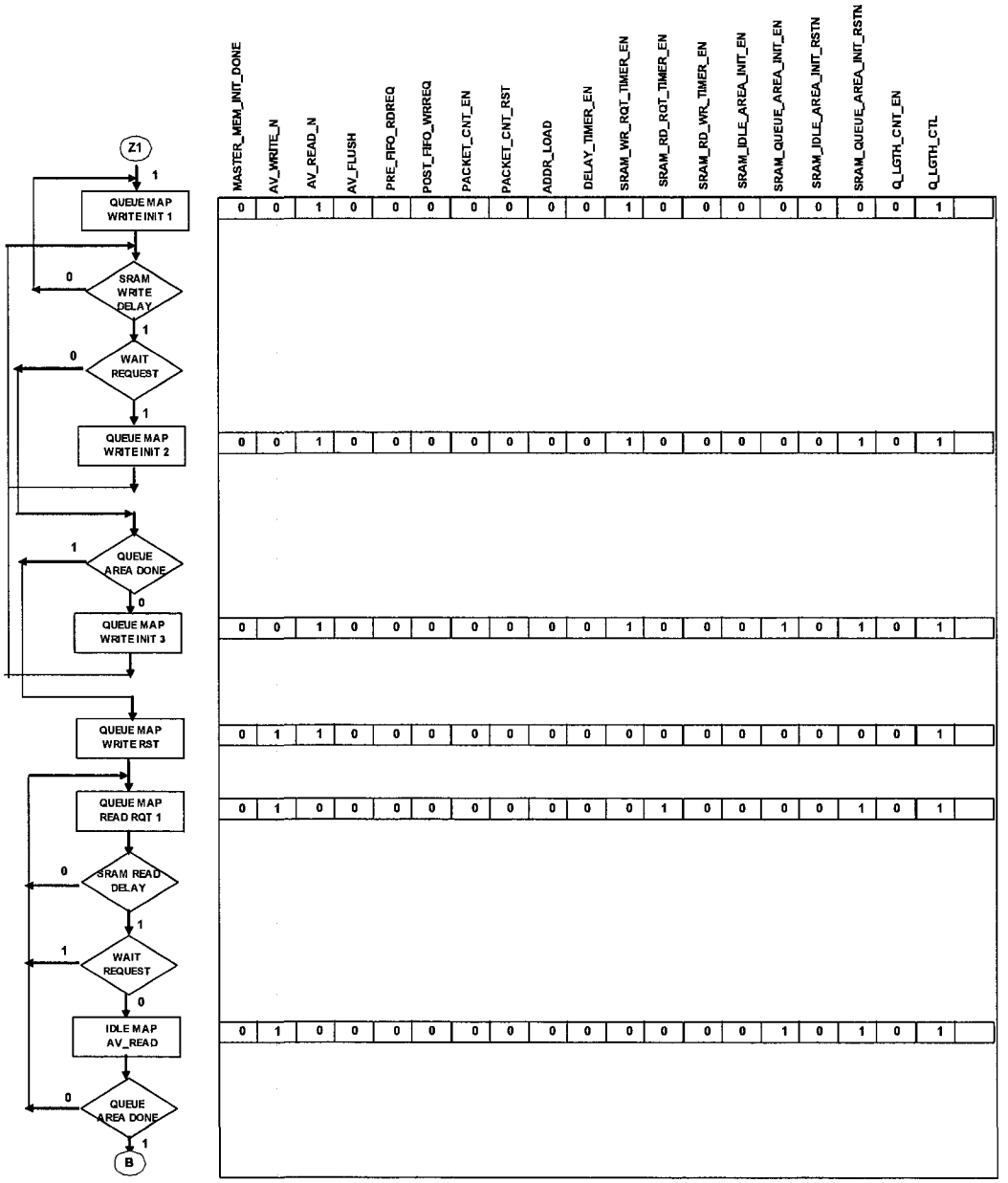


Figure 18. Algorithmic State Machine for the Master Input Logic initialization (part 2)
This initialization is for both the idle and queue area (Part 2)

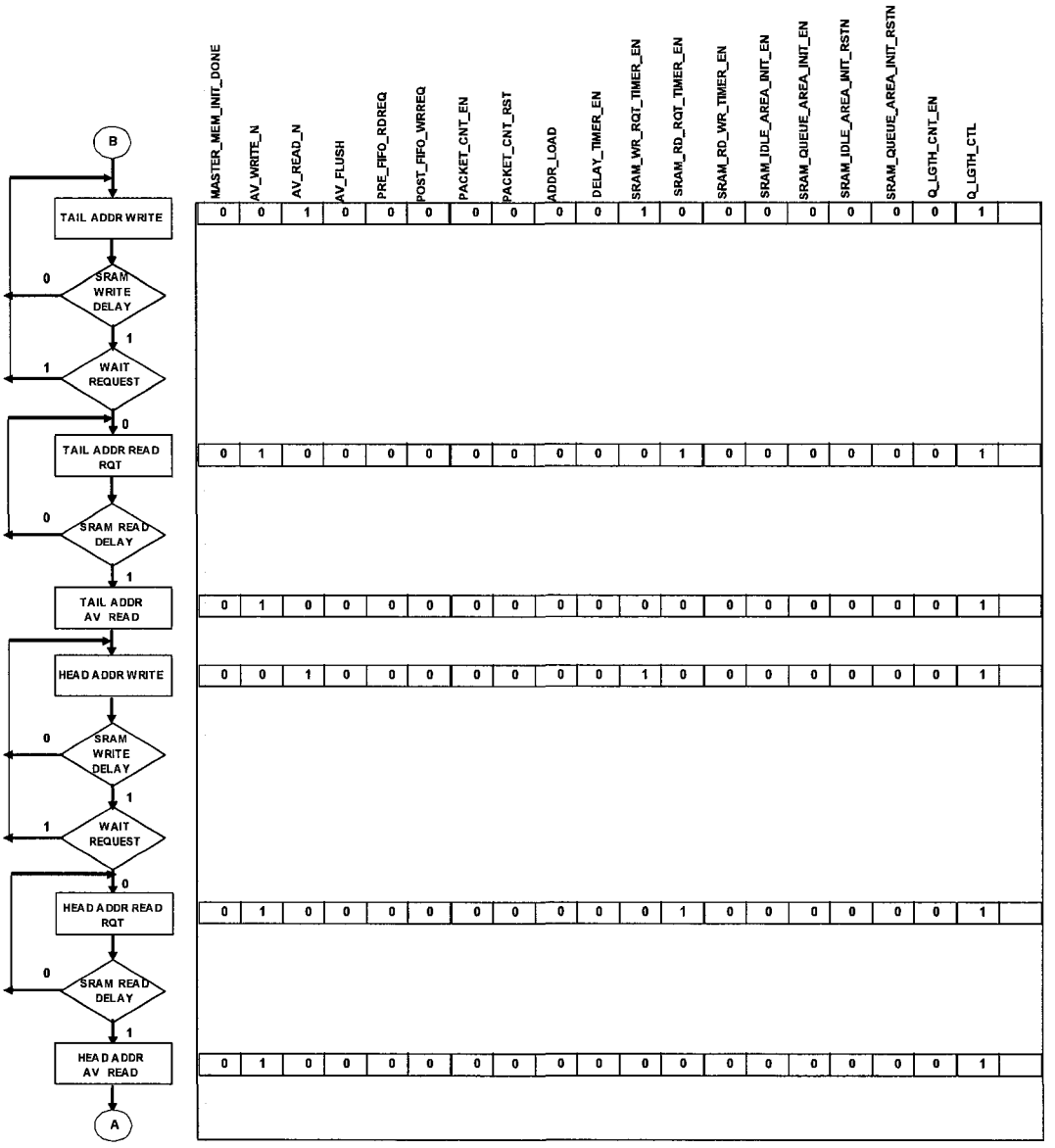


Figure 19. Algorithmic State Machine for the head and tail pointers initialization.

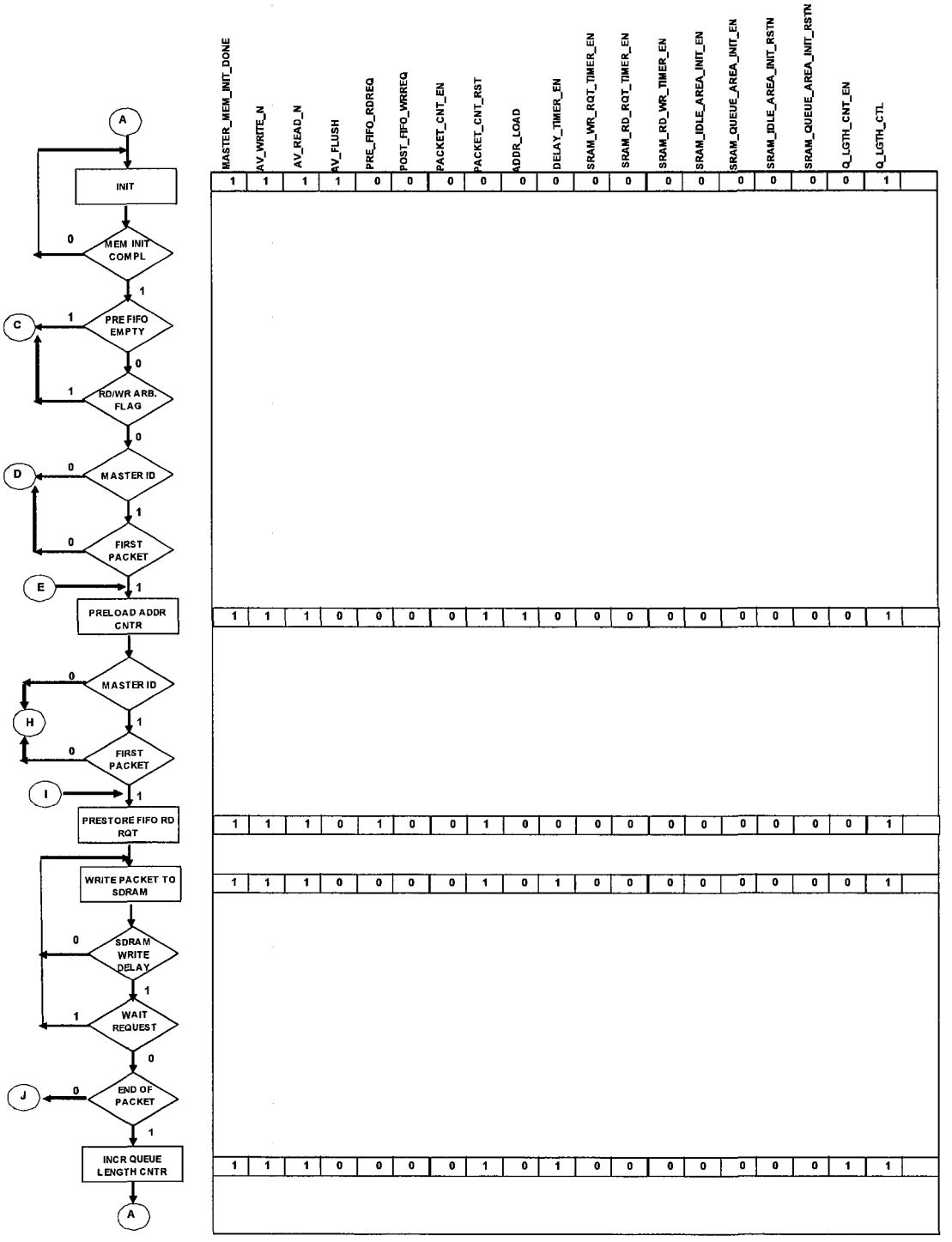


Figure 20. Algorithmic State Machine for writing the first packet (part 1).
This ASM for the Master Input Logic also updates both tail pointer and next pointer for the linked-list

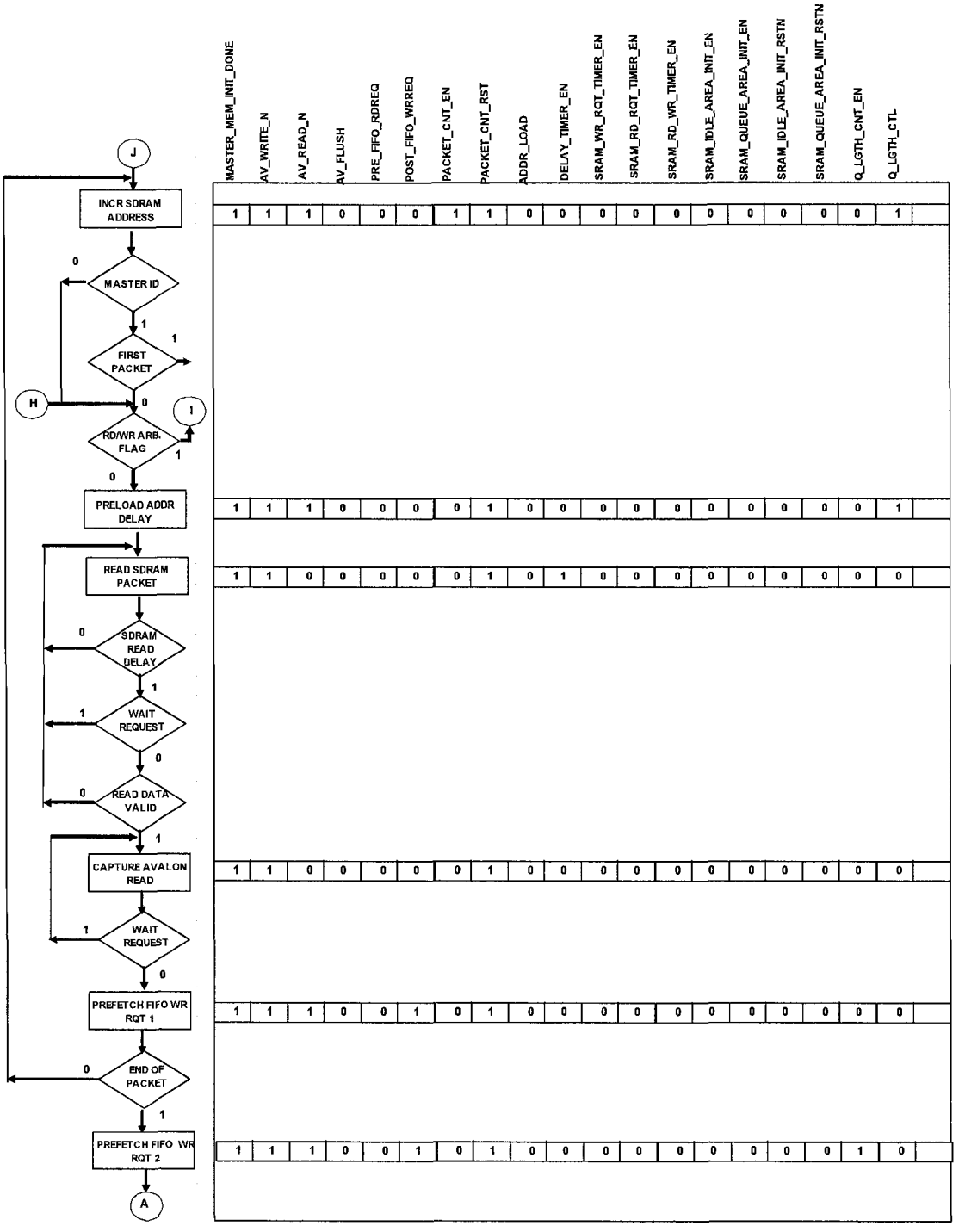


Figure 21. Algorithmic State Machine for writing the first packet (part 2).
This ASM for the Master Input Logic also updates both tail pointer and next pointer for the linked-list

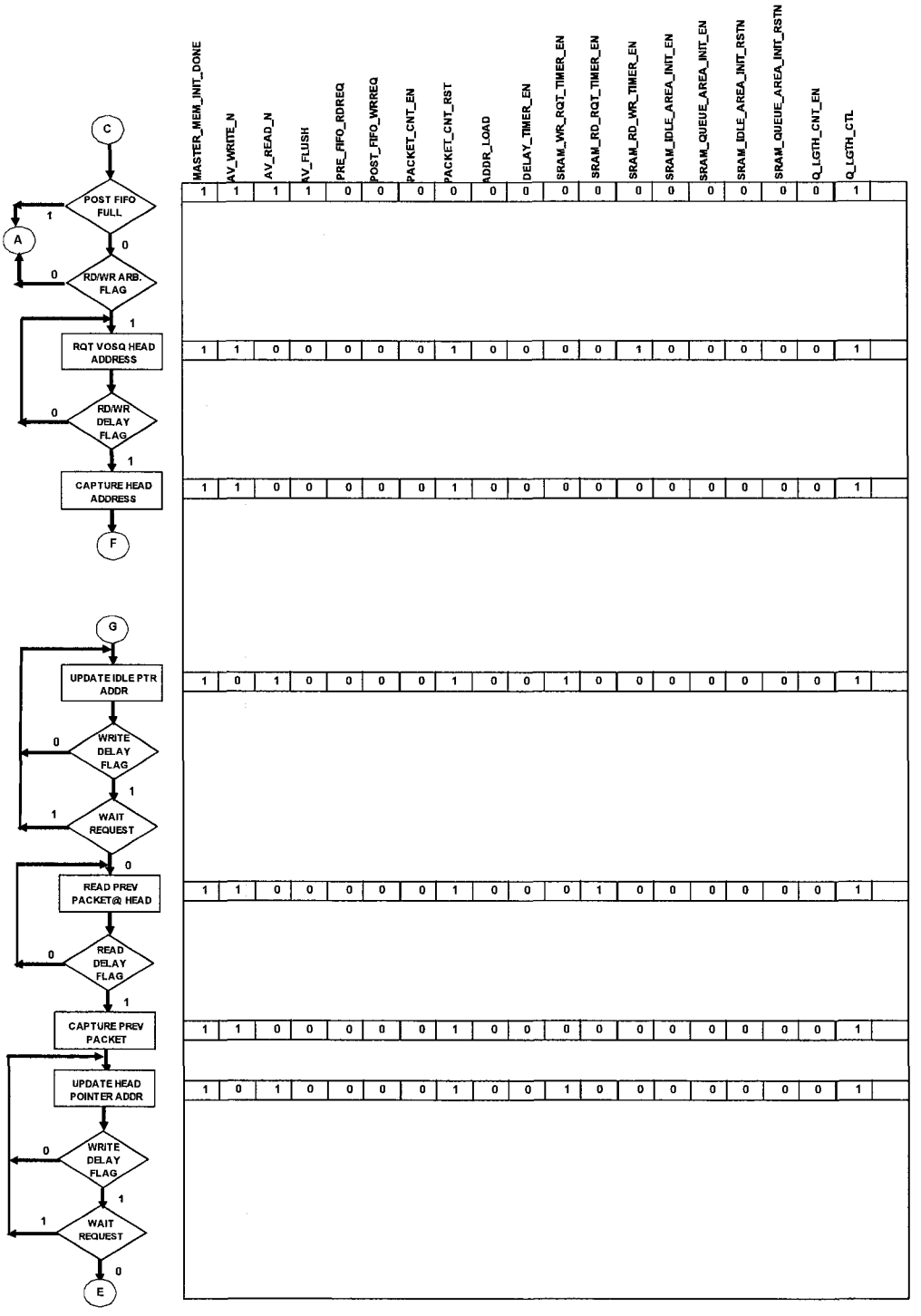


Figure 22. Algorithmic State Machine for reading a packet.
This ASM for the Master Input Logic also updates both head pointer and next pointer for the linked-list

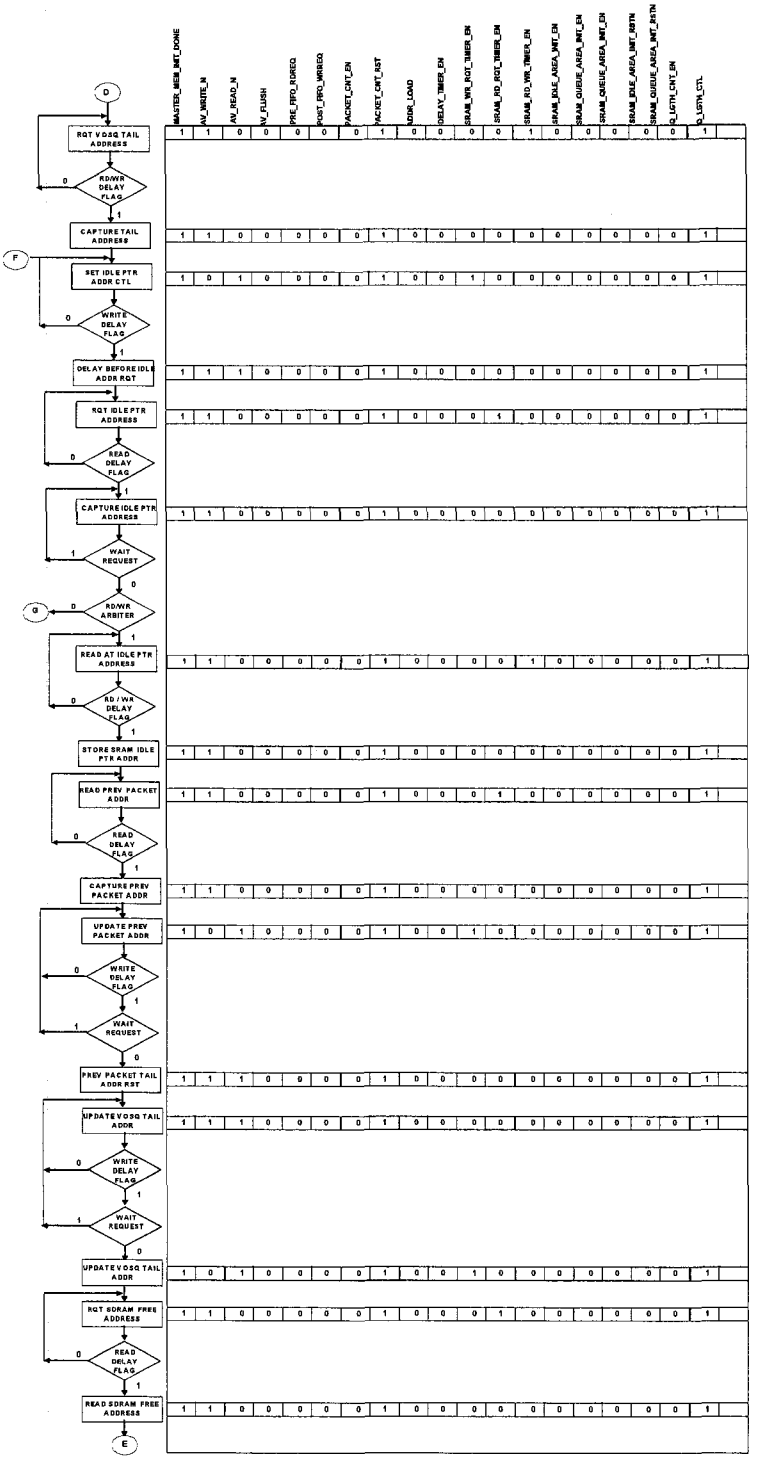


Figure 23. Algorithmic State Machine in normal operation for writing a packet.
This ASM for the Master Input Logic also updates both tail pointer and next pointer for the linked-list

3.3.2 Idle Address Generator Peripheral

The *Idle Address Generator* peripheral is designed to connect to the Altera^{TM} Avalon Bus and is responsible for returning to the *Master Input Logic* peripheral a valid address for the DRAM. This address will represent either a location in which a new packet will be stored or a location from which a packet will be read for a particular linked-list (VOSQ). It obtains the address information by communicating with the *Free Space Pointer* peripheral. Figure 24 shows the algorithmic state machine for the *Idle Address Generator* and the values for each of its external signals. It is triggered by an assertion of the RESET and it will monitor the Avalon Bus^{TM} until the CHIPSELECT signal is asserted. The Master Input Logic peripheral initiates a write request to the Avalon bus with the correct address of the Idle Address Generator. This triggers the assertion of the chipselect signal from the Avalon Bus which causes a control signal to be transmitted to the Idle Address Generator peripheral which subsequently enables the Free Space Pointer peripheral to retrieve a valid pointer address.

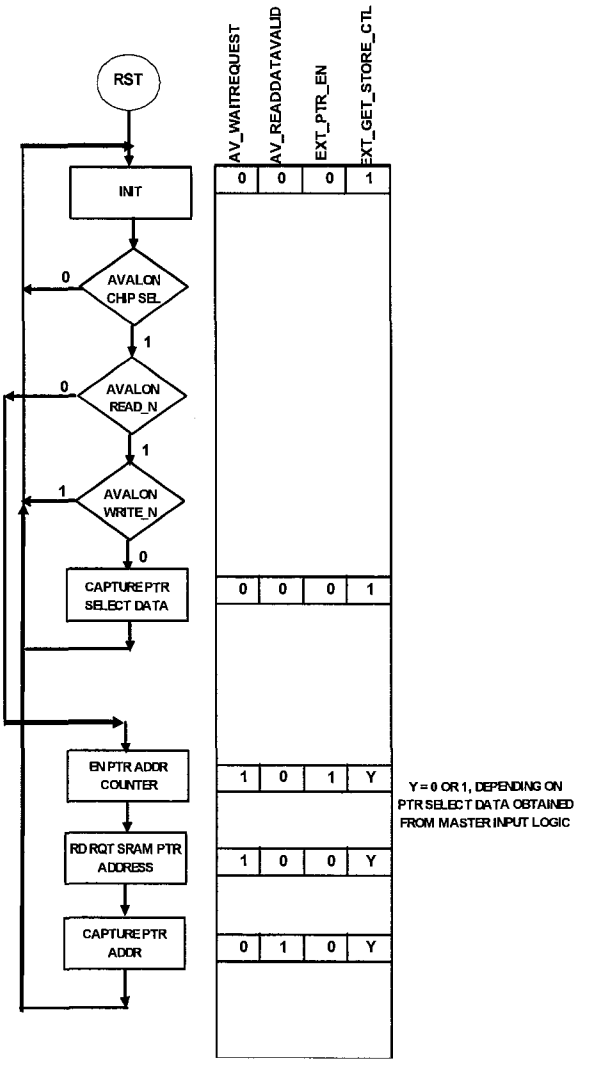


Figure 24. Algorithmic State Machine for fetching a valid SDRAM address.
This ASM for the Idle Address Generator identifies both scenarios of reading a packet or writing a packet for a particular VOSQ (linked-list).

3.3.3 DRAM Address Generator and Packet Fragment Counter

The *DRAM Address Generator and Packet Fragment Counter* peripherals are designed to connect directly to the *Master Input Logic* peripheral and are responsible for providing a DRAM address and a control signal, respectively. Since packets themselves can be larger than the memory bus width (32-bits), the *Packet Fragment Counter* is designed to assert a control signal to disable the *DRAM Address Generator* when the total number of segmented packet fragments has been

reached. The *DRAM Address Generator* is a 32-bit counter designed with a pre-load control, reset, enable, and a pre-loaded address; whereas, the *Packet Fragment Counter* is a n-bit counter designed with an enable, reset and a comparator whose output is asserted when the count sequence reaches the hard-coded fragment value set. Both the n-bit value and the hard-coded value are preset in the top level entity before compilation takes place and can be set according to the size of the packet. In this case it is set to a value of two since the packet size is 64 bits.

3.3.4 Free Space Pointer Peripheral

The *Free Space Pointer* peripheral is designed to connect directly to the *Idle Address Generator* peripheral and is responsible for returning an SRAM address. The SRAM address that is returned depends on the control signal sent to the peripheral. The control signal selects between two sets of pointers, a read and a write, that are implemented as 32-bit counters with enable, reset and a comparator that asserts a status signal when both pointers are equal. Figure 25 depicts the circuit diagram for the *Free Space Pointer* peripheral and shows both pointer counters, the multiplexer that selects between both counters and output flip-flops that register the output data for each positive clock edge. The *Free Space Pointer* peripheral is designed such that a write pointer is always returned under a reset condition. Since a packet is first taken from the prestore FIFO to be written in memory and later forwarded to the prefetch FIFO, a write pointer is always required at reset rather than a read pointer. For this reason a control signal named FIRST_WRITE is created within the write pointer block to guarantee that this always occurs. Doing this allows both FULL_STATUS and EMPTY_STATUS signals, which refers to the shared-memory space (SDRAM) required for all packets, to be properly initiated.

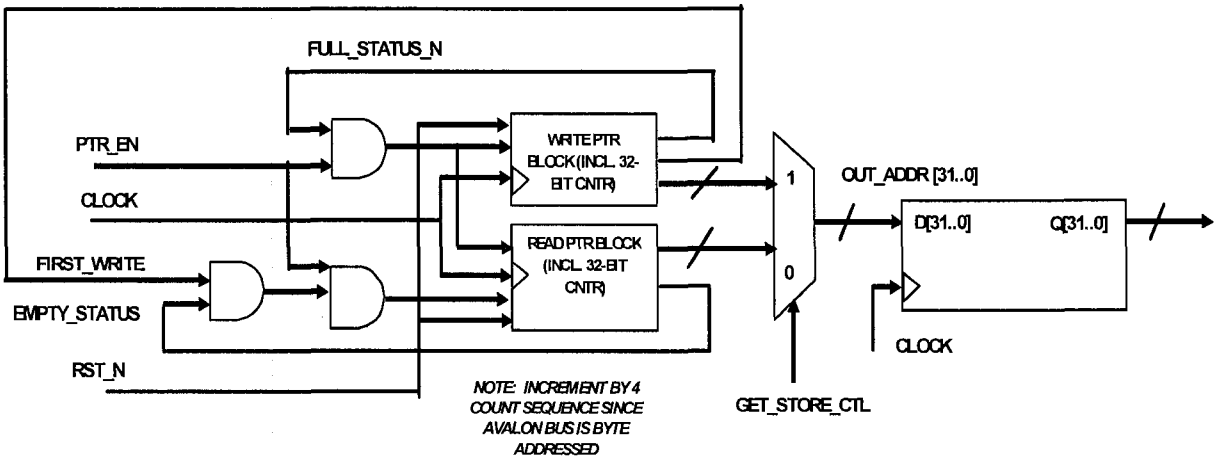


Figure 25. Logic circuit diagram of the Free Space Pointer

3.3.5 SRAM Tristate Memory Controller Peripheral

The *SRAM Memory Controller* peripheral is designed to connect the Altera^{TM} Avalon Bus and the off-chip SRAM. It is responsible for asserting the control signals on the off-chip memory making sure that the timing constraints needed to perform both read and write operations are respected. A tristate port is needed for the output data signal because the off-chip SRAM uses the same data port for both read and write operations. Unfortunately a tristate port will incur additional delays when reading or writing to the off-chip memory device. Figure 26 depicts the circuit diagram for the memory controller. Logical '1's and '0's within the multiplexer ports represent the FPGA core voltage Vcc and GND respectively. Reference 21 provides the details of the design and operation of the SRAM and provides the interface signals. When the *Master Input Logic* sends a write or read request to the SRAM controller, the CHIPSELECT_N signal is monitored and as soon as it is de-asserted, the controller will immediately assert the WAITREQUEST for a duration of 2 clock cycles in order to comply with the external control-to-data output signals (i.e. WE_N, and OE) timing needed for the off-chip memory.

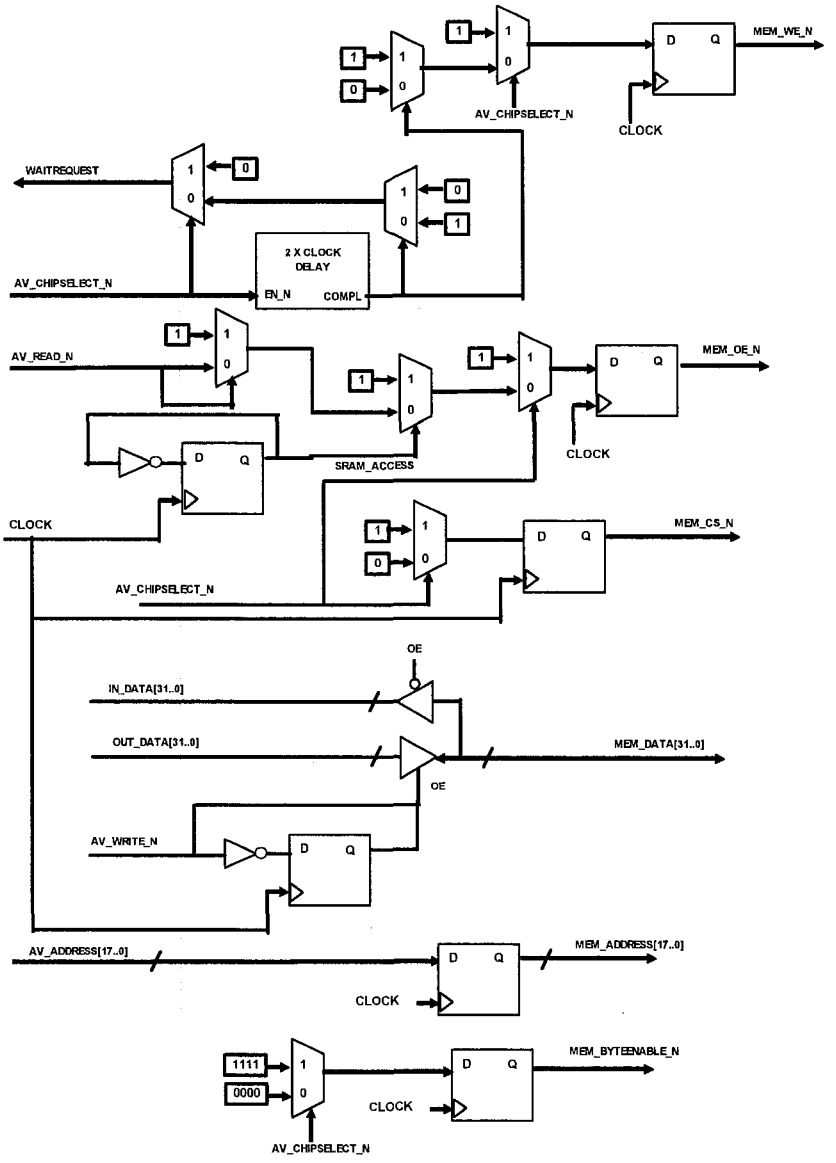


Figure 26. Logic circuit diagram of the SRAM Tristate Memory Controller Peripheral

3.4 Test Bench and Simulation Results

The design of the linked-list memory manager has been implemented using a 0.13 μm process Altera^{TM} Stratix FPGA EPS10F780C6 family that contains up to 10,570 logical elements and up to 426 I/O pins. Some of the key peripherals are connected directly to the Avalon Bus by using the System-On-Programmable-Chip (SOPC) tool within the Quartus Simulator. These peripherals

include the *Master Input Logic*, the *Idle Address Generator*, the *Avalon Bus*, the *SDRAM controller*, and finally the *SRAM controller*. Each of the logical blocks are programmed using VHDL (Very High Speed Integrated Circuit Hardware Description Language). The Quartus simulator has built-in synthesis in order to create the logical elements and calculate the worst case propagation delays for each of the signals routed between Logic Array Blocks (LABs) and to the I/O ports. An evaluation board with off-chip memory was used for the design of the linked-list memory manager. Each of the peripherals has been individually compiled within Quartus in order to calculate on-chip timing; whereas, the behavioral simulation of the entire linked-list memory manager has been compiled using Modelsim SE 6.0 by Mentor Graphics Corporation. Only the behavioral simulation will be detailed in this section and for the thesis.

Since the linked-list memory manager design is based primarily on the use of off-chip memory, external behavioral and timing models for the off-chip memory must be used to construct the system test bench. This test bench using external models will help to reduce the hardware debugging when the design is synthesized onto the FPGA. The models for both the SDRAM and SRAM have been taken from Reference 22 and Reference 23, respectively.

Table 1 depicts the numerical values used for all peripherals within the linked-list memory manager and the off-chip memory.

Table 1. Simulation parameters for the Linked-List Memory Manager

Signal Name	Value	Units	Description
FPGA Oscillator Reference Clock Frequency	50	MHz	Clock frequency used for the linked-list memory manager. Reference clock is a crystal oscillator
SDRAM Oscillator Reference Clock Frequency	50	MHz	Clock frequency used for the SDRAM memory. Output directly from the FPGA, not directly from the crystal oscillator reference.
SDRAM address bus row width	12	bits	Row size for the SDRAM off-chip memory
SDRAM address bus column width	8	bits	Column size of the SDRAM off-chip memory

Table 1. Simulation parameters for the Linked-List Memory Manager

Signal Name	Value	Units	Description
SDRAM banks	4	banks	Number of banks within the SDRAM
SDRAM data width	32	bits	Number of bits that represents a storage cell within the memory
SDRAM CAS latency	3	cycles	The Column-Address Strobe (CAS) is the number of clock cycle delay between a read and the appearance of the data
SDRAM max. clock frequency	143	MHz	Maximum clock frequency at which the SDRAM will operate (speed grade -7 is used)
SDRAM power-up delay	100	µs	Delay required at reset before any commands are issued to the memory
SDRAM refresh cycle period	15.625	µs	Memory capacitive refresh period
SDRAM refresh duration (t_{rfc})	70	ns	Period at which the refresh commands are issued
SDRAM Precharge duration (t_{rp})	20	ns	Time period from an assertion of a precharge command to an assertion of an active command. Active command opens a row within a specific bank to be accessed during a read or a write operation
SDRAM Active-to-Read or Write Delay (t_{rcd})	20	ns	Time period from an assertion of an active command to an assertion of either a read or a write command
SDRAM Access Time (t_{ac})	5.5	ns	Time between the assertion of a read command and the time when the data is available and is placed on the bus. The data is made available after this time.
SDRAM Write Recovery Time (t_{wr})	14	ns	Time from the assertion of a write command to the time when a precharge command is asserted to deactivate the row where the data is being stored. The data is required immediately when the write command is asserted
SRAM Address Width	18	bits	The total width of the address bus on the off-chip SRAM

Table 1. Simulation parameters for the Linked-List Memory Manager

Signal Name	Value	Units	Description
SRAM Data Width	32	bits	The total width of the data bus on the off-chip SRAM. There are 2 x off-chip SRAMs that share an address bus and are connected via the OE signal
SDRAM Controller Avalon Address Range	0x00000000 to 0x00FFFFFF	hex	The Avalon bus address range used by the SDRAM controller slave peripheral
SRAM Tristate Controller Maximum Avalon Address Range	0x01000000 to 0x013FFFFF	hex	The Avalon bus address range used by the SRAM Tristate controller slave peripheral
SRAM Idle Address Range	0x01010000 to 0x0101FFFC	hex	The Avalon bus address range that defines the shared idle addresses to be used for a particular Master Input Logic block (i.e. each VOSQ)
SRAM Queue Address Range	0x01000000 to 0x0100FFFC	hex	The Avalon bus address range that defines the location of the SDRAM addresses which that be used to write and read packets to/from a particular linked-list. For writing these are free addresses and for reading these are occupied address locations
Idle Address Generator Avalon Address Range	0x01400000 to 0x01400003	hex	The Avalon bus address range used by the Idle Address Generator slave peripheral.
Packet Length	64	bits	The total size of the packet including the overhead bits. Since the data width for the memory is 32 bits, the fragmented value used by the Packet Fragment Counter is equal to packet_length / memory_data_width = 2

Figure 27 shows the entire test bench diagram for the linked-list manager. The test bench includes a packet generator logic (simple counter) that writes the data directly into the prestore FIFO and flows this information to the input of the linked-list manager. The linked-list manager then takes the packet, append it into a particular list using shared-memory and updates all required pointers

to this list. The output of the linked-list manager writes the data taken from a queue list into the prefetch FIFO. The test bench is created to resemble the exact interface of the input sector to demonstrate its functionality. Each of the prestore and prefetch FIFOs have 32-bit wide words and can contain up to 128 words. Each are designed with active high 'read' and 'write' enable signals as inputs and 'empty' and 'full' signals as outputs. Each FIFO can asynchronously 'read' and 'write' by using different clock frequencies; however, for this test bench the clock frequencies were kept identical.

Figure 28 and Figure 29 depict the behavioral timing diagram for transferring packets from the Prestore FIFO to the linked-list (VOSQ). The idle address is retrieved and all necessary pointers for the linked-list are updated.

Figure 30 to Figure 32 depict the behavioral timing diagram for transferring packets from the linked-list (VOSQ) to the Prefetch FIFO. The idle address is returned back to the Free Space Pointer and all necessary pointers for the linked-list are updated.

The timing diagrams associated with the initalization of the Linked-List Memory Manager can be found in Appendix C.

Table 2 shows the measured performance times for a packet length of 64 bits for the following operations: initialization of off-chip memory, writing the first packet and updating pointers into a VOSQ, normal operation of writing other packets and updating pointers into a VOSQ and finally normal operation of reading packets and updating pointers from a VOSQ.

Table 2. Linked-List Memory Manager Timing Analysis

Parameter	Time (ns)	Time relative to Tclk
Linked-List Initialization		
>> *SRAM Idle Region Write*	655,440	322,772
>> *SRAM Idle Region Read*	1,310,820	65,541
>> *SRAM Queue Region Write*	655,440	322,722

Table 2. Linked-List Memory Manager Timing Analysis

Parameter	Time (ns)	Time relative to Tclk
>> *SRAM Queue Region Read*	1,310,820	65,541
>> *SRAM Head Address Write*	40	2
>> *SRAM Head Address Read*	80	4
>> *SRAM Tail Address Write*	40	2
>> *SRAM Tail Address Read*	80	4
Writing the first packet & updating pointer references	240	12
Transferring packets from the Prestore FIFO to the VOSQ and updating pointer references	880	44
Transferring packets from the VOSQ to the Prefetch FIFO and updating pointer references	1040	52

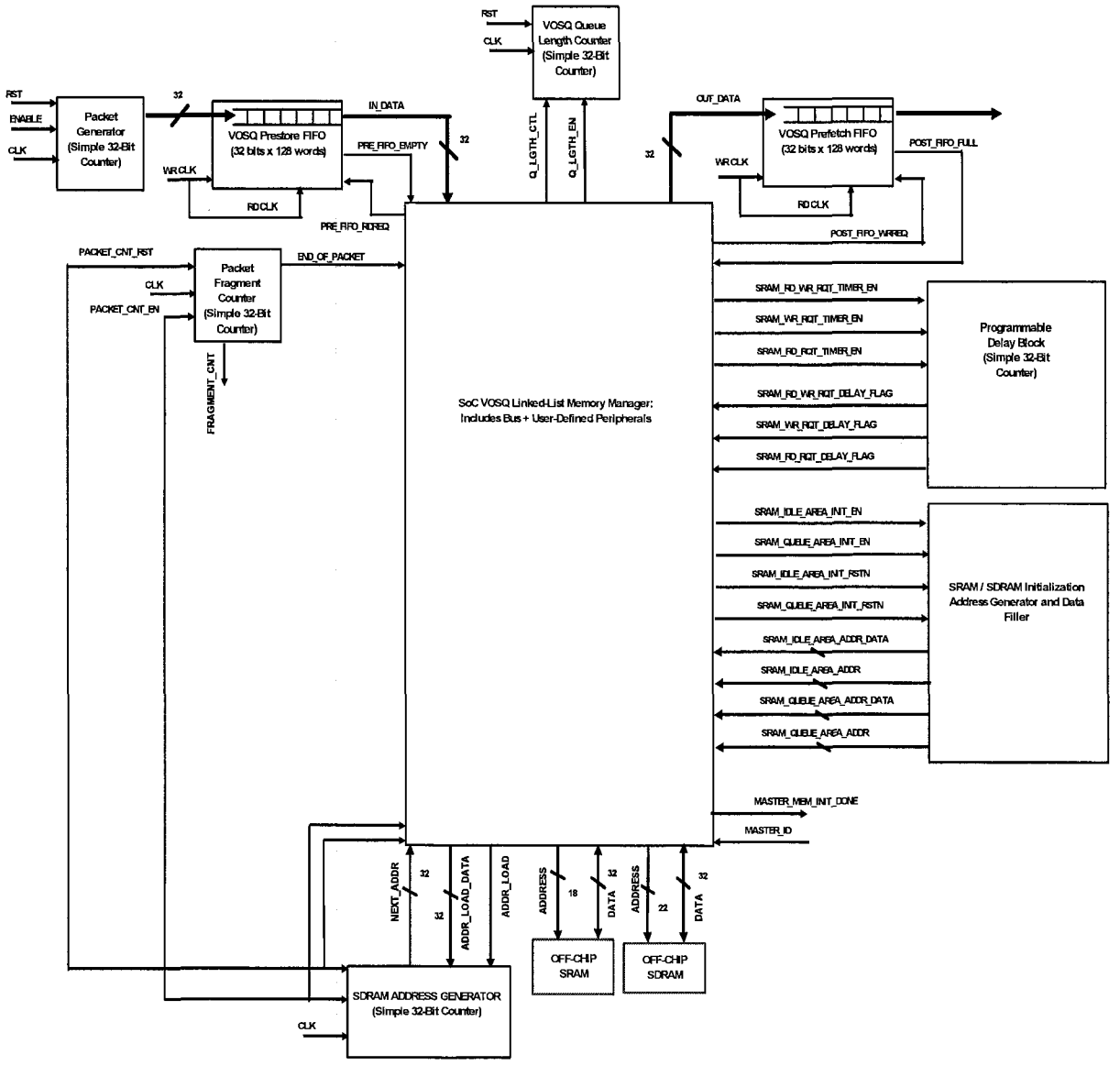


Figure 27. Linked-List Memory Manager test bench diagram.

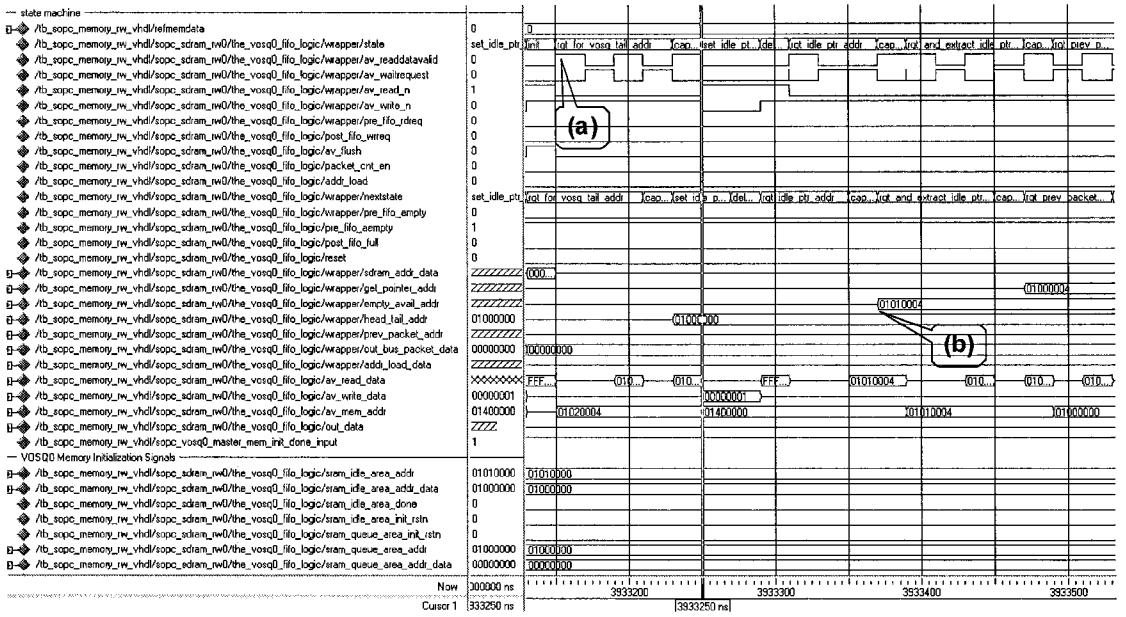


Figure 28. Timing diagram for reading from the Prestore FIFO to the VOSQ (part 1).
(a) Depicts the beginning of the state machine for writing a packet and the request for the VOSQ tail address. (b) Depicts the request for the idle address obtained using the Free Space Pointer from SRAM.

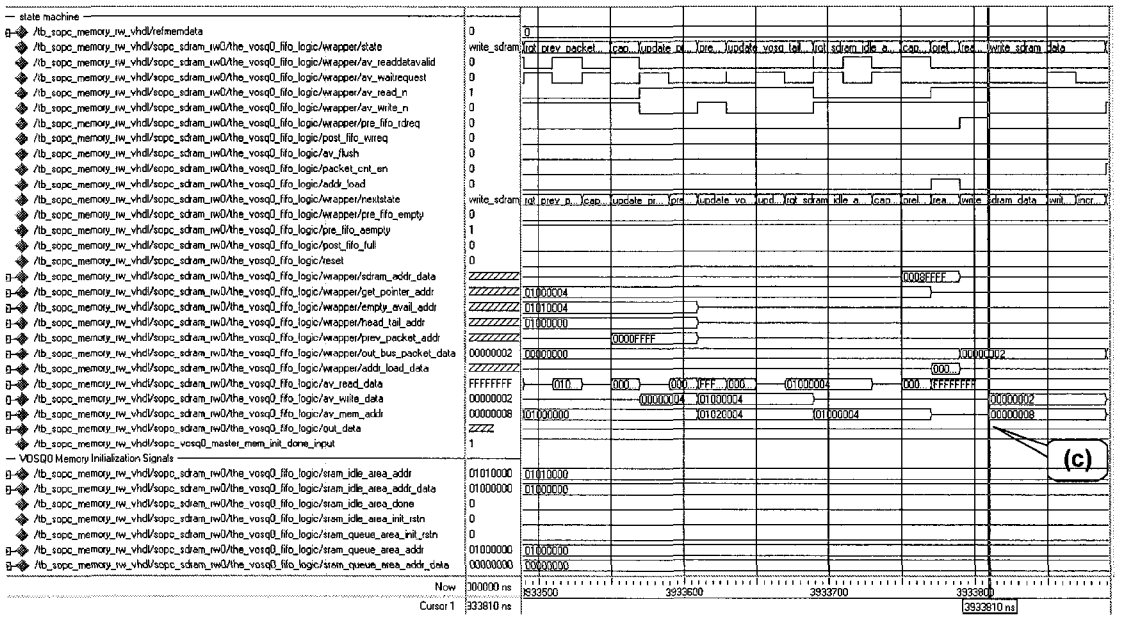


Figure 29. Timing diagram for reading from the Prestore FIFO to the VOSQ (part 2).

(c) Depicts the next packet fragment from the Prestore FIFO which is to be written into the SDRAM address location obtained from the Idle Address Generator. In the sequence of extracting data from the Prestore FIFO, this represents the first packet fragment of the second packet. This shows that at address = '0x00000008', the FIFO data extracted from the FIFO is '0x00000002'. The address is 4 bytes offset from the previous to comply with the Avalon Bus addressing requirements.

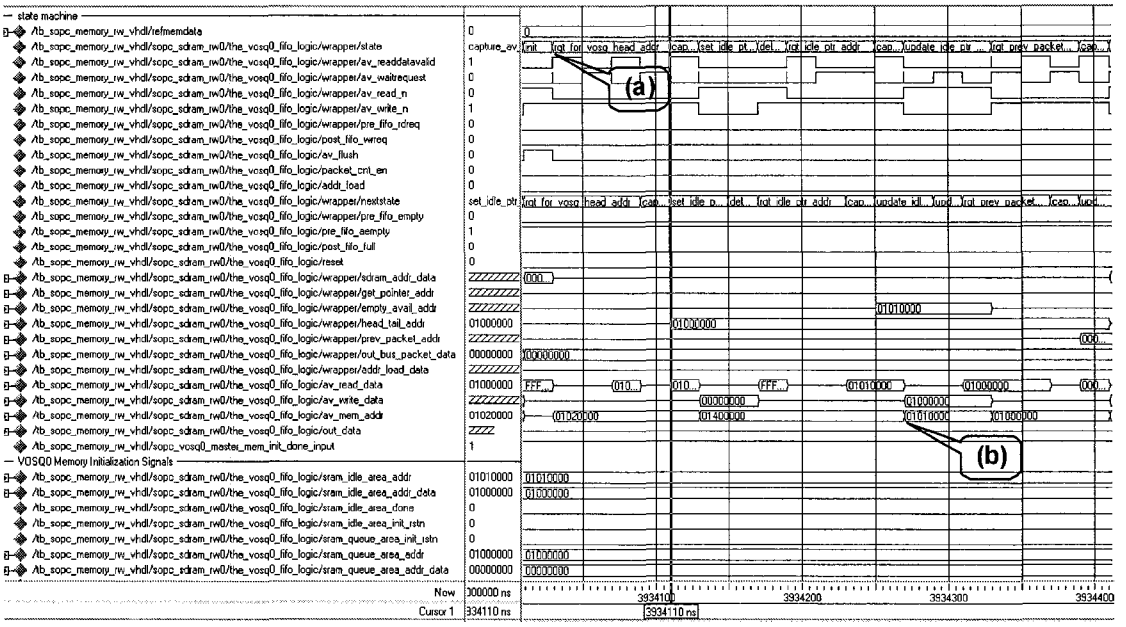


Figure 30. Timing diagram for writing packets from a VOSQ to the Prefetch FIFO (part 1).

(a) Depicts the Master Input Logic requesting for the VOSQ head address. (b) Depicts the Free Space Pointer being updated with the VOSQ head address. Since it is shared-memory, this address is reused for other incoming packets. Here the idle address '0x01010000' in SRAM will be populated with the idle pointer '0x01000000'.

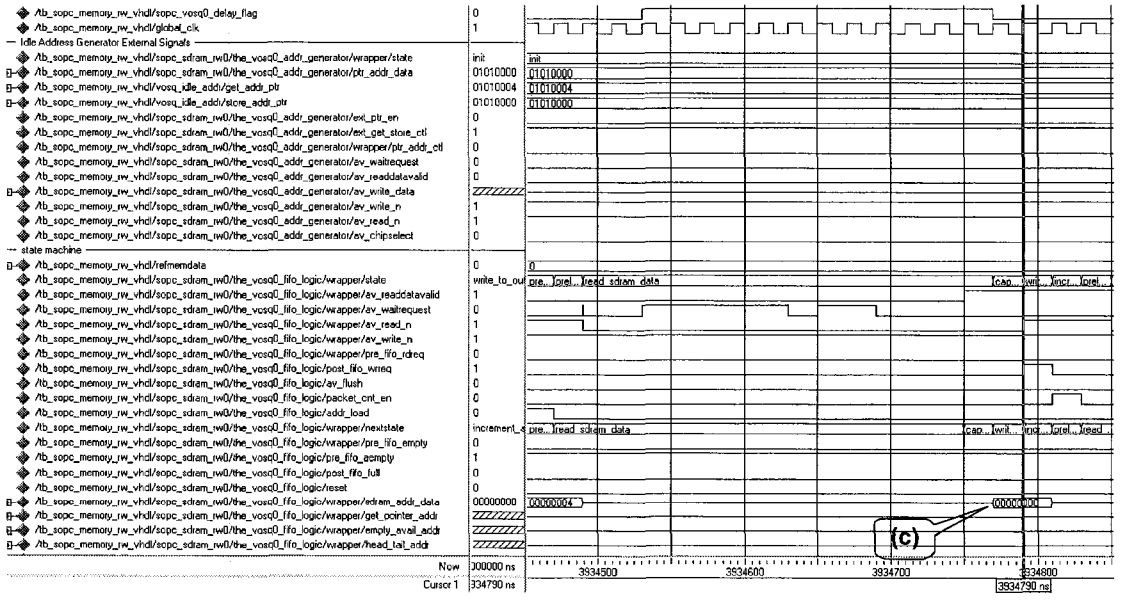


Figure 31. Timing diagram for writing packets from a VOSQ to the Prefetch FIFO (part 2).

(c) Depicts the packet fragment that has been read from SDRAM for a particular VOSQ. The data read is the first fragment of the first packet = '0x00000000'

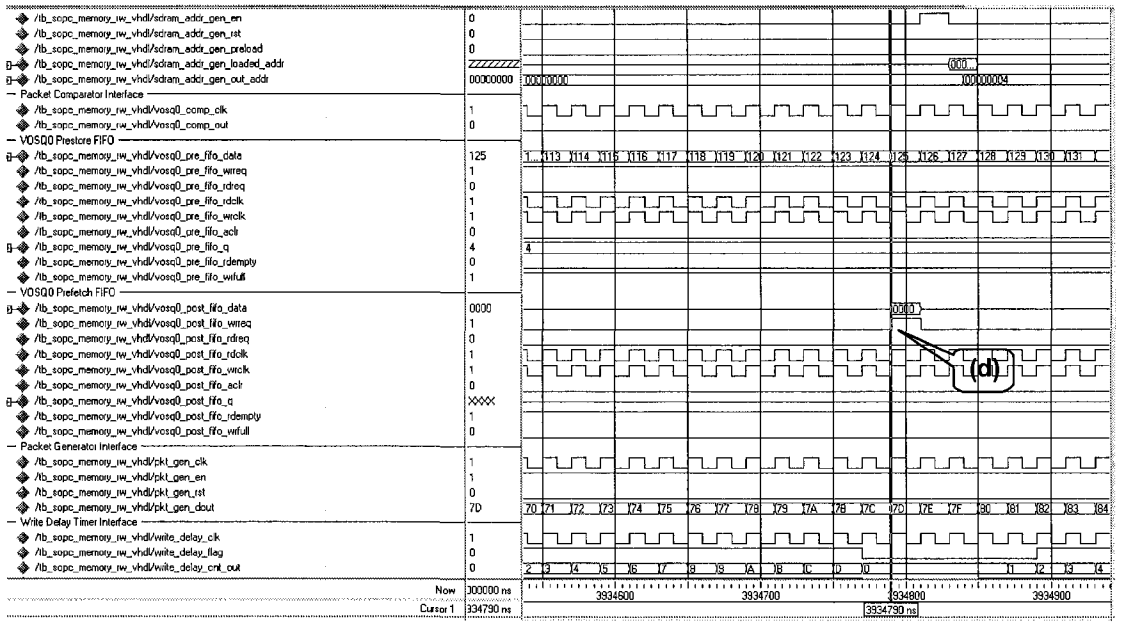


Figure 32. Timing diagram for writing packets from a VOSQ to the Prefetch FIFO (part 3).
(d) Depicts the first fragment of the first packet '0x00000000' being written into the Prefetch FIFO. The write request for the Prefetch FIFO is asserted by the Input Master Logic

Chapter 4 Optical Interconnect

In the previous chapters we've seen the architecture for the Clos-like architecture to be used for the design of the packet switch with an optical core. This architecture requires high bandwidth interconnections to service the queues at the input sectors for fast transport to the output sectors. With the large bandwidth capabilities provided by photonics, current technologies are available to provide this fast transport between the electronic sectors. Since the nature of the Flexible Bandwidth Provision scheduling algorithm requires setting up variable bandwidth paths between sectors, one of the potential candidates as the interconnect technology to perform this operation is the Wavelength Selectable Switch (WSS). This technology can be used to conceptually create a stack-up low port count switches creating a non-blocking switching fabric that can allow a fixed number of paths to be provisioned between sectors; however, achieving such functionality does not go without suffering physical penalties. There are definite physical impairments that exists when creating the switching fabric which inhibits idealistic transmission. In this chapter the impairments that exist for this type of interconnect are addressed. Furthermore, the chapter will detail the types of impairments, describe the measurement setup required to measure these impairments and finally show the actual measured results.

4.1 Overview of Physical Optical Parameters for an Optical Interconnect

There are several optical impairments that limit the transmission along a light path formed by the interconnect; specifically when the light path contains components or devices that have spatial dependencies. The WSS technology is created such that different wavelengths of light travel to specific broadband 1 X N switches which guide a specific wavelength from an input port to an output port. Each wavelength is extracted from the connection through a demultiplexing filter which has output ports that are tuned to each color of light. The impairments that are important for this interconnect can be summarized as follows:

a) Insertion Loss
b) Wavelength Isolation/Port-to-Port Isolation
c) Interferometric (inband) crosstalk

d) Polarization Mode Dispersion

e) Polarization Dependent Loss

f) Switching or Reconfiguration Time

g) Return Loss

These parameters will affect the quality of the signal once it is received at the output sector. The power margin can be significantly compromised if the interconnect performance is not carefully designed. Clearly these parameters are not anything new in the world of optical communications but they are relevant when describing optical interfaces for the architecture of this packet switch. Other types of technologies can be used for creating the optical switch fabric; however, the subset of impairments described here are a minimum for describing the optical interconnect and apply to any technology. The following diagram demonstrates the block diagram with the use of wavelength selectable switch technology using component technology developed by a company named Metconnex (product name: WSS5400 [Reference 1]). The fabric contains two of the modules connected through their common port creating a 9 x 9 non-blocking switching fabric; Only an 8 x 8 configuration is required for "the hardware demonstrator" but all 9 ports were analyzed. Therefore the definitions for the impairments will discuss both configurations - the individual modules and the cascaded pair.

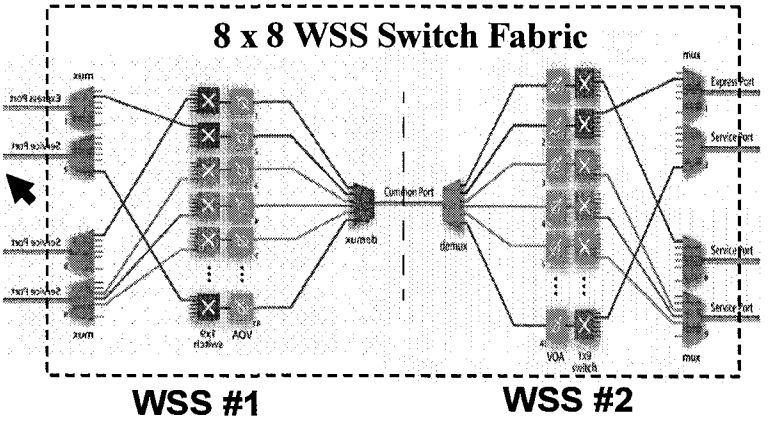


Figure 33. Optical Crossbar Using Wavelength Selectable Switch Technology

This technology of wavelength selectable switches in the form of an optical crossbar can also be extended in use within a star network where an edge node within the network can be linked to

another edge node through a core switching fabric. In this type of architecture, the light paths can be quite long between edge node and core node. Each light path now is more sensitive to the performance impairments of the core node since this will need to be considered on top of the already negative effects of both fiber dispersion and fiber non-linearities. Link budget designers will strongly rely on these parameters to optimize the transmission distance between nodes; specifically for asymmetric networks. These impairments and the methodology used to measure these will be described. Additional information surrounding the measurement procedures for some of the experiments and also the effects on signal quality can be found in both Reference 2 and Reference 3.

Insertion loss:

The insertion loss is defined as the ratio of output power to input power through a device in decibels (dB). The equation used for the insertion loss is Pout(λ) - Pin(λ). The value is typically negative indicating a reduction in power through the device. Figure 34 shows the test setup used to measure the insertion loss of the wavelength selectable switch in both configurations. In the case of the module configuration, the signal is injected into the common port and the output is extracted through each 9 output port. In the case of the cascaded pair, the input signal is injected into one of the 9 output ports of the first WSS, and then extracted out of one of the second WSS 9 output ports. In this scenario, there is a 9 x 9 insertion loss matrix that is obtained. The broadband amplified spontaneous emission source is used in place of a tunable light source to avoid any coherent effects with fiber connectors. Since a broadband light source is used, the profile on a spectrum analyzer is obtained for both the input and output. Insertion loss is a key parameter in the cascaded pair configuration since this will limit the total transmission distance allowed between edge nodes without amplification. The dynamic range at the receiver at the edge node would have significantly low power to properly provide a desired low bit-error rate of 10^{-15}. Since the receiver has a specified dynamic range to guarantee a low BER of 10-15, the measured insertion loss of the cascaded pair should allow the received optical power to be bound within this range.

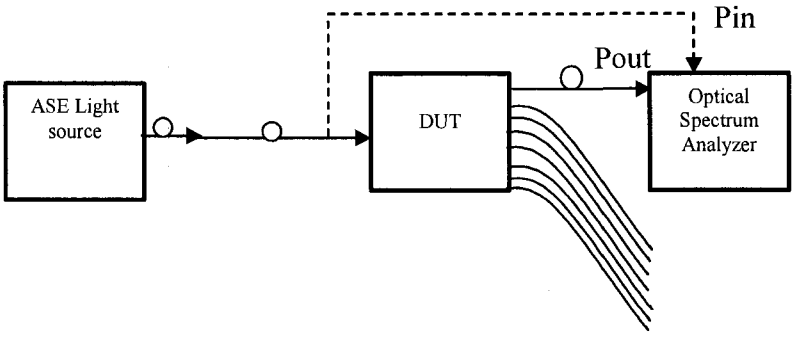


Figure 34. Insertion Loss Measurement Set-Up

Wavelength and Port-to-Port Isolation:

The wavelength isolation is defined as the ratio of the worst-case transmission power between the unwanted wavelength at a particular port relative to the desired (reference) wavelength, expressed in decibels given by $10*\log10(P_{unwanted}(\lambda n) / P_{wanted}(\lambda ref))$. The value is typically negative indicating that there should be very little presence, if not any of the unwanted wavelength. The measurement provides an understanding of the modules ability to block the port from any other wavelength except the one destined for that port. The module has the capability to allow only specific wavelengths to transmit and block the remaining. Figure 35 shows the test setup used to measure the insertion loss of the wavelength selectable switch in both configurations.

The port-to-port isolation is defined as the ratio of the transmission power of a particular wavelength at the output of a desired port to the same wavelength at the output of the adjacent ports expressed as $10*\log10(P'(\lambda n) / P(\lambda n))$ when the device is configured to transmit all wavelengths to the desired port and block the wavelengths to the adjacent ports. $P'(\lambda 1)$ represents the power at the output of any adjacent port and $P(\lambda 1)$ represents the power at the output of the desired port. Figure 36 shows the test setup used to measure the insertion loss of the wavelength selectable switch in both configurations.

For the case of the optical crossbar switch fabric used for the interconnection between the sectors for the packet switch; that is, for the case the WSS is used as a cascaded pair, poor isolation would result in multiple wavelengths being incident at a receiver of the sector. This would lead to signal distortion resulting from intersymbol interference in the desired traffic destined for that output port

due to the incoherent mixing that would occur at the receiver.

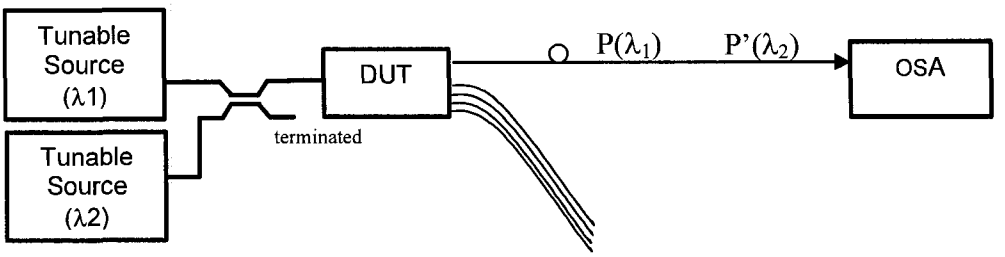


Figure 35. Port Wavelength Isolation Measurement Set-Up

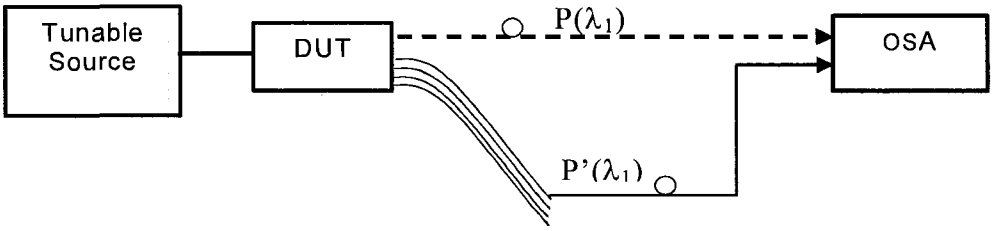


Figure 36. Port-to-Port Isolation Measurement Set-Up

Polarization Dependent Loss:

The polarization dependent loss is generally defined as the absolute or relative difference, expressed in decibels (dB), of the maximum and minimum transmission loss given all possible input polarization states at the output ports of the module. The components used inside the WSS module can potentially induce birefringence, that is, there is a change in the refractive index along the optical path that causes the light to experience different propagation constants for the two orthogonally polarized electric fields. High polarization dependent loss (PDL) will cause power fluctuations over time causing bit-errors if the received power approaches the sensitivity of the receiver. The rate of these fluctuations are typically on a slow time scale as they are correlated to ambient changes and/or stress along the optical path. In the case of the packet switch, there can be a fair amount of tolerance to this impairment if and only if the aggregate insertion loss of the cascaded pair is low enough to allow sufficient margin to accommodate the PDL and still be within the sensitivity of the receiver. Figure 37 depicts the polarization scanning method test set-up used to measure the PDL of the wavelength selectable switch in both configurations. The laser in the

setup is used to tune to the desired wavelength for the WSS and the polarization scrambler consisting of fiber loops are rotated arbitrarily such that they rotate the states of polarization completely around a Poincaré sphere, and the change of power is detected at a photodetector. The instrument photodetector consists of a fast acquisition system to be able to capture sudden changes in polarization and register this for both the maximum and minimum transmission power.

Since the components in each of the modules are not theoretically identical, the optical path state of polarization can change significantly between the individual module and the cascaded pair. For this reason the pair was measured.

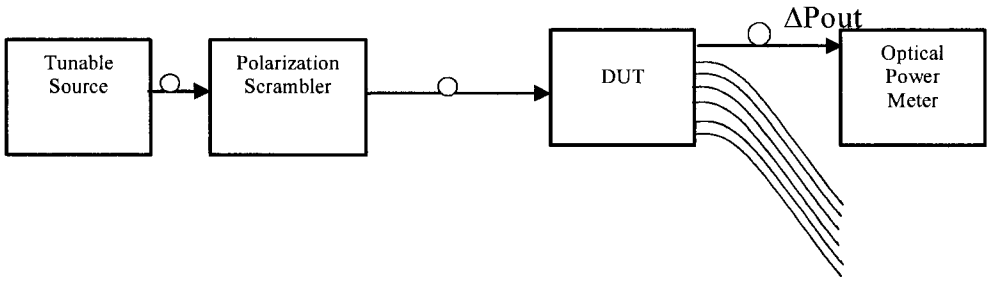


Figure 37. PDL Measurement Set-Up

Polarization Mode Dispersion:

Polarization Mode Dispersion is a very detailed topic in the field of high-speed optical communication systems which requires significant knowledge of stochastic processes and Maxwellian theory. For this reason, only high-level discussions will be provided for the purpose of briefly detailing the impairment and it's impact on the packet switch. Polarization mode dispersion; otherwise known as the differential group delay (DGD), is the time difference between the principal states of polarization for a given wavelength and a time. As a result, the data in time can experience a broadening by the amount of the PMD, if the pulse is not transmitted exactly along the same polarization states as the optical pathway created between the sector pairs for the packet switch. The main reason for this variation comes from the fact that the WSS contains multiple optical components each having different birefringence, so when an optical pathway is created, there is a small probability that all the components when combined will produce the same polarization state to the data signal that is launched. The time-average value is typically expressed

for the DGD. The mathematical modeling of the PMD for concatenation of components can be taken as the sum-squared of the individual PMD [Reference 4].

The measurement platform is based on the Jones-Matrix Eigenanalysis Method by using a swept-wavelength interferometer system. Figure 38 depicts the optical measurement system design developed by Luna Technologies. The method using Jones matrix is based on applying 3 known linear states of polarization at the input to the device using one polarization controller. Typically 0, 45 and 90 degrees are used; however, in this system there are two polarization controllers P1 and P2 used. [Reference 5] shows both P1 and P2 within the test optical diagram of spliced test station. Each of these controllers are optimized to allow orthogonal polarization at both detectors s & p. The intensities for each detector is then obtained for each wavelength and since the paths produce different phase delays, which can be visualized and captured by shifted impulse responses with delays of $\tau_d + \tau$ and $\tau_d - \tau$, a fourier transform is taken and the transfer function for the device under test is then calculated. Since the measurement platform can produce significant noise in the measurement when determining the impulse response, there are several settings that were required to maintain optimal PMD accuracy. The calibration procedure is taken over 44 nm beginning with 1529 nm to 1572 nm and averaged 4 times. The PMD measurement is then averaged 64 times and the time domain window resolution bandwidth is manually adjusted to reduce unwanted noise from the DGD.

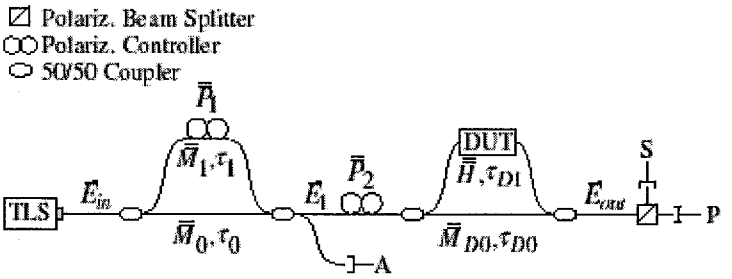


Figure 38. Luna PMD Measurement System

Switching Time (Reconfiguration Time):

The switching time is defined as the total time between the moment the transmission signal exiting from an output port disappears to the moment the transmission signal appears at any of the other

output ports as configured. The switching time for the WSS is an important parameter that can be viewed as an impairment for the packet switch as this contributes directly to the scheduling algorithm delay. A large delay would require that the queues at the input sector be sized accordingly to store the packets received during that period. Figure 39 depicts the test setup required to measure the switching time. The photodetectors are placed at any two output ports for the WSS with a broadband source connected directly to the input. The broadband source is used to simulate the entire wavelength band switching over from one port to another. The photodetectors have a wide bandwidth (~6 GHz) and can easily capture very fast transitions from one port to another on the wide bandwidth (500 MHz) oscilloscope. For this measurement only the modules themselves were measured and not the cascaded pair since the cascaded pair will be configured simultaneously by the global controller within the packet switch architecture. The total time is then taken as the worst-case time between the two modules measured.

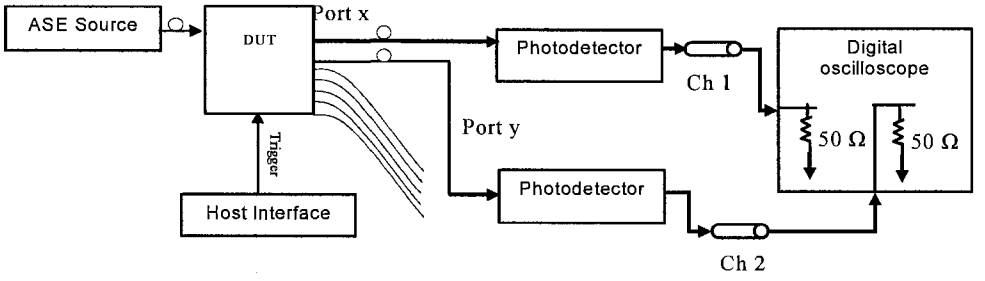


Figure 39. Switching Time Measurement Set-Up

Total Return Loss:

The total return loss is defined as the amount of power that is reflected from a particular port from the device under test expressed in dB. The value is typically negative indicating lower power than what is initially transmitted through a particular port. Figure 40 depicts the test station used to measure the return loss. A broadband source is used to capture the effects of all wavelengths from the WSS for each port. A circulator is used to prevent any light from going back into the source and guide the signal in a clockwise direction to the port where the optical power meter is connected. Due to the high isolation from the circulator and high resolution on the power meter, a return loss limit of approximately -45 dB can be achieved. The total return loss is a viable impairment for the

switching fabric since any fractional power being reflected at a particular interface can in fact be returned to the transmitter which had originated the transmitted signal. If the transmitter is not properly optically isolated, there can be adverse effects resulting in a reduction in transmit power; similarly on the receiver, there would be a decrease in the optical power detected which would ultimately lead to bit-error penalties. For the case of the WSS, each of the unused ports are properly terminated whilst the reference signal is transmitted to a desired port in order to remove any open connector cases which will result in a false high return loss value.

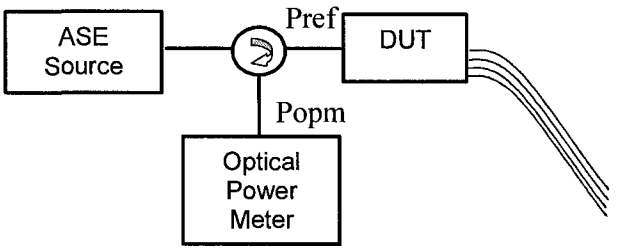


Figure 40. Return Loss Measurement Set-Up

Interferometric (Inband) Crosstalk

Interferometric, or in-band crosstalk is defined as a distortion of a transmitted signal when replicas of the signal are detected at a receiver which originate from a time-delayed version of that signal from the same source or other signals from other sources within the device imposed onto the same receiver within its detection bandwidth. There has been multiple areas of research on this topic within both device level and system level architectures. [Reference 8] and [Reference 9] both provide studies of the effects of both crosstalk effects from multi-point reflections and demonstrate the effects on the bit-error rate on a signal at 2.5Gb/s. Reported in these papers, a 2 dB receiver penalty is seen on bit-error rate (using 1x10^-9 as a reference) with just under 20 dB crosstalk. The essence of the crosstalk is the measure of the amount of intensity noise produced by the phase noise due to incoherent or coherent homodyne effects at the receiver. Therefore, the measurement technique quantifies the amount of crosstalk through an equivalence to a self-homodyne technique and integrates the total Lorentzian spectral density on a RF spectrum analyzer to derive the total amount of beat-noise power that will be used to calculate the cross-talk ratio in decibels (dB). The derivation of the total linear beat-noise power is provided in Appendix B and is given by

$P_{noise} = K_{cal} \cdot P^2_a \cdot \gamma$., where Pa is the average input power, g is the level of crosstalk and Kcal is a constant that identifies the responsivity of the entire detection path (photonic and electronic).

Figure 41 depicts the initial setup used to measure the calibration constant K_{cal}, and Figure 42 depicts the diagram used to measure the crosstalk for the wavelength selectable switch. Figure 41 includes a variable optical attenuator (VOA) in the mid-stage in order to control the desired amount of crosstalk. Integrating the power spectral density, fixing the cross-talk value through the attenuator setpoint, and measuring the average optical power incident to the photodetector, yields the calibration constant K_{cal}. The detector has a bandwidth of < 2.5 GHz, but since the laser only has a linewidth of 30 MHz, the spectrum in integrated over a much smaller range to capture a large percentage of the power, as it is known that the Lorentzian functional tale drops off in amplitude quickly at higher frequencies and would not contribute significantly to the total power. The integral can be taken by using the rectangular rule and Reimann sums. The length of fiber (L1) is such that it exceeds the coherence length of the laser; a value of 20 m is chosen. The tunable filter within the test setup removes the effects of total broadband noise from the EDFA and only keeps the amplified signal wavelength of interest. The wavelength used within the test setup is 1552.52 nm.

Figure 42 shows the use of two variable back-reflectors to control the amount of back-scatter light into the WSS. The calibration constant is verified in order to achieve a cross-talk level that is equivalent to the sum of both back-reflectors when no WSS is present. The back reflectors are adjusted to their maximum setting in order to ensure there is no round-trip path that can occur and only the light that is internally reflected within the WSS exits the output path towards the photodetector. The length of fiber for L1 and L2 are identical and are both 20 m.

Unfortunately due to the high loss of the device, and the limited sensitivity of the RF analyzer, crosstalk levels below 60 dB are not measurable with the setup. For this reason, the pair configuration can not be measured for this test, and only the individual modules are measured. Incidentally, a worst-case approximation can be taken by assuming that the crosstalk for each WSS are summed coherently in order to provide an estimate of the aggregate crosstalk for the pair.

When executing the measurement, the WSS output ports are terminated to completely remove the effects of any potential back-reflection from other ports.

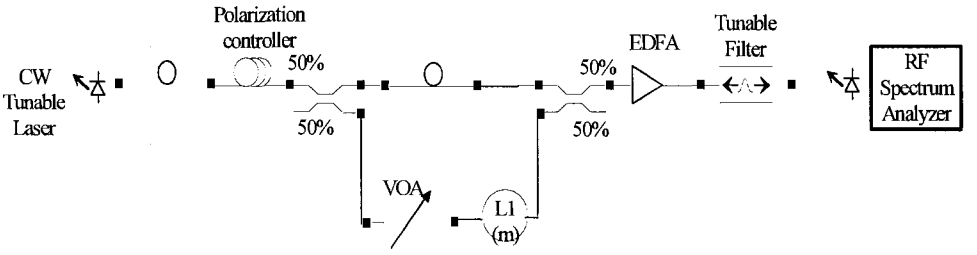
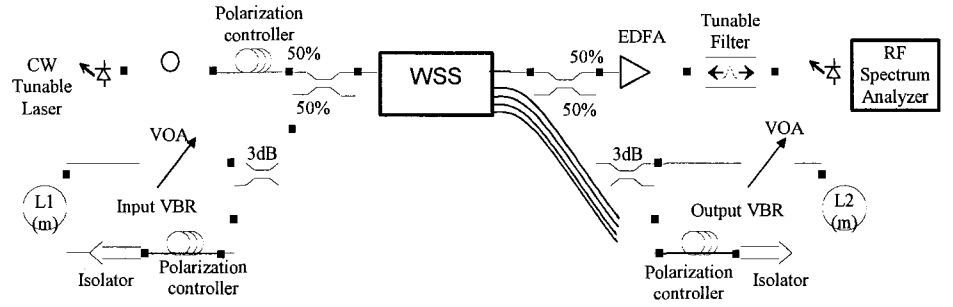


Figure 41. Cross-talk K-Calibration Measurement Set-Up



Note that the length of fiber within the VBR depends on the coherence length of the source
The photodetector is a receiver that contains a high transimpedance gain to boost the signal

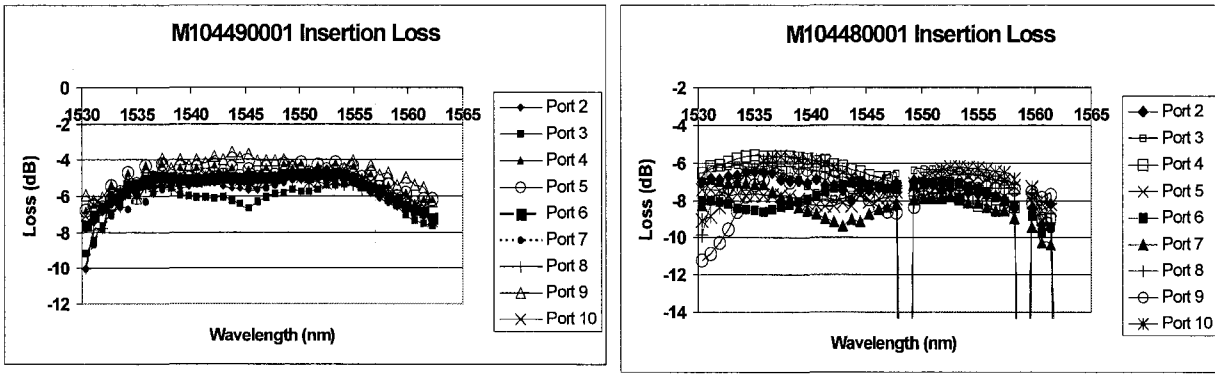
Figure 42. Interferometric Cross-talk Measurement Set-Up

4.2 Experimental Results

The results obtained from testing were focused on both individual wavelength-selectable switch modules and the cascaded pair. In some cases the cascaded pair was not measured either due to the limitations (sensitivity) of the instruments or because the measurement was obtained by the individual modules. All measurements were taken across 41 wavelengths in the C-band (1530 - 1565 nm) which are identified on the ITU grid (Reference ITU-T G.694.1 or Bellcore GR-2918-CORE). Module #1 is referred to as serial number M1044800001 and module #2 is referred to as M1044900001. The list of equipment used for the measurement setups can be found in Appendix A. There is a graphical user interface provided by Metconnex that allows RS-232 commands to a test jig interface that will configure ports and perform wavelength selection for the module.

Insertion Loss:

Figure 43 depicts the insertion loss of both of the individual WSS modules between the common port and 9 of the output ports. Figure 43 (b) shows that there are some wavelengths with excessive loss. This excessive loss is a result of a failed variable optical attenuator component in the optical path of the following wavelengths located after the switching element: 1548.51 nm, 1558.98 nm, and 1562.23 nm. From the graph, the minimum and maximum insertion loss excluding the failed wavelengths for module #1 and module #2 is -11.25 dB and -5.63 dB, respectively. This boundary sets the worst case peak-to-peak wavelength dependence to 4.98 dB. Figure 44 depicts the distribution of the wavelength dependence and the aggregate insertion loss for all port combinations for the cascaded pair. The peak of the distribution is centered between -12.0 to -13.0 dB with a wavelength dependence distribution peak centered between 5.2 to 5.6 dB.



(a) (b)

Figure 43. Measured Insertion Loss for WSS modules across wavelength (a) S/N: M104490001, (b) S/N M1044800001

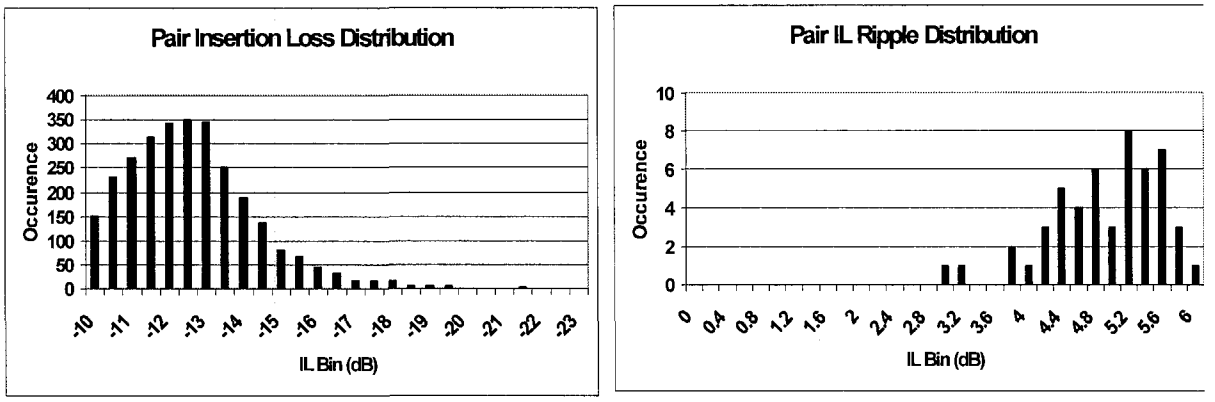


Figure 44. Measured Distribution of the WSS Cascaded Pair

Wavelength and Port-to-Port Isolation:

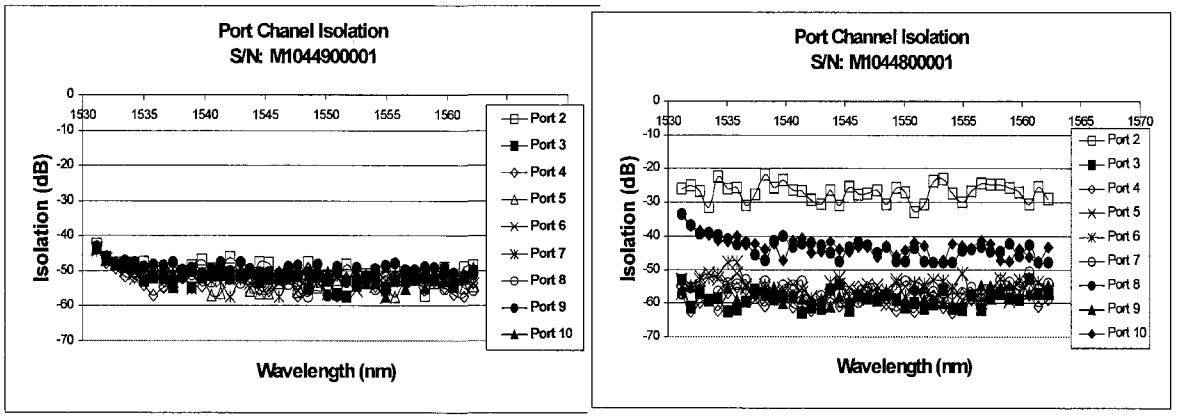
Figure 45 depicts the wavelength isolation of the WSS modules for each output port. A reference wavelength of 1530.33 nm was selected while the other wavelengths were blocked. The sensitivity of the optical spectrum analyzer was set to -70 dBm, in order to measure very low isolation. The worst-case wavelength isolation across all the ports is < -43.0 dB for module #2, and < -22.0 dB for module #1. Module #1 shows that the worst case isolation occurs for Port 2; however, all other ports are very comparable to module #2. The plots of the isolation for module #1 includes the wavelengths that experience high insertion loss; that is, 1548.51 nm, 1558.98 nm, and 1562.23 nm since the isolation for these channels would be high and would not contribute to creating the worst case isolation for the module as these channels power level would fall below the noise floor of the spectrum analyzer.

Figure 46 depicts the port-to-port isolation of the WSS modules. The worst-case port-to-port isolation for module#1 is given as -30.7 dB and -35 dB for module#2. The distribution of isolation for all combinations of measured ports show isolation values better than -50 dB. The sensitivity limit for the measurement was < -65 dB, since the optical spectrum analyzer noise floor was set to -70 dBm.

For these parameters, the cascaded pair was not measured since the measured insertion loss of the cascade would result in transmit powers that would be below the noise floor of the optical spectrum

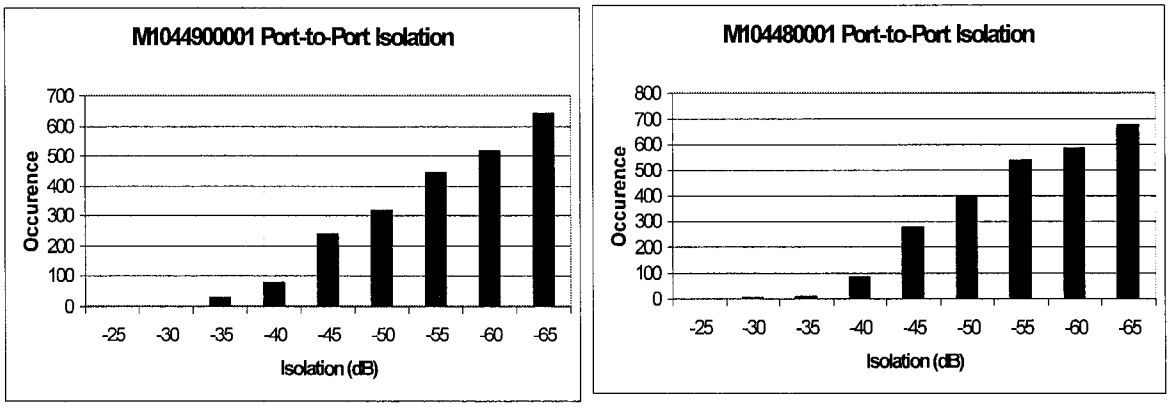
analyzer. For the cascaded pair, the isolation can be summed together since all measurements are relative between ports or at a given port in relation to the common input port. Conceptually if there was a different optical pathway not through the common input port that would be generated in the cascaded configuration, then the isolation would not sum linearly. Fortunately, this is not the case for the central stage configuration.

A measure of the worst-case port-to-port isolation out of a given port can be assumed when the worst-case isolation occurs for every channel to each of the output ports. In "the hardware demonstrator" each port of the central stage switch fabric (cascaded pair) will be connected to different wavelengths. The isolation between sectors pairs with this configuration is given by: (Nmod1 x ILmod1 + Nmod2 x ILmod2); where N is the number of wavelengths, which will be equal to the total number of ports from the WSS, and IL is the isolation expressed in linear units for a particular module. The calculation is then re-expressed back into decibels. For this configuration setting, the port-to-port isolation is 10 x log10(9 x 10^(0.1 * -30.7) + 9 x 10^(0.1 * -35)) = -19.80 dB.



(a) (b)

Figure 45. Measured Wavelength Isolation for WSS modules across wavelength (a) S/N: M104490001, (b) S/N M1044800001

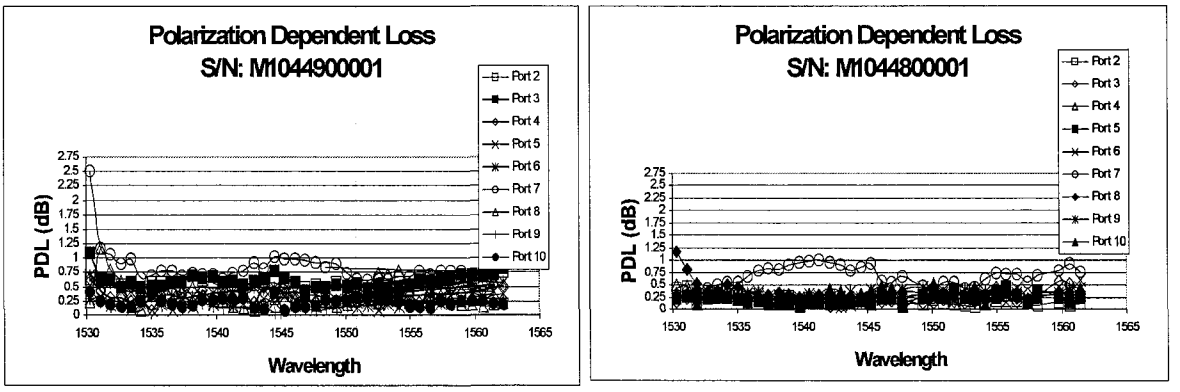


(a) (b)

Figure 46. Measured Port-to-Port Isolation For WSS Modules
(a) S/N: M104490001, (b) S/N M1044800001

Polarization Dependent Loss:

Figure 47 depicts the polarization dependent loss of the WSS modules for each output port. The data for module #1 does not include those wavelengths that experience high insertion loss; that is, 1548.51 nm, 1558.98 nm, and 1562.23 nm. The PDL of module #1 ranges from a minimum value of 0.1 dB to a maximum of 2.5 dB and the PDL of module #2 ranges from a minimum value of 0.1 dB to a maximum of 1.2 dB. Figure 48 depicts the polarization dependent loss of the cascaded pair. The distribution, excluding those wavelengths with excessive insertion loss is shown for all input to output port combinations for the 9 x 9 cross-connect. The mean of the distribution is centered towards 0.6 dB. Fortunately, the receiver sensitivity that will be used for the sectors is high and the input power to the receiver will be within the dynamic range; therefore, the significance of this high PDL would not impose significant penalties on the data traversing the paths between them.



(a) (b)

Figure 47. Measured PDL for WSS modules across wavelength (a) S/N: M104490001, (b) S/N M1044800001

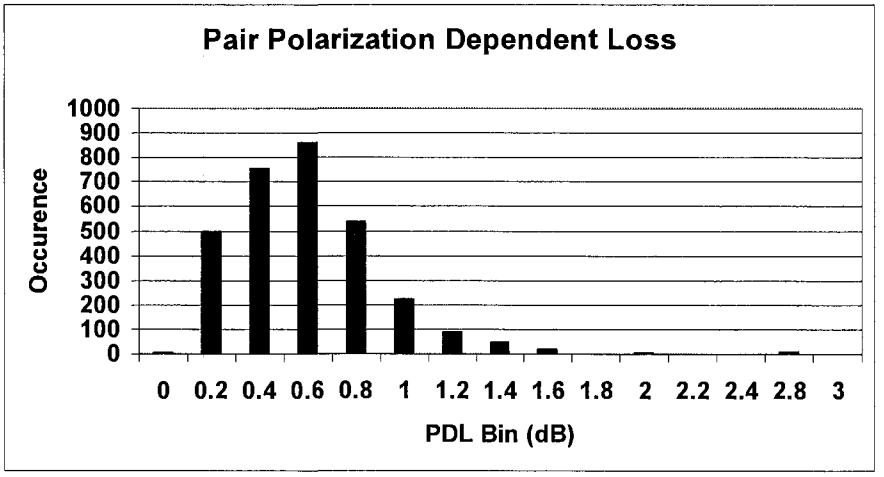
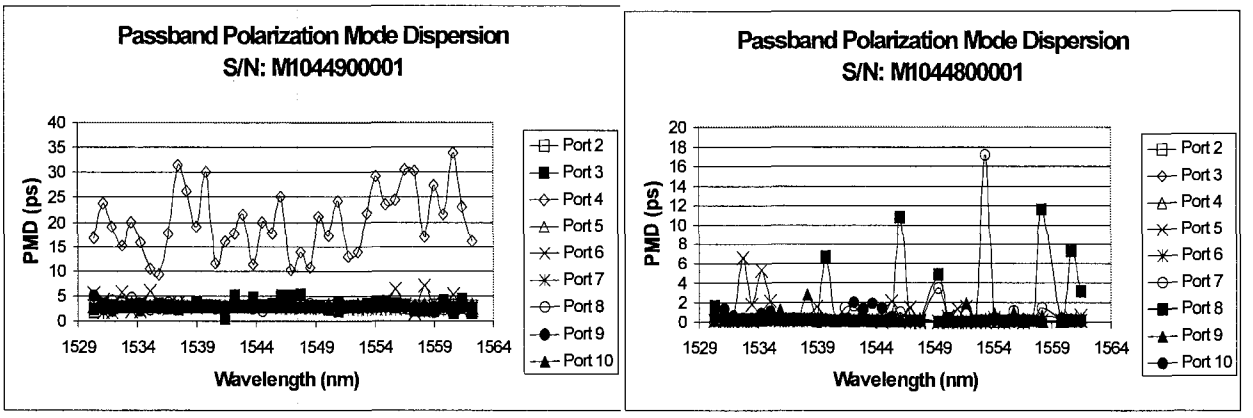


Figure 48. Measured PDL Isolation For the Cascaded Pair

Polarization Mode Dispersion:

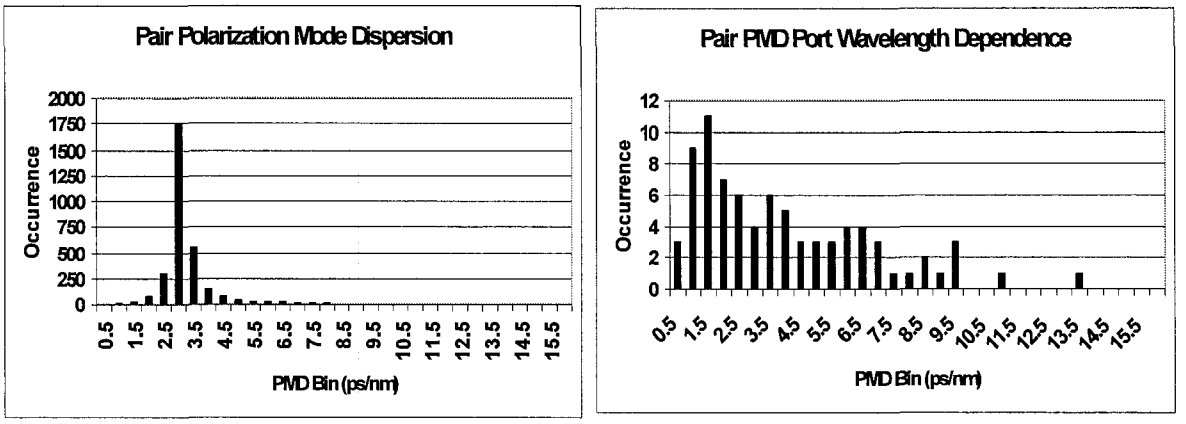
Figure 49 depicts the polarization mode dispersion of the WSS modules for each output port. The data for module #1 does not show the data for those wavelengths that experience high insertion loss; that is, 1548.51 nm, 1558.98 nm, and 1562.23 nm. The PMD of module #1 ranges from a minimum value of 0.1 ps to a maximum of 17.2 ps and the PMD of module #2 ranges from a minimum value of 0.5 ps to a maximum of 33.8 ps. Figure 50 depicts the polarization mode

dispersion distribution of the cascaded pair for all port combinations. Figure 50 (a) demonstrates the mean of the distribution is centered at 3.0 ps; whereas, Figure 50 (b) details a resolution of the distribution across wavelength for each input to output port combination. For this particular figure, the worst-case deviation across wavelength for each optical pathway (input port to output port) was calculated. The plot shows that for a given optical pathway there are cases where depending on the wavelength that is transmitted, a PMD beyond 8.5 ps can exist. Since the transmission line rate between sectors using optical transceivers operating at a rate of 2.5Gbps (bit period = 400 ps), the worst-case value PMD would not impose any significant impairment in the overall bit-error rate.



(a) (b)

Figure 49. Measured PMD for WSS modules across wavelength
(a) S/N: M104490001, (b) S/N M1044800001

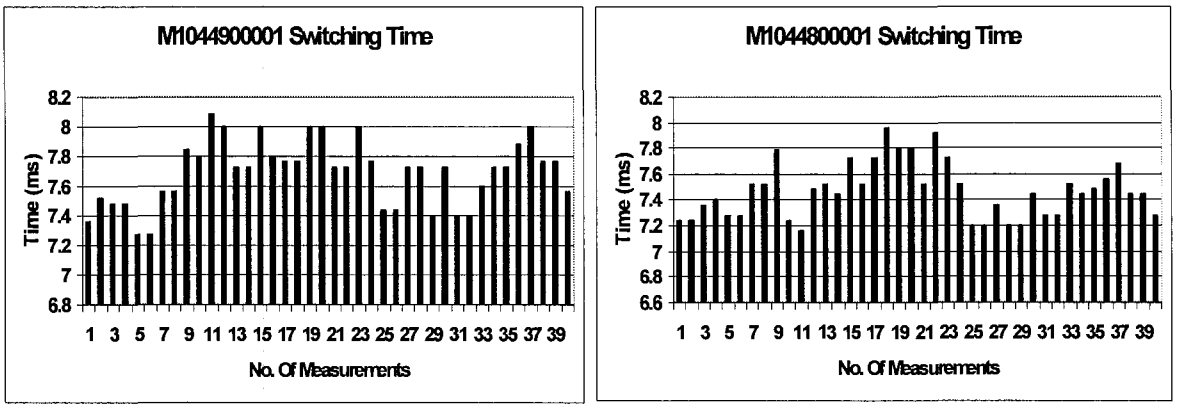


(a) (b)

Figure 50. Measured Pair PMD
(a) absolute PMD across all wavelengths and ports, (b) PMD deviation across wavelength for each port

Switching Time (Reconfiguration Time):

Figure 51 depicts a histogram of all the port combinations for switching time measured for each WSS module. The worst case switching time measured from both modules is 8.08 ms. The time taken to switch between ports varies depending on the optical technology used for the WSS module. Different technologies will produce different switching times. The WSS technology developed by Metconnex contains micro-electrical mechanical systems (M.E.M.S) and generally have longer times compared to other active type technologies, such as sold-state devices. A company called Civcom and Nozomi Photonics creates a mixed technology switch consisting of liquid crystal/free-space materials and PLZT technology, respectively to produce switching time scales of 400 and 5 nanoseconds, respectively [Reference 6, Reference 7]. Unfortunately due to budget constraints, these switches were not obtained while the thesis was being written. However from the specification, these types of switches are already several orders of magnitude faster than MEMS.



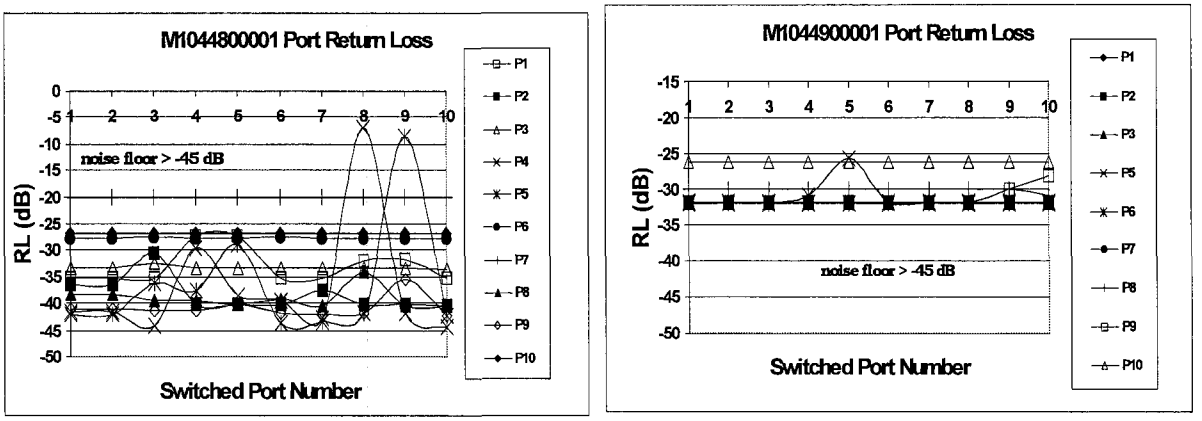
(a) (b)

Figure 51. Measured Switching For WSS Modules
(a) S/N: M104490001, (b) S/N M1044800001

Total Return Loss:

Figure 52 depicts the measured total return loss for the individual modules. The second configuration - the cascaded pair was not measured since the ports would be identical to the individual modules and thus would be a redundant measurement. The legend for the plots lists the ports which were connected to the circulator reference path; that is the ports that were being measured for return loss; whereas, the x-axis on the plots shows how the return loss varies when the WSS was physically changed to another output port without disconnecting the reference signal from the test station. The plots show that when the WSS is configured at an output port being measured (Port 2 from the legend, and Port 2 on the x-axis for example), the total return loss. is generally < -24 dB except for the case of one of the modules which experienced a return loss of -20 dB. When the WSS is configured away from a specific port (Port 2 from the legend, and Port 3 or Port 4 etc...on the x-axis for example), the WSS shows significant return loss dependencies. Module #1 return loss was as high as - 7 dB for the case when Port 8 and Port 9 was selected respectively for the measurement of Port 4 and Port 5. What this could possibly highlight is that anything connected to the other output ports while a specific port is being accessed; that is, when an optical pathway at a specific wavelength is being constructed, the other ports will experience high reflectivities and could cause penalties such as overloading a receiver, bit-error penalties

and/or cross-talk penalties.



(a) (b)

Figure 52. Measured Return Loss For WSS Modules
(a) S/N: M104480001, (b) S/N M1044900001

Interferometric (Inband) Crosstalk:

Figure 53 depicts the crosstalk levels for both module #1 and module #2 for each output port. The inband crosstalk for the module shows a maximum at -49 dB and a minimum of -55 dB for module #1, while module #2 had a maximum at -55 dB, and a minimum at -57 dB. A K_{cal} value of 1650 was measured and tuned for the test station. The K_{cal} factor was verified for repeatability over days and showed a variation of no more than +/- 1.0 dB. The crosstalk value was measured while the back reflectors were configured to a value of -27 dB. The fact that the level of crosstalk is low means that there will not be any bit-error penalties attributed from the modules when a light path is constructed when interconnected in a pair configuration.

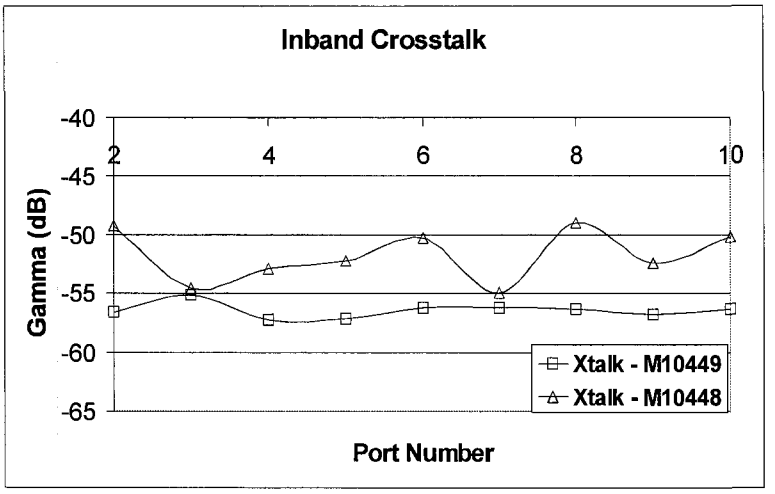


Figure 53. Measured Inband Crosstalk for both WSS modules. The crosstalk is measured between the common port and an output port

Chapter 5 Conclusion and Future Work

In the previous chapter a detailed view of the optical impairments for the central stage crossbar was presented. This chapter summarizes the research and findings and also provides areas for future work.

5.1 Review of the Research

A System-on-Chip linked-list memory manager and optical impairments of a wavelength-selectable switch in a cross-connect configuration have been described in the context of the implementation of a three-stage Clos-like sectored packet switch with a reconfigurable optical central stage.

Two of the main parameters essential for the design of the FBP sectored packet switch with an optical core are the amount of time taken to write and read packets to and from a virtual output sector queue, and the second is the reconfiguration time needed for the central photonic crossbar.

Within the three-stage sectored packet switch, a measure of the time taken to write and read packets to and from memory including the time required to update all necessary pointers was obtained as 880 ns and 1040 ns, respectively. If we consider the *'hardware demonstrator'* with an input sector each with a port linerate of 10 Mbit/sec (156.25 Kpackets/sec), then each port would want to write to the shared-memory every 6.4 μs. There is clearly a long enough margin to complete a write to a VOSQ in memory and allow for an internal speed-up for the read if need be. This speed up can be calculated as an internal speed-up for reading a VOSQ. A computed speed-up value of approximately 5 times is allowed (i.e. (6400 - 880) / 1040 ns). However, if we consider servicing all 4 ports (all ports are a high-traffic load) in a round-robin fashion, then there is insufficient time before the next packet arrives to be able read-out all the incoming packets stored in the VOSQ. Using the same simulation values, only 2 packets (i.e [6400 - (4 * 880)] / 1040 ns) would be read from the VOSQs. This implies that a faster access time is needed to avoid gradually filling the memory to a point where it becomes full and packet loss occurs.

A measure of the amount of time taken to reconfigure the optical cross-connect was obtained for

all desired port configurations with a worst-case value of 8.08 ms. This is the out-of-service time of the central stage and represents the minimum time between successive reconfigurations. The out-of-service time must be at least twice the measured value of 8.08ms; otherwise, a large temporal speed-up (i.e. increased bit rates in the central stage) would be needed to compensate for the out-of-service time to obtain 100% throughput for the packet switch. For the *'hardware demonstrator'*, a timeslot corresponds to the packet size, in bits, divided by the line rate in bits per second. This corresponds to a value of 64 bits / 10Mbps which is equal to 6.4 μs. In the worst case, 1263 packets (i.e. 8.08 ms / 6.4 μs) can arrive for one of the ports of the packet switch destined to one of the VOSQs. If there are 4 input ports for a given sector, then up to 5052 packets can be stored in shared-memory (SDRAM). Since the SDRAM has available 32768 (2^{15}) address locations for storing packets, the demonstrator will be able to support the MEMs technology for the central stage (assuming that the scheduling algorithm keeps the queues fairly short during servicing time). If the line rate is scaled up by more than 6.49 times (32768 / 5052), then the packet switch would run out of memory space and would start dropping packets. Increasing the line rate therefore would require more memory in order to reduce the number of packets dropped. Surely a different technology can also be used to support the increased line rate. A faster photonic technology such as liquid crystal for the central stage can be used instead to reduce the out-of-service time; however, the drawback in doing so is scalability. Faster switches typically are in the form of 2 x 2, or 4 x 4 configurations and to scale these to larger dimensions means architectural complexities.

The linked-list memory manager is an integral part of a shared-memory switch that constitutes the input sector of the packet-switch. One of the main goals for using linked-lists for the various queues is to make a more efficient use of the memory space by minimizing the unused memory locations when the queues are very different in length from one another. Another primary function as designed and described in the thesis is to allow packets to be stored into SDRAM while the pointers to track the next packet within the same list (VOSQ) are stored into SRAM. This division of pointers and packets into separate memories allows the more dense and slower memory (SDRAM) to contain the bulk of the data which does not need any further processing while the less dense and faster memory (SRAM) manages the pointers that is constantly changing. The linked-list memory manager was designed using a System-on-Chip (SoC) approach where each of the customized peripherals, implemented using a Mealy state machine, are tasked to perform specific

functions related to pointer management with SRAM, accessing SDRAM for data storage/retrieval and monitoring input FIFO buffers all within an Altera^{TM} Stratix FPGA. Since there can be many VOSQs (linked-list) in an input sector; that is, instantiations of identical customized peripherals (masters), this SoC approach in the FPGA is ideal because each peripheral, including the controllers used for the off-chip memory, is connected to a common bus shared with all the peripherals. A unique approach in this design is that the peripherals do not share one main bus arbitrator for all master peripherals (VOSQ linked-list); rather, the FPGA Avalon Bus architecture allows one arbitrator to be instantiated for each slave peripheral. This allows one master peripheral to be free to communicate with other slave peripherals when other master peripherals are accessing the bus and are each accessing different slave peripherals. The more standard bus structure prevents any one specific master in communicating with any slave peripheral and forces the entire bus to wait while that specific master who made the request completes its operation which is not the case for the slave arbitration used in this FPGA. Although not shown directly in this thesis, this could occur in the input sector of the packet switch when multiple VOSQs (linked-list) peripherals are instantiated and one of them performs an update to a pointer by accessing SRAM via the controller peripheral and another reads or writes packet data by accessing the SDRAM via the controller peripheral. The linked-list also reports its length (VOSQ queue length); a parameter that is sent as one possible type of traffic matrix required for the FBP algorithm.

The architecture of the packet switch also allows for the use of reconfigurable optics to be used in the central stage so that the sectors may be either close together or far from each other. One of the types of photonic technologies that was examined in this thesis for possible implementation of the optical core is a cross-connect built with two wavelength selectable switches in a back-to-back configuration. When the distance, line-rate and complexity between the sectors grows so does the importance of the optical impairments for the central stage. The impairments that are of large concern are the insertion loss, wavelength isolation, interferometric (inband) crosstalk, polarization mode dispersion, polarization dependent loss, and finally return loss. These parameters can: (1) limit the distance between sectors based on received optical power, (2) limit the transmitted signal quality by increasing the bit-error rate and (3) limit the types of architectures that can be used to construct large photonic non-blocking cross-connect switches.

This thesis has reported the measured values for each impairment of a MEMs based cross-connect consisting of two wavelength-selectable switches. These measurements are required for further future studies on the effects of transmission. The typical measured values for insertion loss, polarization dependent loss, and the worst-case values for polarization mode dispersion and in-band crosstalk were -13 dB, 0.6 dB, 8.5 ps, and -49 dB, respectively. If we assume transmission on standard non-dispersion shifted fiber (NDSF) where the fiber attenuation is approximately 0.25 dB / km, the net chromatic dispersion is 17 ps / nm / km. Moreover, if we use a distributed feedback laser (DFB) with a transmit power of 0 dBm and optical spectral width of 0.3 nm, and an optical receiver using an avalanche photodiode (APD) with a sensitivity of -28 dBm, then an input-to-output sector distance can be computed, assuming, that we neglect the use of optical amplifiers or regenerators. The total allowed span loss without distortion penalties is 28 dB (i.e., 0 dBm - (-28 dBm)) which translates to 112 km (i.e. 28 dB / 0.25 dB/km). By using the wavelength-selectable switch cross-connect, and only considering the insertion loss, the sector-to-sector distance allowed is 60 km (i.e. 112 km - 52 km). This bit-error rate typically decreases when the temporal effects such as fiber chromatic dispersion and crosstalk are taken into consideration.

In the case of crosstalk, this also imposes constraints on the dimensions of the central stage and what types of architectures can be used. The effect of crosstalk in this 8 x 8 MEMs based configuration can be neglected since its value is very small. For larger packet switches where crossbars can approach dimensions of 32 x 32 (i.e. a 256 x 256 packet switch with 16 sectors), the crosstalk will become more significant. The crosstalk dependencies in terms of switch dimension are discussed in [Reference 13]. It should be noted that there are no commercially available MEMs based switch today for a central stage of that dimension. The large dimensions are strictly available by building them using smaller port count switches.

The total pulse spreading at the computed distance of 60 km including both chromatic dispersion and polarization mode dispersion would then be 0.315 ns. Since the *'hardware demonstrator'* has a bit period equal to 100 ns (i.e., 1 / 10 Mbps), this represents 0.314% of the bit period. This percentage increases when the line rate increases. Using the bandwidth-product equation from Reference 3, a maximum value of 49 Gbps*km can be computed. When considering a distance of

60 km, this translates to a maximum bandwidth 816.7 Mbps without incurring significant bit-error penalties due to dispersion.

Other issues that are of concern as the distance grows between the input and output sectors are the transmission of control signals and synchronization of the sectors with the optical switch. Proper timing is required so as to avoid sending packets to the wrong destinations or losing them altogether. The Agile-All Photonic Network can be viewed as a distributed version of the FBP packet switch architecture; therefore, understanding these concepts are essential for optimizing the throughput.

5.2 Topics for Future Work

There are several areas that are essential for completing the development of the *'hardware demonstrator'* of the sectored packet switch with a reconfigurable central stage:

- Perform a rigorous timing analysis including setting setup and hold timing constraints on all outgoing signals to the external memory peripherals. The purpose would be to identify whether there are any propagation delays that impose issues for the specific clocking frequency used in accessing the memory. This would allow the hardware verification to be simplified when integrated and synthesized into the FPGA.
- Instantiate a second Master Input Logic peripheral representing a second VOSQ and verify that a second linked-list is operational and that the bus within the FPGA performs slave peripheral arbitration under various queue loading; that is, to confirm that weighted-fair arbitration occurs between two VOSQ with the off-chip shared memory.
- Since arbitration plays an important role in terms of achieving efficient sharing of the memory between VOSQs, another task would be to study the effects of the different types of arbitration for the shared-memory and also different arbitration schemes that can be implemented while accessing both the input and output prestore and prefetch FIFOs. The thesis only assumes that while both prestore and prefetch FIFOs are not empty and full respectively, then 50 percent of the time packets are read from and written into both of the FIFO's respectively.
- Perform studies on redesigning the linked-list memory manager to accommodate different memory technologies. For instance, not using SDRAM; rather only using many fast SRAMs connected together in parallel. Here the study would be to investigate how to reduce the access times without incurring the latency of the capacitive effects and refreshing needs of the SDRAM technology.

- Investigate and introduce a fairness scheme into the memory management; that is, consider (and try to avoid) the scenario when one of the VOSQs is populated with all the packets and consumes most if not all the memory while leaving nothing for the other queues. The current design of the linked-list memory manager in this thesis does not put any thresholds on the amount memory allowed for each of the queues. The study of queue length thresholds (static or dynamic) to achieve fairness can be adopted similar to the ones described in [Reference 25] and the criteria of the best type of memory management for this packet switch architecture can be investigated and implemented.
- Investigate the effects on the physical transmission for different configuration states of the core optical switch. As highlighted, the central switch does not have to be close to the sectors, and since the photonic paths can be created on regular intervals based on the Flexible Bandwidth Provision scheduler, the different states of the optical switch may impact the allowed distance between sectors. The optical characteristics studied and measured in the thesis will allow for the study on these effects when the entire packet switch is put together.

References

1. **"Metconnex WSS Product Brief"**, http://www.metconnex.com
2. Derickson, D. **"Fiber Optic Test and Measurement"**, Prentice Hall, 1998, ISBN 0-13-534330-5.
3. Kazovsky, L.; Benedetto, Sergio; Willner, A., **"Optical Fiber Communication Systems"**, Artech House Inc., 1996, ISBN 0-89006-756-2
4. Karlsson, M., **"Probability Density Function Of the Differential Group Delay in Optical Fiber Communication Systems"**, Journal of Lightwave Technology, Vol.19, No.3, March 2001.
5. **"Optical Vector Network Analyzer For Single Scan Measurements of Loss, Group Delay and Polarization Mode Dispersion"**, White Paper, Luna Technologies, May 2005.
6. **"Ultra-Fast Optical Switch Series UF-OS"**, Document no. DOC-01-161-00, http://www.civcom.com/admin/pdf/SysPic/OSdatasheet.pdf
7. **"AlacerSwitch 0202Q: 4-array 2x2 high-speed optical switch"**, http://www.nozomiphotonics.com/pdfs/en/datasheet_alacer_switch_0202Q.pdf
8. Gimlett, J.; Cheung, N., **"Effects of Phase-to-Intensity Noise Conversion By Multiple Reflections on Gigabit-per-Second DFB Laser Transmission Systems"**, Journal of Lightwave Technology, Vol. 7, No. 6, June 1989.
9. Gallion, P.; Debarge, G., **"Quantum Phase Noise and Field Correlation in Single Frequency Semiconductor Laser Systems"**, Journal of Quantum Electronics, Vol. QE-20, No. 4, April 1984.
10. Kazovsky, W., **"Data and Computer Communication"**, MacMillan Publishing Company., 1994, ISBN 0-02-415441-5
11. Paredes-Zamorano, S., **"Flexible Bandwidth Provision and Scheduling in a Packet Switch with An Optical Core"**, Ph.D Thesis, King's College University of London, March 2005.
12. Clos, C., **"A study of non-blocking switching networks"**, Bell Syst. Tech. Journal, pp. 406-424, 1953.
13. Shanker, R.; Hall, T., **"Core Switch Architectures For the Agile All-Photonic Network: Performance Evaluation with Crosstalk"**, IEEE Indicon 2005 Conference, Dec.2005
14. Liew, S., **"Multicast Routing in 3-Stage Clos ATM Switching Networks"**, IEEE Transactions on Communications, Vol. 42, No. 2/3/4, Feb./Mar./April 1994.
15. **"Stratix Architecture"**, Vol 1, Ch.2, www.altera.com/literature/hb/stx/ch_2_vol_1.pdf.
16. **"Avalon Switch Fabric"**, www.altera.com/literature/hb/qts/qts_qii54003.pdf.
17. **"Avalon Interface Specification"**, www.altera.com/literature/manual/mnl_avalon_spec.pdf.
18. Kelley, A.; Pohl, I., **"A Book On C: Programming in C - 2nd Edition"**, Benjamin/Cummings Publishing Company Inc., 1990, ISBN 0-8053-0060-0.

19. Savoj, J.; Razavi, B., **"A 10-Gb/s CMOS Clock and Data Recovery Circuit with a Half-Rate Linear Phase Detector"**, IEEE J. Solid-State Circuits, Vol. 36, No. 5, pp. 761-767, May 2001.

20. Givone, D., **"Digital Principles and Design - 1st Ed."**, McGraw-Hill, 2003, ISBN 0-07-252503-7

21. **"IDT71V416S/L Data Sheet"**, January 2004, Integrated Device Technology Inc.

22. **"SDRAM MT48LC4M32B2 VHDL Memory Model (MT48LC4M32B2.VHD)"**, Micron Semiconductor Products, Inc (www.micron.com), January 2002.

23. **"SRAM IDT71V416 (IDT71016.VHD)"** VHDL Memory Model, Free Model Foundry (www.FreeModelFoundry.com), September 2002.

24. Aweya J., **"On the design of IP routers - Part 1: Router Architectures"**, Journal of Systems Architectures, vol. 46, no. 6, pp. 483-511, April 2000.

25. Choudhury, A.; Hane, E., **"Dynamic Queue Length Thresholds for Shared-Memory Packet Switches"**, IEEE Transactions on Networking , Vol. 6, No. 2, April 1998.

26. Abdo, A.; Bishtein, V.; Clark, S.; Dicorato, P., Lu, D.; Paredes, S.; Taebi, S.; Hall, T., **"Adaptive Packet Switch with an Optical Core (Demonstrator)"**. Photonics North 2004. Ottawa, Canada, 26-29 September 2004

27. Abdo, A.; Bishtein, V.; Dicorato, P., Paredes, S.; Hall, T.,**"Packet switch using reconfigurable optical central crossbars"**, 2005 Conference on Lasers and Electro-Optics Europe; CLEO/Europe 2005, Munich, Germany, 12-17 June 2005, pp. 484-484.

28. Abdo, A., **"Hardware Implementation of a Packet Switch With An Optical Core"**, Masters Thesis, School of Information Technology and Engineering, University of Ottawa, April 2006.

Appendix A. Test Equipment List

Table 3. Lab Equipment

Description	Manufacturer	Part Number
Optical Spectrum Analyzer	Agilent	86142B
Tunable Laser	Agilent	81680A
Lightwave Measurement System	Agilent	8164A
Power Meter	Agilent	81632
Polarization Scrambler	Agilent	11896A
High-Speed Photodetector	Agilent	83044D
ASE Source (EDFA with no input signal)	Oprel	NA
Optical Grating Tunable Filter	JDSU	TB9
Optical Variable Attenuators	JDSU	HA9
Broad linewidth 14-pin Laser	Altitun	NA
RF Spectrum Analyzer	Agilent	8591E
Digital Oscilloscope	Textronix	TDS 520

Appendix B. Inband Crosstalk Equation Derivation

The following section will detail the derivation of the equations used for the crosstalk test station software. The main guidelines and elements for the derivation was obtained from [Reference 8] and [Reference 9];however, the derivation here is modified from the reference in order to obtain measurable data from lab instrumentation. Figure 54 and Figure 55 depicts a functional view of an interface having different reflectivities and also shows the path of the electric fields before being received at a detector. These are generic diagrams to help visualize where the fields are originating from when then are received by the detector; but the diagrams can also be expanded to include all other possible paths the optical field can travel before being detected at the receiver. Figure 55 shows the field travels in two directions incurring both amplitude penalties at the coupled junctions and shows how one path can produce a time-delay of that signal. The amplitude penalties are equivalent to the loss through the device under test, in our case, the wavelength-selectable switch. Both figures demonstrate that light can potentially travel down a specific path to experience a loss and phase delay; however, can also take the route between the input and output port and return back along this same path via the reflectivities of the interface, and exit the same output port as a weakened version. The condition of the mixing of the two fields is normally referred to as homodyne detection.

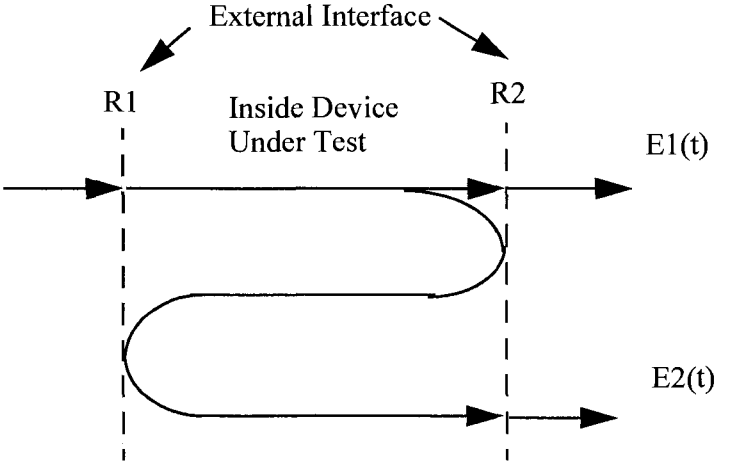


Figure 54. Round-Trip Diagram for Crosstalk Derivation

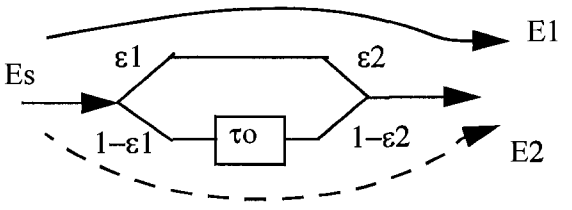


Figure 55. Reflection Interferometer Diagram for Crosstalk Derivation

The optical field incident to the photodetector assuming identical polarization is then:

$$E1 = Ea \cdot \exp(j\omega ot + \phi(t)) \quad (1)$$

$$E2 = Eb \cdot \exp(j(\omega o(t+\tau o) + \phi(t+\tau o))) \quad (2)$$

where Ea = Es * ε1 * ε2 and Eb = Es * (1–ε1) * (1-ε2)

The photodetector current is given by the responsivity of the detector along with the total average power incident on the detector: In other words, the photodetector treats the signal in a square-law sense.

$$iph = \Re ph \times Pavg = \Re ph \times |E_T|^2 \quad (3)$$

Substituting for the E_T into equation #3 and assuming real-valued signals yields:

$$ph = \Re ph \times [E_a \cdot \cos(\omega ot + \phi(t)) + E_b \cdot \cos(\omega o(t+\tau o)) + \phi(t+\tau o)))]$$

Let $A = \omega_o t + \phi(t)$ and $B = \omega_o(t+\tau o) + \phi(t+\tau o)$

$$iph = \Re ph \times [E_a^2 \cdot \cos(A)^2 + 2 \cdot E_a \cdot E_b \cdot \cos(A) \cdot \cos(B) + E_b^2 \cdot \cos(B)^2]$$

Using the trigonometric identity: $\cos(\alpha)\cos(\beta) = 1/2 * \cos(\alpha - \beta) + 1/2 * \cos(\alpha + \beta)$ yields

$$iph = \Re ph \times \left[E_a^2 \cdot \left(\frac{1}{2} + \frac{1}{2} \cdot \cos(2A)\right) + 2 \cdot E_a \cdot E_b \cdot \left(\frac{1}{2} \cdot \cos(A \angle B) + \frac{1}{2} \cdot \cos(A+B)\right) + E_b^2 \cdot \left(\frac{1}{2} + \cos(2B)\right)\right]$$

$$iph = \frac{\Re ph}{2} \times [E_a^2 + E_b^2 + 2 \cdot E_a \cdot E_b \cdot \cos(A \angle B) + E_a^2 \cdot \cos(2A) + E_a^2 \cdot \cos(2B) + 2 \cdot E_a \cdot E_b \cdot \cos(A + B)]$$

Convert all energy values into average power by taking the time-average power and Poynting vector; that is,

$Eo = \sqrt{2|\eta| \cdot Po}$, where Eo is the energy, Po is the time-average power, and η is the intrinsic impedance of the receiver photodiode.

$$iph = \Re ph \cdot |\eta| \times [P_a + P_b + 2 \cdot \sqrt{P_a \cdot P_b} \cdot \cos(A \angle B) + P_a \cdot \cos(2A) + P_b \cdot \cos(2B) + 2 \cdot \sqrt{P_a \cdot P_b} \cdot \cos(A + B)]$$

Since the photodetector does not have infinite bandwidth and is restricted to < 30 GHz (a result of the specification on the internal capacitance), all high-frequency components can be ignored; that is, the cos(A+B), cos(2A) and cos(2B) terms go to zero. Therefore the equation simplifies as follows:

$iph = \Re_T \times [P_a + P_b + 2 \cdot \sqrt{P_a \cdot P_b} \cdot \cos(A \angle B)]$, where Rt is the total responsivity including the material impedance.

Substitute for A & B and simplifying yields:

$iph = \Re_T \times [P_a + P_b + 2 \cdot \sqrt{P_a \cdot P_b} \cdot \cos((2 \cdot \pi \cdot \omega o \cdot \tau o) + \Delta\phi(t, \tau o))]$, where $\Delta\phi(t,\tau o) = \phi(t+\tau o) - \phi(t)$

The equation produces a DC term and a time-varying term. The part that is of interest for the photocurrent is the time-varying term, $i_{noise}(t)$ which contains the transmitted signal phase jitter term which is a random noise process. The noise is typically referred to as the relative intensity noise (RIN). In both [Reference 8] and [Reference 9], the mean-square phase jitter is defined and has a linear dependence on the time delay with a power spectral density spectrum described as a Lorentzian, which is a function of the laser linewidth ($\Delta\nu$). For incoherent interferometric mixing at the receiver, $\Delta\nu * \tau o > 1$, which means that the path length must be much greater than the laser coherence length. This is an important condition in order to keep the power spectrum as mathematically shaped as a Lorentzian as possible without coherent ripple effects on the spectrum.

Let $\omega o = (1/(4*To))$ allows you to reduce the equation so that it is strictly only a function of the phase-jitter:

$$i_{noise}(t) = 2 \cdot \Re_{total} \cdot \sqrt{P_a \cdot P_b} \cdot \sin(\Delta\phi(t, \tau_o)) \quad (4)$$

Noise current is more understood if we can obtain an aggregate time-averaged power to quantify the extent of it's contribution which is typically used in the measure of signal to noise ratio. To calculate the amount of power, Pnoise, by evaluating the autocorrelation (Rnoise) at $\tau = 0$.

$$R_{noise}(\tau) = E[i_{noise}(t+\tau) \cdot i_{noise}(t)]$$

$$R_{noise}(\tau) = E[2 \cdot \Re_{Total} \cdot \sqrt{P_a \cdot P_b} \cdot \sin(\Delta\phi(t+\tau, \tau_o)) \cdot 2 \cdot \Re_{Total} \cdot \sqrt{P_a \cdot P_b} \cdot \sin(\Delta\phi(t, \tau_o))]$$

$$R_{noise}(\tau) = 4 \cdot \Re^2{}_{Total} \cdot P_a \cdot P_b \cdot E\left[\frac{1}{2} \cdot \cos(\Delta\phi(t+\tau, \tau_o) \angle \Delta\phi(t, \tau_o)) \angle \cos(\Delta\phi(t+\tau, \tau_o) + \Delta\phi(t, \tau_o))\right]$$

$$R_{noise}(0) = P_{noise} = 4 \cdot \Re^2{}_{Total} \cdot R_L \cdot P_a \cdot P_b \cdot E\left[\frac{1}{2} \angle \cos(2 \cdot \Delta\phi(0, \tau_o))\right]$$

Note that the time-averaged signal reduces the $\cos(2*\Delta\phi(0,\tau o))$ term to zero, and there is an introduction of the load impedance (R_L) to account for real loads within the measurement which then yields:

$$P_{noise} = 4 \cdot \Re^2{}_{Total} \cdot R_L \cdot P_a \cdot P_b \cdot \frac{1}{2}$$

$$P_{noise} = 2 \cdot \Re^2{}_{Total} \cdot R_L \cdot P_a \cdot P_b \quad (5)$$

If one of the paths is considered as the cross-talk path, then a substitution can be made for one of the powers. Let path "B" represent the cross-talk path, then $Pb = \gamma * Pa$. Substitute this into (5) yields,

$$P_{noise} = 2 \cdot \Re^2{}_{Total} \cdot R_L \cdot P^2{}_a \cdot \gamma$$

Let $K_{cal} = 2 * R^2{}_{total} * R_L$, and Pa = average optical power incident on the detector, then the **total beat-noise power** is given as

$$P_{noise} = K_{cal} \cdot P^2{}_a \cdot \gamma \quad (6)$$

Appendix C. Initialization Simulation Timing Diagrams for Linked-List Memory Manager

This section will show the simulation timing diagrams for the initialization of the off-chip memory by the Linked-List Memory Manager.

Figure 56 to Figure 61 depicts the behavioral timing diagram for the initialization of the SRAM for the idle address region. Since the initialization behavioral results are similar to those of the queue address region (i.e. location of the SDRAM idle addresses), this has been omitted in the timing diagrams. During the initialization time, the SDRAM shared-memory address values are written into SRAM along with their SRAM address pointer values, followed by a read within the same address regions to show that the data has been properly stored into the correct locations.

Figure 62 and Figure 63 depict the behavioral timing diagram for the initialization of both the head and tail pointer address locations. At startup or during a reset condition both the head and tail addresses are equal to the beginning address of the SRAM queue section which contains the beginning address of the SDRAM.

Figure 64 to Figure 66 depicts the behavioral timing diagram for the startup or reset condition of writing the first packet into shared-memory (SDRAM). The algorithm of the linked-list for one of the VOSQ does not take into consideration the first packet; therefore writing the first packet directly to shared-memory and updating the start of the linked-list is executed here.

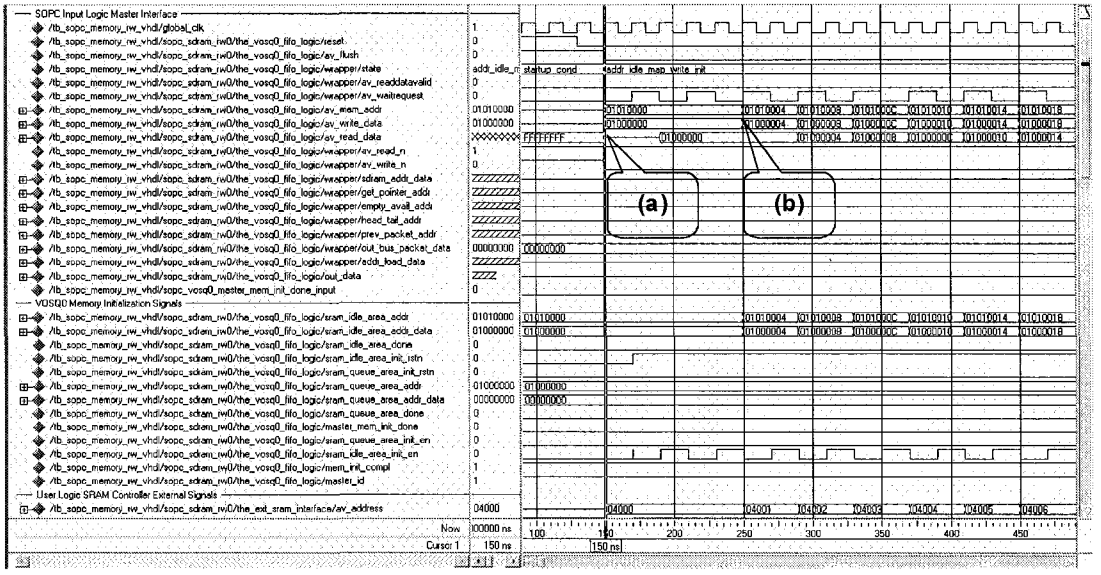


Figure 56. Timing diagram for the SRAM Idle Address Region write initialization (part 1).

(a) Depicts the address and data being sent by the Master Input Logic to the SRAM controller. This data identifies the location in SRAM that will point to the 'idle' SDRAM address required for writing the packet. (b) Depicts the next address and data written which is offset by 4 bytes since the Avalon Bus is byte addressed. A delay is seen before sending the first address and data to the SRAM controller. This delay allows the Avalon Bus to identify a continuous write stream of data to minimize the subsequent writes.

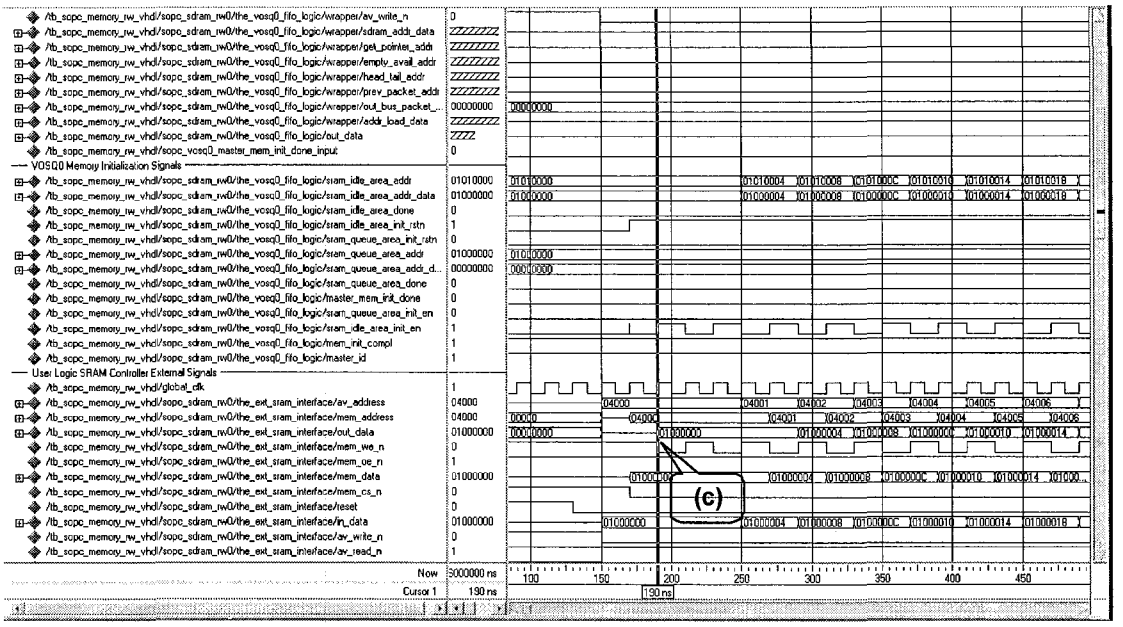


Figure 57. Timing diagram for the SRAM Idle Address Region write initialization (part 2).

(c) Depicts the SRAM model data bus interface to the controller with the correct transmitted data from the Master Input Logic signals presented in Figure 56.

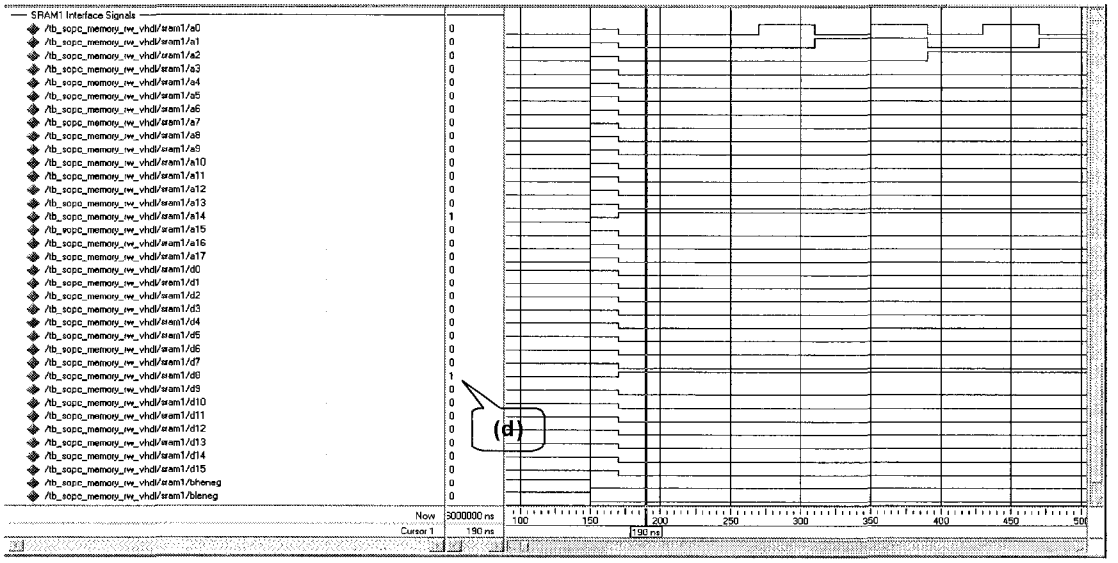


Figure 58. Timing diagram for the SRAM Idle Address Region write initialization (part 3).

(d) Depicts the upper data bytes of the data bus of the SRAM model which confirms that the value of '0x01000000' has been written from the SRAM controller via the Master Input Logic and the Avalon Bus in continuation from the signals shown in Figure 56.

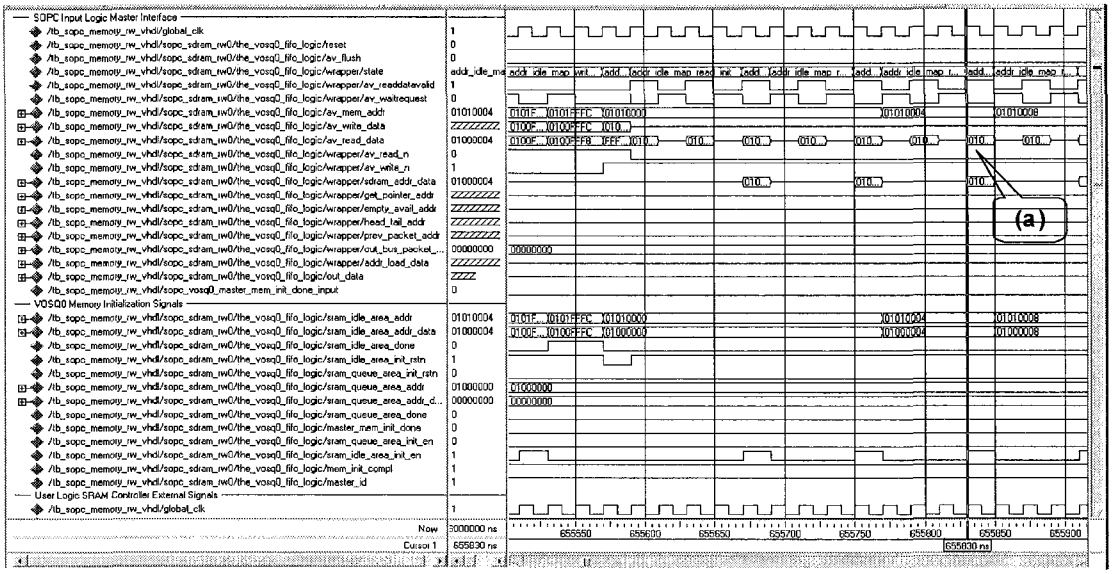


Figure 59. Timing diagram for the SRAM Idle Address Region data verification (part 1).

The initialization also performs a read to each address location within the idle address region of SRAM. (a) depicts the data being captured by the Avalon Bus back to the Master Input Logic. The data at address '0x1010000' is in fact '0x1000000', and the data address '0x1010004' is '0x10000004'. Note that the Avalon Bus takes a longer period of time to do the first read; but this then allows it to identify a continuous read stream to minimize the subsequent reading times.

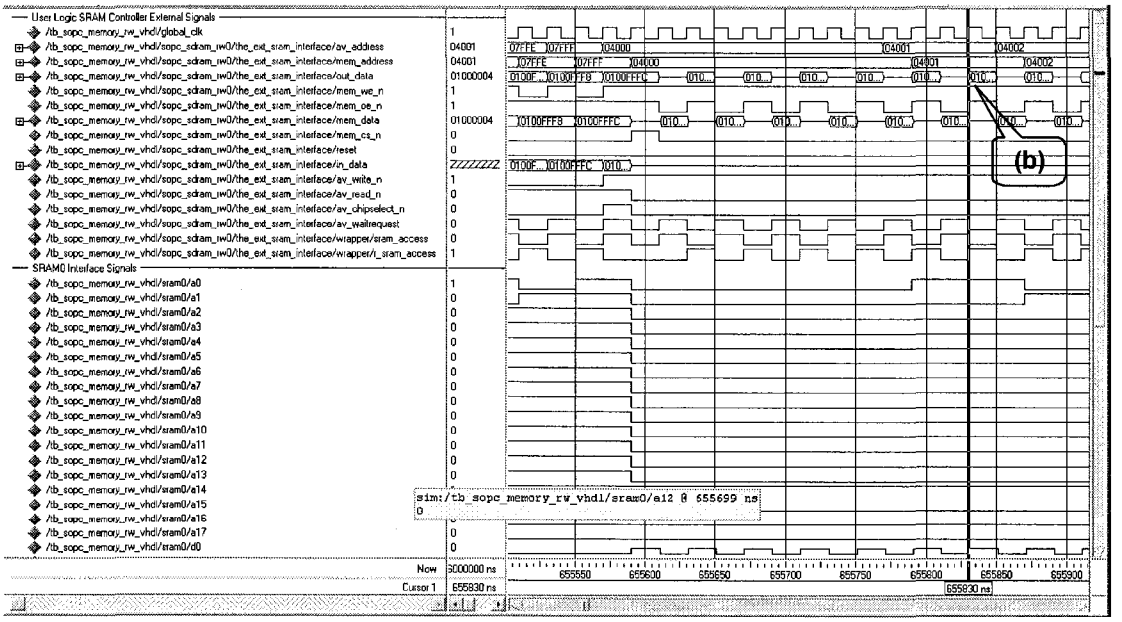


Figure 60. Timing diagram for the SRAM Idle Address Region data verification (part 2).

(b) depicts the parameter 'out_data' which represents the data returned by the SRAM controller to the Avalon Bus from the data bus interface of the SRAM model.

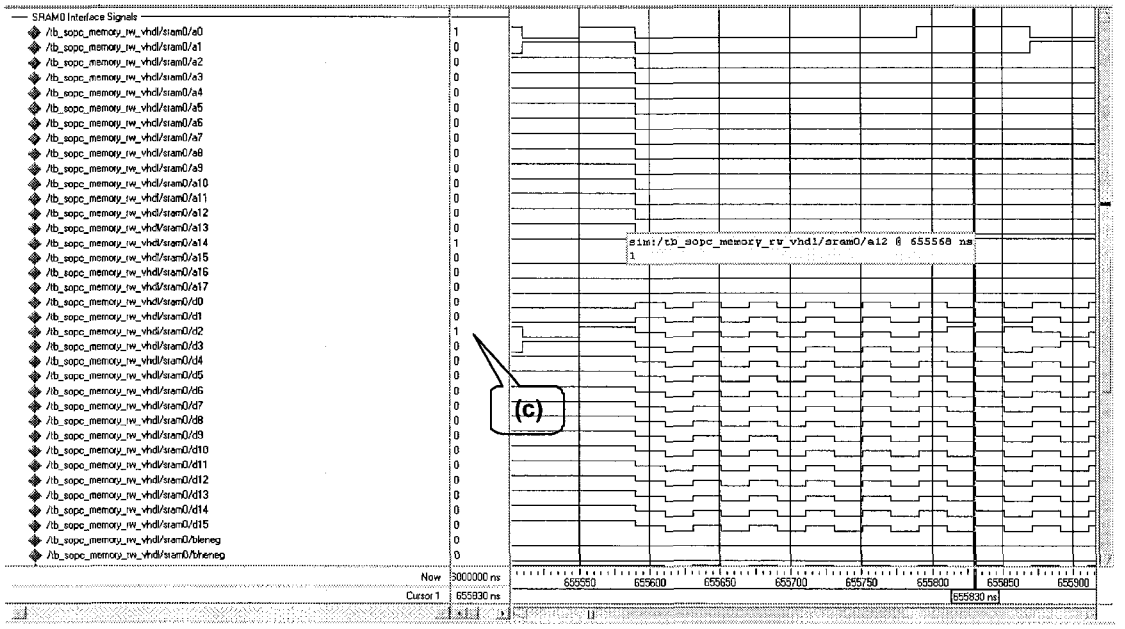


Figure 61. Timing diagram for the SRAM Idle Address Region data verification (part 3).
(c) depicts the last 4 bits of the data retrieved from the SRAM from Figure 59. The third low-significant bit value 'b2' is asserted highlighting a value of '4'. This is the correct value since the data returned is '0x01000004'.

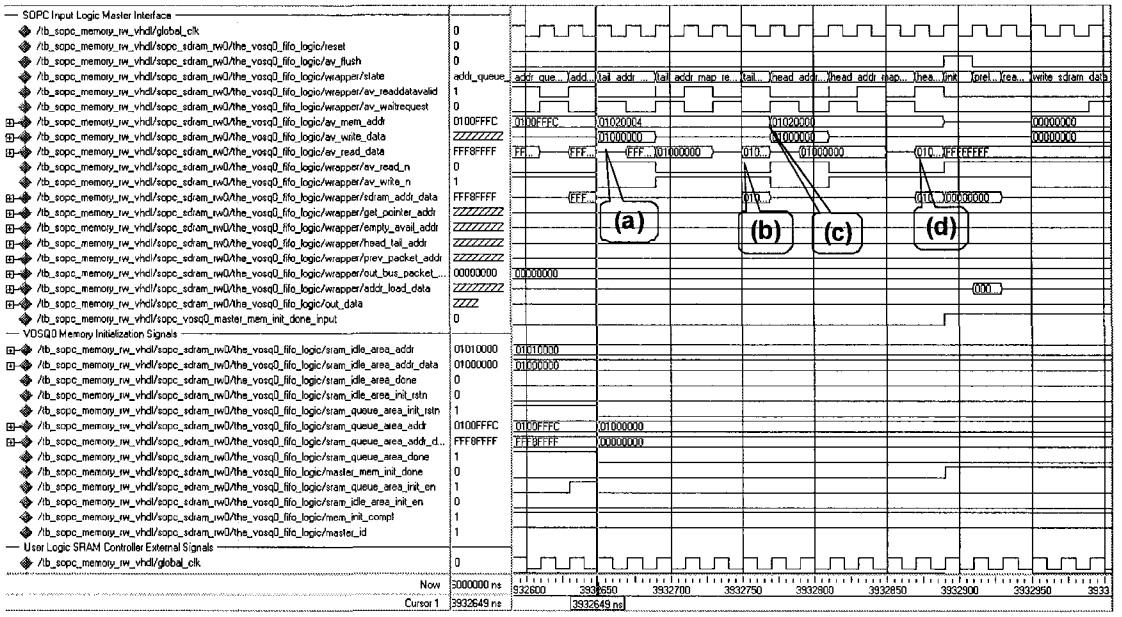


Figure 62. Timing diagram for the Head & Tail Pointer address initialization (part 1).

(a) Depicts the address location of the tail and the data written at this address. (b) Depicts reading the tail address from the same location to confirm that the data was properly written. The data shown here is the correct data of '0x01000000'. (c) Depicts the address location within SRAM of the head and the data being written. The head at startup will equal the tail address. (d) Depicts reading the head address from the same location to confirm the data was properly written. The data shown here is the correct data of '0x01000000'.

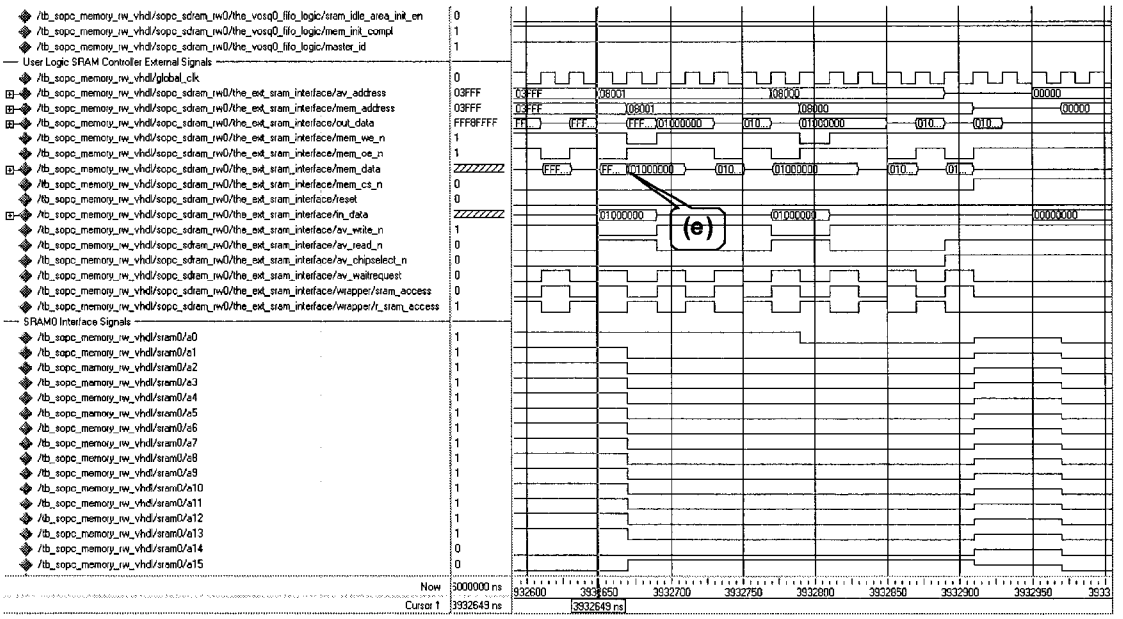


Figure 63. Timing diagram for the Head & Tail Pointer address initialization (part 2).
(e) Depicts the time at which the tail address is transmitted from the controller directly to the SRAM memory model data bus.

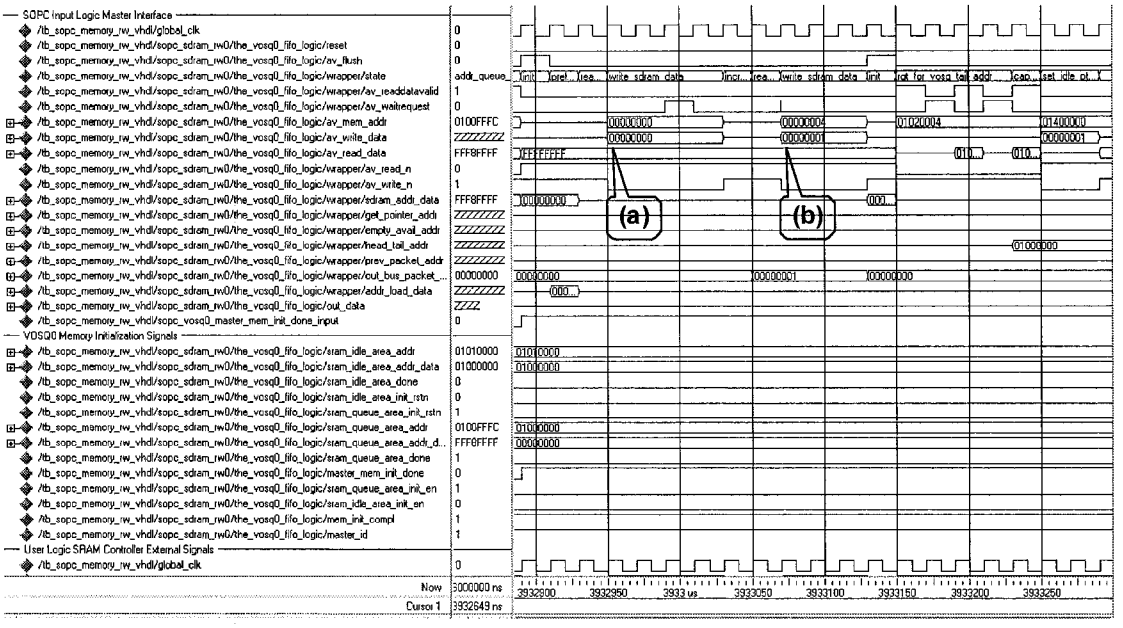


Figure 64. Timing diagram for writing the first packet into SDRAM (part 1).

This also updates the VOSQ. (a) Depicts the point in time where the Master Input Logic writes the first packet fragment from the Prestore FIFO onto the beginning address of the SDRAM. The data is transmitted to the SDRAM controller from the Master Input Logic block. (b) Depicts the second fragment of the packet being stored at the next idle address (4 bytes offset) since a packet is defined here as two fragments.

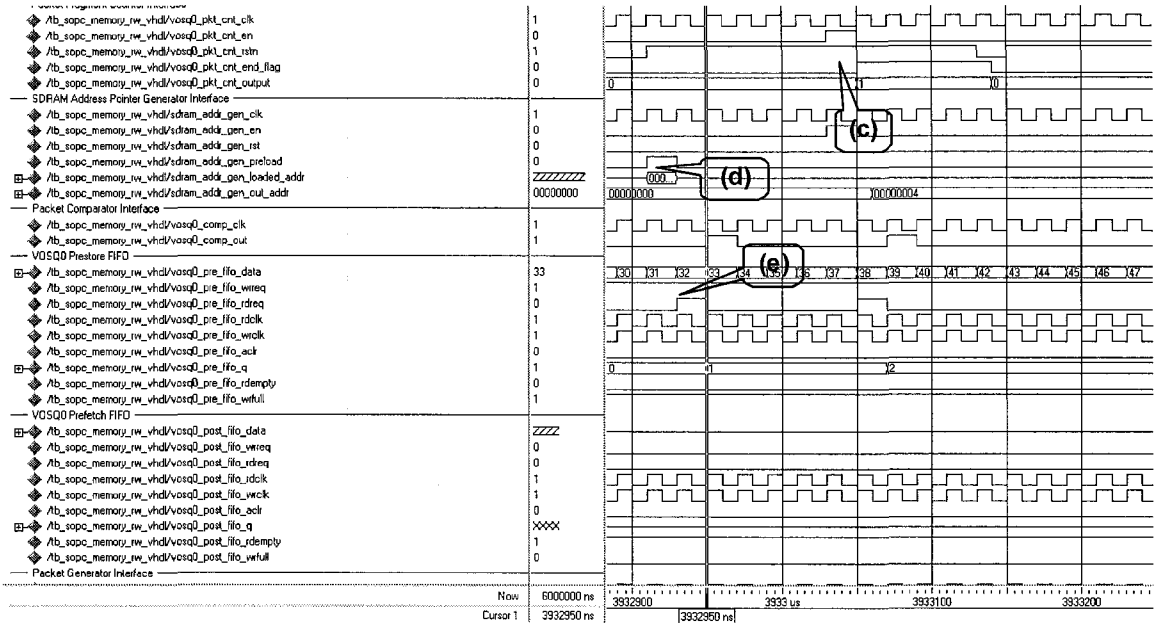


Figure 65. Timing diagram for writing the first packet into SDRAM (part 2).

This also updates the VOSQ. (c) Depicts the signal that is asserted by the Packet Fragment Counter when the number of fragments has been reached. (d) Depicts when the Master Input Logic preludes the SDRAM address generator with the idle address extracted from the SRAM. (e) Depicts the assertion of the Prestore FIFO read request signal by the Master Input Logic The Prestore FIFO contains incremental values starting from '0x00000000' onwards. The first data at the output when the read request is asserted is '0x00000000', as expected

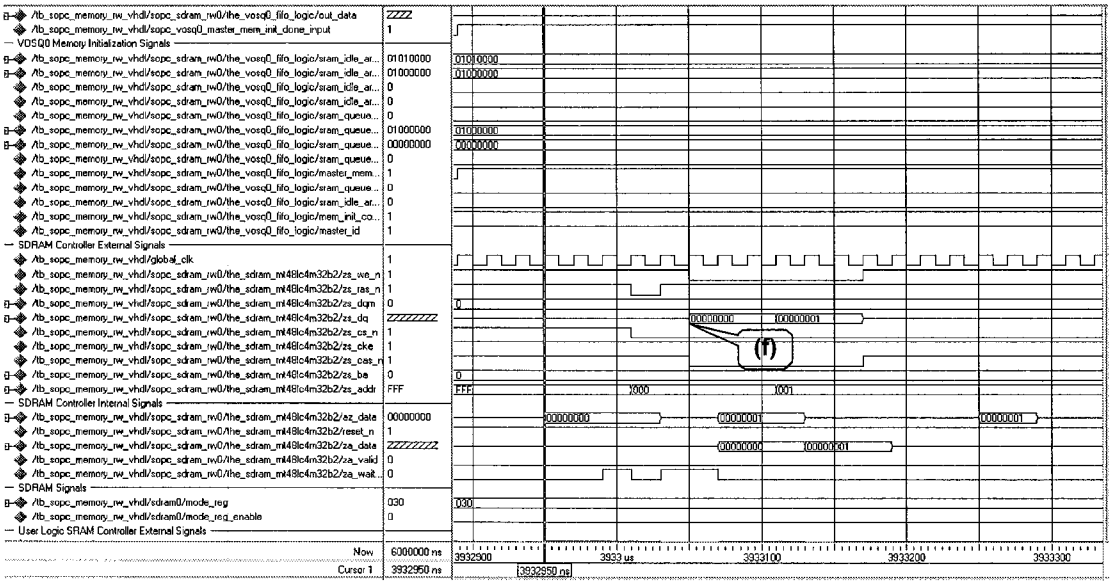


Figure 66. Timing diagram for writing the first Packet into SDRAM (part 3).

This also updates the VOSQ. (f) Depicts the SDRAM controller placing the data for each address location onto the SDRAM data bus. Address location '0' gets the value '0', then '1' gets the value '1'. Both combined represent the first and second fragment of a complete packet.

Appendix D. Interface Signals For Memory Manager Peripheral Blocks

This section will provide the interface signal definitions for each of the main logical state machine blocks; that is, the Master Input Logic, the Idle Address Generator and the External Memory Tristate Bridge, respectively. Each of the logic block interfaces are provided and identified in the following figures:

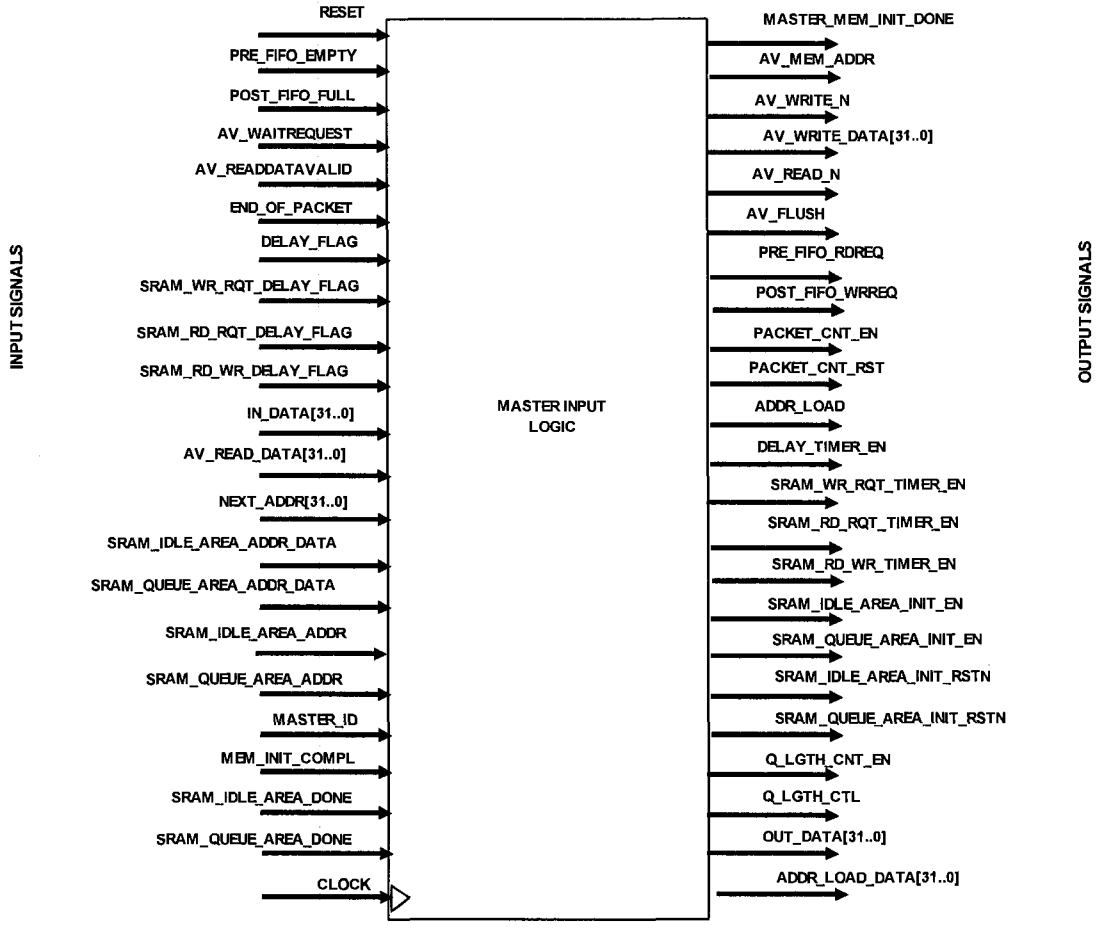


Figure 67. Master Input Logic interface signal block diagram

Table 4. Master Input Logic interface signal definitions

Signal Name	Signal Definition
CLOCK	Clock frequency required for the logical interface.

Table 4. Master Input Logic interface signal definitions

Signal Name	Signal Definition
RESET	Reset signal for the state machine.
PRE_FIFO_EMPTY	Empty status flag from prestore FIFO
POST_FIFO_FULL	Full status flag from the prefetch FIFO
AV_WAITREQUEST	Waitrequest signal required to interface with the Avalon Bus. This signal allows the peripheral to wait until data is available from the bus.
AV_READDATAVALID	Readdatavalid signal required to interface with the Avalon Bus. This signal allows data to be captured when it is valid from the bus.
END_OF_PACKET	Signal that is used to flag when a fragmented stream that makes-up one packet is completed.
DELAY_FLAG	Status signal to indicate that the delay time has been reached.
SRAM_WR_RQT_DELAY_FLAG	Status signal that indicates the delay time before a waitrequest signal gets asserted from the Avalon Bus during a write request to SRAM.
SRAM_RD_RQT_DELAY_FLAG	Status signal that indicates the delay time before a waitrequest signal gets asserted from the Avalon Bus during a read request to SRAM.
SRAM_RD_WR_RQT_DELAY_FLAG	Status signal that indicates the delay time required between consecutive write-to-read requests to/from SRAM.
IN_DATA	Packet data from prestore FIFO.
AV_READ_DATA	The data read back from the Avalon Bus to the peripheral.
NEXT_ADDR	The SDRAM address for reading/writing a packet provided by the SDRAM Address Generator.
SRAM_IDLE_AREA_ADDR_DATA	The SRAM idle area address value.
SRAM_QUEUE_AREA_ADDR_DATA	The SRAM queue area address value.
SRAM_IDLE_AREA_ADDR	The SRAM idle area address
SRAM_QUEUE_AREA_ADDR	The SRAM queue area address
MASTER_ID	Signal that identifies whether the peripheral is the 'primary' master.
MEM_INIT_COMPL	Status signal that represents when the 'primary' master is complete with the SRAM address map initialization.
SRAM_IDLE_AREA_DONE	Status signal that indicates when the initialization of the idle address space in SRAM has been completed.
SRAM_QUEUE_AREA_DONE	Status signal that indicates when the initialization of the queue address space in SRAM has been completed.
MASTER_MEM_INIT_DONE	Status signal that represents the completion of the SRAM address map initialization.

Table 4. Master Input Logic interface signal definitions

Signal Name	Signal Definition
AV_MEM_ADDR	The Avalon Bus address
AV_WRITE_N	The Avalon Bus write signal
AV_WRITE_DATA	The data that is sent from the peripheral to the Avalon Bus.
AV_READ_N	The Avalon Bus read signal
AV_FLUSH	The signal that indicates to the Avalon Bus to flush all pending data.
PRE_FIFO_RDREQ	Prestore FIFO read request.
POST_FIFO_WRREQ	Prefetch FIFO write request.
PACKET_CNT_EN	Enable signal used to start the packet fragment counter.
PACKET_CNT_RST	Reset signal used to reset the packet fragment counter.
ADDR_LOAD	The preload signal for the SDRAM address generator.
DELAY_TIMER_EN	Delay timer enable signal.
SRAM_WR_RQT_TIMER_EN	SRAM write request timer enable signal.
SRAM_RD_RQT_TIMER_EN	SRAM read request timer enable signal.
SRAM_RD_WR_RQT_TIMER_EN	SRAM write-to-read request timer enable signal.
SRAM_IDLE_AREA_INIT_EN	Enable signal for the initialization address map for the idle address space in SRAM.
SRAM_QUEUE_AREA_INIT_EN	Enable signal for the initialization address map for the queue address space in SRAM.
Q_LGTH_CNT_EN	Enable signal for the queue length counter.
Q_LGTH_CTL	Control signal that identifies whether to increase or decrease the queue length counter. ('1' = increase, and '0' = decrease).
OUT_DATA	Packet data to the prefetch FIFO.
ADDR_LOAD_DATA	The preloaded SDRAM address value.

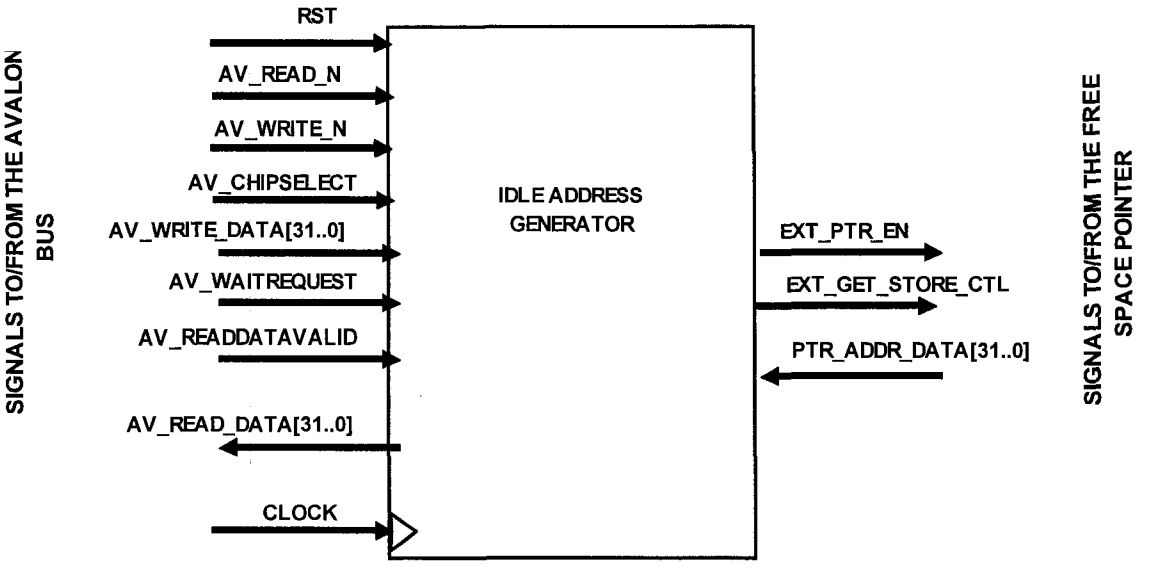


Figure 68. Idle Address Generator interface signal block diagram

Table 5. Idle Address Generator interface signal definitions

Signal Name	Signal Definition
CLOCK	Clock frequency required for the logical interface.
RESET	Reset signal for the state machine.
AV_WAITREQUEST	Waitrequest signal required to interface with the Avalon Bus. This signal stalls the bus for any master accessing this peripheral until the peripheral is completed its task and is ready to transmit.
AV_READDATAVALID	Readdatavalid signal required to interface with the Avalon Bus. This signal indicates that the address data provided to the bus is valid.
AV_READ_DATA	The data written to the Avalon Bus.
AV_WRITE_N	The Avalon Bus write signal
AV_WRITE_DATA	The data that is sent to the peripheral from the Avalon Bus.
AV_READ_N	The Avalon Bus read signal
AV_CHIPSELECT	The Avalon address and read request decode signal from the Avalon Bus.
PTR_ADDR_DATA	The pointer data obtained from the external address generator.
EXT_PTR_EN	Enable signal for the free-space pointer counter.
EXT_GET_STORE_CTL	Control signal for the free-space pointer counter.

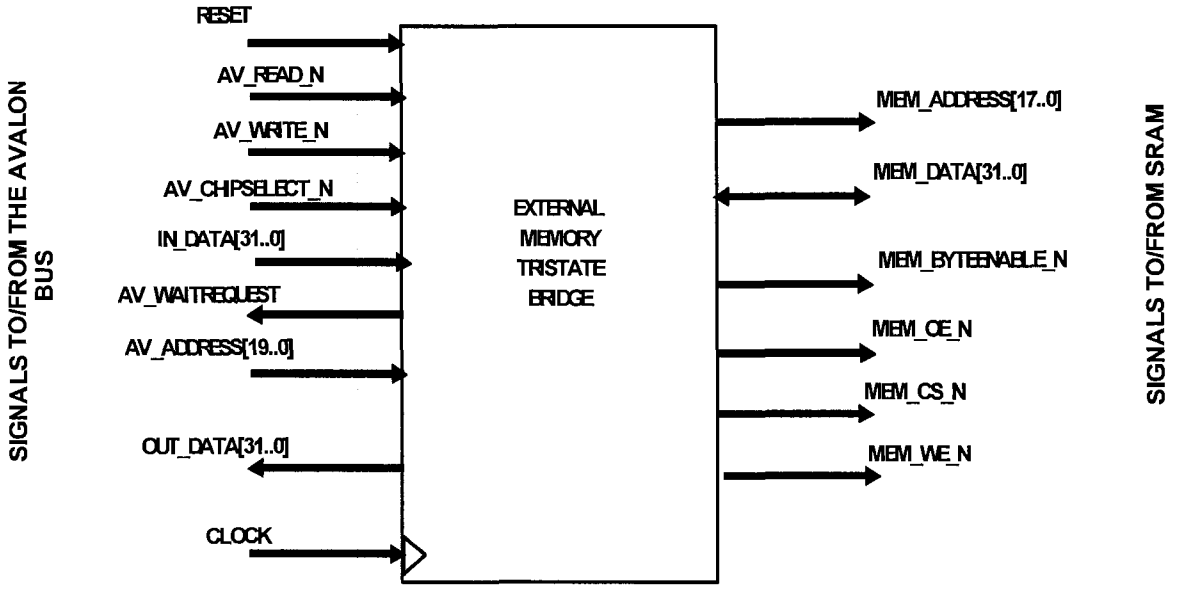


Figure 69. Off-chip SRAM memory tri-state bridge interface signal block diagram

Table 6. Off-Chip SRAM memory tri-state bridge interface signal definitions

Signal Name	Signal Definition
CLOCK	Clock frequency required for the logical interface.
RESET	Reset signal for the state machine.
AV_WAITREQUEST	Waitrequest signal required to interface with the Avalon Bus. This signal stalls the bus for any master accessing this peripheral until the peripheral is completed its task and is ready to transmit.
AV_READDATAVALID	Readdatavalid signal required to interface with the Avalon Bus. This signal indicates that the address data provided to the bus is valid.
OUT_DATA	The data written to the Avalon Bus.
AV_WRITE_N	The Avalon Bus write signal
IN_DATA	The data that is sent to the peripheral from the Avalon Bus.
AV_READ_N	The Avalon Bus read signal
AV_CHIPSELECT	The Avalon address and read request decode signal from the Avalon Bus.
AV_ADDRESS	The address obtained from the Avalon bus.
MEM_ADDRESS	The address provided to the off-chip memory.

Table 6. Off-Chip SRAM memory tri-state bridge interface signal definitions

Signal Name	Signal Definition
MEM_DATA	The bidirectional data line to the off-chip memory.
MEM_BYTEENABLE_N	The byte enable signal for the off-chip memory.
MEM_OE_N	The tristate output enable signal for the off-chip memory.
MEM_CS_N	The chipselect signal for the off-chip memory.
MEM_WE_N	The write enable signal for the off-chip memory.

Appendix E. VHDL Source Code For Linked-List Memory Manager Peripherals

This section will detail all the VHDL code for each of the peripherals designed for the linked-list memory manager and the test bench. The code was generated, compiled and synthesized using Quartus 5.0 Simulator from Altera™ Corporation and imported to ModelSim 6.0 SE by Cadence for performing system behavioral simulations.

E.1 SRAM Memory Filler.VHD

This module generates both the addresses and data values needed by the Master Input Logic peripheral at initialization to initialize the off-chip SRAM. The data is split into 2 x 16 bit halves whereby the upper 16 bits is the actual SDRAM address, and the lower half is provisioned with FF's. These FFs will be updated later in the Master peripheral to contain the "next" pointer address for the packet. Since the Avalon Bus uses Native Address Alignment & Dynamic Bus Sizing methods for addressing slave peripherals with either same or different bus widths, the address to the memory will require to be incremented by the appropriate offset. Here we use [offset] + 0x4 where 'offset' is obtained by the SOPC builder address column, and will be listed as a constant within this module to initiate the address sequence.

```
-- Library Clause
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.std_logic_unsigned.ALL;

-- Entity Declaration
ENTITY SRAM_Memory_Filler IS
PORT(
    CLK     :IN STD_LOGIC;
    MEM_CNT_EN          :IN STD_LOGIC;
    RST_N               :IN STD_LOGIC;

    MEM_SEQ_COMPL_FLAG OUT STD_LOGIC;
    MEM_FILL_ADDR OUT STD_LOGIC_VECTOR(31 DOWNTO 0);
    OUT_ADDR      OUT STD_LOGIC_VECTOR(31 DOWNTO 0)
);
END ENTITY SRAM_Memory_Filler;

ARCHITECTURE main_cntr_rtl OF SRAM_Memory_Filler IS

constant START_MEM_ADDR:std_logic_vector(31 DOWNTO 0) := x"01000000";
constant END_MEM_ADDR:std_logic_vector(31 DOWNTO 0) := x"0100FFFC";
constant MEM_FILL_START_ADDR:std_logic_vector(31 DOWNTO 0) := x"01010000";
constant MEM_FILL_END_ADDR:std_logic_vector(31 DOWNTO 0) := x"0101FFFC";

BEGIN

PROCESS (CLK, RST_N)
    -- Define parameters
    variable seq_compl_status       std_logic;
```

```
    variable init_value_flag        :std_logic;
    variable curr_addr              :std_logic_vector(31 DOWNTO 0);
    variable curr_mem_fill_addr     :std_logic_vector(31 DOWNTO 0);
BEGIN

    IF (RST_N = '0') THEN
            curr_addr := START_MEM_ADDR;
            curr_mem_fill_addr := MEM_FILL_START_ADDR;
            seq_compl_status := '0';
            init_value_flag := '1';
            MEM_SEQ_COMPL_FLAG <= '0';
            OUT_ADDR <= START_MEM_ADDR;
            MEM_FILL_ADDR <= MEM_FILL_START_ADDR;

    ELSIF (CLK'EVENT AND CLK = '1') THEN

            IF MEM_CNT_EN = '1' THEN

                    IF (curr_addr /= END_MEM_ADDR and init_value_flag = '0') THEN
                            curr_addr := curr_addr + 4;
                            curr_mem_fill_addr := curr_mem_fill_addr + 4;
                    ELSE
                            curr_addr := START_MEM_ADDR;
                            curr_mem_fill_addr := MEM_FILL_START_ADDR;
                    END IF;
                    init_value_flag := '0';

                    IF (curr_addr /= END_MEM_ADDR) THEN
                            seq_compl_status := '0';
                    ELSE
                            seq_compl_status := '1';
                    END IF;

            ELSE

                    curr_addr := curr_addr;
                    curr_mem_fill_addr := curr_mem_fill_addr;
                    seq_compl_status := seq_compl_status;
                    init_value_flag := init_value_flag;

            END IF;

            MEM_SEQ_COMPL_FLAG <= seq_compl_status;
            OUT_ADDR <= curr_addr;
            MEM_FILL_ADDR <= curr_mem_fill_addr;

            END IF;
END PROCESS;


END ARCHITECTURE main_cntr_rtl;
```

E.2 SDRAM Memory Filler.VHD

This module generates both the addresses and data values needed by the Master Input Logic peripheral at initialization to initialize the off-chip SDRAM. Since the Avalon Bus uses Native Address Alignment & Dynamic Bus Sizing methods for addressing slave peripherals with either same or different bus widths, the address to the memory will require to be incremented by the appropriate offset. Here we use [offset] + 0x4 where 'offset' is obtained by the SOPC builder address column, and will be listed as a constant within this module to initiate the address sequence.

```
-- Library Clause
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.std_logic_unsigned.ALL;

-- Entity Declaration
ENTITY SDRAM_Memory_Filler IS
    PORT(
            CLK                 :IN STD_LOGIC;
            MEM_CNT_EN          :IN STD_LOGIC;
            RST_N               :IN STD_LOGIC;

            MEM_SEQ_COMPL_FLAG  :OUT STD_LOGIC;
            MEM_FILL_ADDR       :OUT STD_LOGIC_VECTOR(31 DOWNTO 0);
            OUT_ADDR            :OUT STD_LOGIC_VECTOR(31 DOWNTO 0)
    );
END ENTITY SDRAM_Memory_Filler;

ARCHITECTURE main_cntr_rtl OF SDRAM_Memory_Filler IS

    constant START_MEM_ADDR:std_logic_vector(31 DOWNTO 0) := x"00000000";
    constant MEM_FILL_START_ADDR:std_logic_vector(31 DOWNTO 0) := x"01000000";
    constant MEM_FILL_END_ADDR:std_logic_vector(31 DOWNTO 0) := x"0100FFFC";

BEGIN

PROCESS (CLK, RST_N)
    -- Define parameters
    variable seq_compl_status     :std_logic;
    variable init_value_flag :    std_logic;
    variable curr_addr            :std_logic_vector(31 DOWNTO 0);
    variable curr_mem_fill_addr   :std_logic_vector(31 DOWNTO 0);
BEGIN
    IF (RST_N = '0') THEN
            curr_addr := START_MEM_ADDR;
            seq_compl_status := '0';
            init_value_flag := '1';
            curr_mem_fill_addr := MEM_FILL_START_ADDR;
            MEM_SEQ_COMPL_FLAG <= '0';
            OUT_ADDR <= START_MEM_ADDR;
            MEM_FILL_ADDR <= MEM_FILL_START_ADDR;

    ELSIF (CLK'EVENT AND CLK = '1') THEN
```

```
            IF MEM_CNT_EN = '1' THEN

                  IF (curr_mem_fill_addr /= MEM_FILL_END_ADDR and init_value_flag = '0') THEN
                        curr_addr := curr_addr + 8;  -- offset of 8 required since packet format takes 2 x 32 bit
                         -- address locations within SDRAM.
                        curr_mem_fill_addr := curr_mem_fill_addr + 4;
                  ELSE
                        curr_addr := START_MEM_ADDR;
                        curr_mem_fill_addr := MEM_FILL_START_ADDR;
                  END IF;
                        init_value_flag := '0';

                  IF (curr_mem_fill_addr /= MEM_FILL_END_ADDR) THEN
                        seq_compl_status := '0';
                  ELSE
                        seq_compl_status := '1';
                  END IF;

            ELSE

                  curr_addr := curr_addr;
                  seq_compl_status := seq_compl_status;
                  init_value_flag := init_value_flag;
                  curr_mem_fill_addr := curr_mem_fill_addr;

            END IF;

            MEM_SEQ_COMPL_FLAG <= seq_compl_status;
            OUT_ADDR <= curr_addr(15 downto 0) & "1111111111111111";  -- lower half of data space is "FF's"
and upper half
              -- is the actual SDRAM address.  Data width divided in half
            MEM_FILL_ADDR <= curr_mem_fill_addr;

    END IF;
END PROCESS;


END ARCHITECTURE main_cntr_rtl;
```

E.3 SRAM Read Delay.VHD

This module generates the time delay required to READ from the SRAM Controller slave peripheral from the Avalon Bus. A delay of 1 x Tclk is required, where fclk = 50 MHz. Therefore, a 4-bit counter is more than sufficient. Once the delay is achieved the output of the logical block holds-off the count sequence.

```
-- Library Clause
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

-- Entity Declaration
ENTITY SRAM_READ_DELAY IS
    GENERIC(
            READ_DELAY  :POSITIVE :=1
            );
    PORT(
            CLK             :IN STD_LOGIC;
            TIMER_EN_N      :IN STD_LOGIC;

            CNT_OUT         :OUT INTEGER RANGE 0 to 15;
            DELAY_FLAG      :OUT STD_LOGIC
    );
END ENTITY SRAM_READ_DELAY;

ARCHITECTURE main_cntr_rtl OF SRAM_READ_DELAY IS

BEGIN

    PROCESS (CLK, TIMER_EN_N)
            VARIABLE count  :INTEGER RANGE 0 to 15;
    BEGIN

            IF (TIMER_EN_N = '0') THEN
                    count := 0;
            ELSIF (CLK'EVENT AND CLK = '1') THEN
                    IF (count = 15) THEN
                            count := count;
                    ELSE
                            count := count + 1;
                    END IF;
            END IF;
            CNT_OUT <= count;

            -- Decode the output counter for the correct delay
            IF (count >= READ_DELAY) THEN
                    DELAY_FLAG <= '1';
            ELSE
                    DELAY_FLAG <= '0';
            END IF ;

    END PROCESS;
END ARCHITECTURE main_cntr_rtl;
```

E.4 Write SDRAM Ctrlr Delay.VHD

This module generates the time delay required to WRITE to the SDRAM controller slave peripheral from the Avalon Bus. A delay of 2 x Tclk is required; therefore, a 4-bit counter is implemented. Once the delay is achieved then this module will hold the count sequence.

```
-- Library Clause
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

-- Entity Declaration
ENTITY Write_SDRAM_Ctrlr_Delay IS
    GENERIC(
            WRITE_DELAY :POSITIVE :=2
            );
    PORT(
            CLK             :IN STD_LOGIC;
            TIMER_EN_N      :IN STD_LOGIC;

            CNT_OUT         :OUT INTEGER RANGE 0 to 15;
            DELAY_FLAG      :OUT STD_LOGIC
    );
END ENTITY Write_SDRAM_Ctrlr_Delay;

ARCHITECTURE main_cntr_rtl OF Write_SDRAM_Ctrlr_Delay IS

BEGIN

    PROCESS (CLK, TIMER_EN_N)
            VARIABLE count  :INTEGER RANGE 0 to 15;
    BEGIN

            IF (TIMER_EN_N = '0') THEN
                    count := 0;
            ELSIF (CLK'EVENT AND CLK = '1') THEN
                    IF (count = 15) THEN
                            count := count;
                    ELSE
                            count := count + 1;
                    END IF;
            END IF;
            CNT_OUT <= count;

            -- Decode the output counter for the correct delay
            IF (count >= WRITE_DELAY) THEN
                    DELAY_FLAG <= '1';
            ELSE
                    DELAY_FLAG <= '0';
            END IF ;

    END PROCESS;
END ARCHITECTURE main_cntr_rtl;
```

E.5 Packet Counter VHDL.VHD

This module is a counter that only counts up to the number of fragments within the Prestore FIFO which represents one complete packet and is a multiple of the bus width. For the Avalon Bus, this is 32-bits. As an example, for a 64-bit packet and a 32-bit word (bit-wide) FIFO and a 32-bit wide bus, the number of fragments is 2; that is DATAWIDTH = 2.

```
-- Library Clause
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.std_logic_unsigned.ALL;

-- Entity Declaration
ENTITY packet_counter_VHDL IS
    GENERIC(
      DATAWIDTH          INTEGER:= 2
       );
    PORT(
            CLK                 :IN STD_LOGIC;
            RQT_EN              :IN STD_LOGIC;
            RST                 :IN STD_LOGIC;

            PKT_END_FLAG:OUT STD_LOGIC;
            PACKET_NUMBER:OUT INTEGER RANGE 0 TO (DATAWIDTH -1)
    );
END ENTITY packet_counter_VHDL;

ARCHITECTURE main_cntr_rtl OF packet_counter_VHDL IS
BEGIN

    counter0: PROCESS(CLK)
            VARIABLE count1:INTEGER RANGE 0 TO (DATAWIDTH-1);
    BEGIN
            IF (RST = '0') THEN
                    count1 := 0;
            ELSIF (CLK'EVENT and CLK = '1') THEN
                    IF (RQT_EN = '1') THEN
                            IF count1 = (DATAWIDTH - 1) THEN
                                count1 := 0;
                            ELSE
                                count1 := count1 + 1;
                            END IF;
                    ELSE
                            count1 := count1;
                    END IF;
            END IF;
            PACKET_NUMBER <= count1;

            -- Decode the output counter for the end of the sequence
            IF count1 = (DATAWIDTH -1) THEN
                    PKT_END_FLAG <= '1';
```

```
            ELSE
                PKT_END_FLAG <= '0';
            END IF ;

    END PROCESS counter0;

END ARCHITECTURE main_cntr_rtl;
```

E.6 SDRAM Packet Addr Counter.VHD

This module is a counter that generates an address required for the read/write operations to the SDRAM controller for each packet. It is enabled based on the number of fragments that make-up a packet since the bus width is limited to 32 bits. Since the Avalon Bus uses Native Address Alignment & Dynamic Bus Sizing methods for addressing slave peripherals with either same or different bus widths, the address to the memory will require to be incremented by the appropriate offset. Here we use [offset] + 0x4 where 'offset' is obtained by SOPC builder address column.

```
-- Library Clause
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.std_logic_unsigned.ALL;

-- Entity Declaration
ENTITY SDRAM_Packet_Addr_Counter IS
    PORT(
            CLK             :IN STD_LOGIC;
            RQT_EN          :IN STD_LOGIC;
            RST             :IN STD_LOGIC;
            PRE_LOAD        :IN STD_LOGIC;
            LOADED_ADDR:IN STD_LOGIC_VECTOR(31 DOWNTO 0);

            OUT_ADDR        :OUT STD_LOGIC_VECTOR(31 DOWNTO 0)
    );
END ENTITY SDRAM_Packet_Addr_Counter;

ARCHITECTURE main_cntr_rtl OF SDRAM_Packet_Addr_Counter IS
    SIGNAL count1:STD_LOGIC_VECTOR(31 DOWNTO 0);

BEGIN

    counter0: PROCESS(CLK)
    BEGIN
            IF (RST = '1') THEN
                count1 <= (others => '0');
            ELSIF (CLK'EVENT and CLK = '1') THEN
                IF (PRE_LOAD = '1' OR count1 = "11111111111111111111111111111111") THEN
                    count1 <= LOADED_ADDR;
                ELSIF (RQT_EN = '1') THEN
                    count1 <= count1 + 4;
                ELSE
```

```
                    count1 <= count1;
                END IF;
            END IF;
            OUT_ADDR <= count1;
    END PROCESS counter0;

END ARCHITECTURE main_cntr_rtl;
```

E.7 Memory Address Generator.VHD

This module is a slave peripheral used for communicating with the Avalon Bus and directly with the Master Input Logic peripheral. It is responsible in obtaining a SDRAM address from the 'free' or 'used' pointers stored within the SRAM for a specific virtual sector queues.

```
-- Library Clause
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

-- Entity Declaration
ENTITY MEMORY_ADDRESS_GENERATOR IS
PORT(
    CLK                 : IN   STD_LOGIC;
    RESET               : IN STD_LOGIC;
    AV_READ_N           : IN STD_LOGIC;
    AV_WRITE_N          : IN STD_LOGIC;
    AV_CHIPSELECT       : IN STD_LOGIC;
    AV_WRITE_DATA       : IN STD_LOGIC_VECTOR (31 DOWNTO 0);
    PTR_ADDR_DATA       : IN STD_LOGIC_VECTOR (31 DOWNTO 0);
    AV_WAITREQUEST      :OUTSTD_LOGIC;
    AV_READDATAVALID : OUTSTD_LOGIC;
    AV_READ_DATA        : OUTSTD_LOGIC_VECTOR (31 DOWNTO 0);
    EXT_PTR_EN          : OUTSTD_LOGIC;
    EXT_GET_STORE_CTL: OUTSTD_LOGIC

);

END ENTITY MEMORY_ADDRESS_GENERATOR;

-- Architecture Body
ARCHITECTURE main_rtl OF MEMORY_ADDRESS_GENERATOR IS

-- State Machine Declaration
TYPEEXCHANGE_STATEIS   (INIT,   EN_PTR_ADDR_COUNTER,   AVALON_PTR_ADDR_CAPTURE,
ACQ_PTR_ADDR, CAPTURE_CTL_DATA);
SIGNAL  state, nextstate :EXCHANGE_STATE;

SIGNAL  ptr_addr_ctl  :STD_LOGIC;

BEGIN
```

```
    state_register: PROCESS (CLK,  RESET)
    BEGIN
            -- Trigger on the positive edge of the clock.  Synchronous operation.
            -- Reset asynchronously to the initialization state if RESET is pulled high
            IF (RESET = '1') THEN
                    state <= INIT;
            ELSIF (CLK'EVENT AND CLK = '1') THEN
                    state <= nextstate;
            END IF;
    END PROCESS state_register;

    rw_control: PROCESS (state, AV_READ_N, AV_WRITE_N, AV_CHIPSELECT, AV_WRITE_DATA,
PTR_ADDR_DATA, ptr_addr_ctl)
    BEGIN
            CASE state IS
                    WHEN INIT =>
                            AV_WAITREQUEST <= '0';
                            AV_READDATAVALID <= '0';
                            AV_READ_DATA <= (others => 'Z');
                            EXT_PTR_EN <= '0';
                            EXT_GET_STORE_CTL <= '1';

                            ptr_addr_ctl <= ptr_addr_ctl;

                            IF (AV_CHIPSELECT = '1') THEN
                                IF (AV_READ_N = '0') THEN
                                    nextstate <= EN_PTR_ADDR_COUNTER;
                                ELSE
                                    IF (AV_WRITE_N = '0') THEN
                                        nextstate <= CAPTURE_CTL_DATA;
                                    ELSE
                                        nextstate <= INIT;
                                    END IF;
                                END IF;
                            ELSE
                                nextstate <= INIT;
                            END IF;

                    WHEN EN_PTR_ADDR_COUNTER =>

                            AV_WAITREQUEST <= '1';
                            AV_READDATAVALID <= '0';
                            AV_READ_DATA <= (others => 'Z');
                            EXT_PTR_EN <= '1';
                            EXT_GET_STORE_CTL <= ptr_addr_ctl;

                            nextstate <= ACQ_PTR_ADDR;

                    WHEN ACQ_PTR_ADDR =>

                            AV_WAITREQUEST <= '1';
```

```
                    AV_READDATAVALID <= '0';
                    AV_READ_DATA <= PTR_ADDR_DATA;
                    EXT_PTR_EN <= '0';
                    EXT_GET_STORE_CTL <= ptr_addr_ctl;

                    nextstate <= AVALON_PTR_ADDR_CAPTURE;

                WHEN AVALON_PTR_ADDR_CAPTURE =>

                    AV_WAITREQUEST <= '0';
                    AV_READDATAVALID <= '1';
                    --AV_READ_DATA <= sdram_addr_data;
--                  AV_READ_DATA <= SDRAM_AVALON_ADDR;
                    AV_READ_DATA <= PTR_ADDR_DATA;
                    EXT_PTR_EN <= '0';
                    EXT_GET_STORE_CTL <= ptr_addr_ctl;

                    nextstate <= INIT;

                WHEN CAPTURE_CTL_DATA =>

                    AV_WAITREQUEST <= '0';
                    AV_READDATAVALID <= '0';
                    AV_READ_DATA <= (others => 'Z');
                    EXT_PTR_EN <= '0';
                    EXT_GET_STORE_CTL <= '1';  -- just a temporary assignment; value is obtained
                                -- via av_write_data()

                    ptr_addr_ctl <= AV_WRITE_DATA(0);

                    nextstate <= INIT;

            END CASE;
        END PROCESS rw_control;

    END ARCHITECTURE main_rtl;
```

E.8 Free Memory Address Counter.VHD

This module generates and keeps track of the read and write address pointers for the free memory space available for packet storage. This depends on the size of the SRAM. A 16 bit counter will be used since the memory data width of the SRAM is 32 bits of which 16 bits is used for queue pointer address, and the other 16 bits is used for the actual data addresss. Therefore the address counter will be limited to 16 bits only. Since the SRAM has a 18 bit address size and since 16 bits is used up for pointers and packet addressing, then the other segment of SRAM can be used for head/tail pointer locations.
This calculates as follows:
2^16 = 65536 (# of pointers)
2^16 = 65536 (# of free space)
Total = 131,072 of address space consumed for both of these functions SRAM has a total of 2^18 = 262,144 locations. Removing the above yields 131,072 (17 bits worth) for just head/tail pointers. Therefore to make this simple in terms

of division of boundaries,
START_MEM_ADDR -> location to the start of the free space location in SRAM
END_MEM_ADDR -> location to the end of the free space location in SRAM. It must be less than NUM_OF_PACKETS by 1.
GET_STORE_CTL -> '1' represents getting an address for writing a packet into SDRAM and '0' representso storing a used address from SRAM when reading from SDRAM

Typically num_of_packets = 131072 (17 bits) for the SRAM
NOTE: To use this properly, it assumes that the fetching of an address (i.e. get_addr) takes priority when used in a higher level state machine. It assumes that the SRAM pointers are initially at (0,0) so that you want to be able to write to the SDRAM as the first thing and then follow that with a read operation.

```
-- Library Clause
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.std_logic_unsigned.ALL;

-- Entity Declaration
ENTITY free_memory_addr_counter IS
    GENERIC (NUM_OF_BITS : POSITIVE :=32);
    PORT(
            CLK                 :IN STD_LOGIC;
            RST_N               :IN STD_LOGIC;
            PTR_EN              :IN STD_LOGIC;
            GET_STORE_CTL       :IN STD_LOGIC;

            EMPTY_FLAG          :OUT STD_LOGIC;
            FULL_FLAG           :OUT STD_LOGIC;
            GET_ADDR_PTR        :OUT  STD_LOGIC_VECTOR (NUM_OF_BITS-1 DOWNTO 0);
            STORE_ADDR_PTR      :OUT  STD_LOGIC_VECTOR (NUM_OF_BITS-1 DOWNTO 0);
            OUT_ADDR            :OUT  STD_LOGIC_VECTOR (NUM_OF_BITS-1 DOWNTO 0)
    );
END ENTITY free_memory_addr_counter;

ARCHITECTURE main_cntr_rtl OF free_memory_addr_counter IS

    constant START_MEM_ADDR:std_logic_vector(NUM_OF_BITS-1 DOWNTO 0) := x"01010000";
    constant END_MEM_ADDR:std_logic_vector(NUM_OF_BITS-1 DOWNTO 0) := x"0101FFFC";

BEGIN

    PROCESS (CLK, RST_N)
            -- Define parameters
            variable full_status    :std_logic;
            variable empty_status :std_logic;
            variable no_read        :std_logic;
            variable get_addr       :std_logic_vector(NUM_OF_BITS-1 DOWNTO 0);
            variable store_addr     :std_logic_vector (NUM_OF_BITS-1 DOWNTO 0);
            variable updated_addr:std_logic_vector (NUM_OF_BITS-1 DOWNTO 0);
            variable returned_addr:std_logic_vector (NUM_OF_BITS-1 DOWNTO 0);
    BEGIN
            IF (RST_N = '0') THEN
```

```
--              get_addr := (START_MEM_ADDR-4);
                get_addr := START_MEM_ADDR;-- start the pointer for idle locations at the second address
                                -- not the first, since this is taken care of in the Master peripheral
                                -- for writing the first packet.
                store_addr := (START_MEM_ADDR-4);
                updated_addr := (others => '0');
                returned_addr := START_MEM_ADDR;
                full_status := '0';
                empty_status := '1';
                no_read := '1';     -- guarantees a get operation is done before a store operation
                EMPTY_FLAG <= '1';
                FULL_FLAG <= '0';
                OUT_ADDR <= START_MEM_ADDR;
--              GET_ADDR_PTR <= (START_MEM_ADDR-4);
                GET_ADDR_PTR <= START_MEM_ADDR;
                STORE_ADDR_PTR <= (START_MEM_ADDR-4);

        ELSIF (CLK'EVENT AND CLK = '1') THEN

                IF PTR_EN = '1' THEN

                        IF (GET_STORE_CTL = '1') THEN-- retrieve an address

                            IF (full_status = '0') THEN
                                IF (get_addr /= END_MEM_ADDR) THEN
                                    get_addr := get_addr + 4;
                                    updated_addr := get_addr+4;
                                ELSE
                                    get_addr := START_MEM_ADDR;
                                    updated_addr := get_addr;
                                END IF;
                            ELSE
                                get_addr := get_addr;
                                updated_addr := get_addr;
                            END IF;
                            store_addr := store_addr;
                            returned_addr := get_addr;
                            no_read := '0';

                            -- after incrementing check to make sure you did not catch-up
                            -- to the other address counter.  If you did then this means
                            -- you do not have any more address spaces from SRAM to retrieve
                            IF updated_addr = store_addr THEN
                                full_status := '1';
                            ELSE
                                full_status := '0';
                            END IF;
                            empty_status := '1';


                        ELSE        -- return an address to memory
```

```
                    IF (empty_status = '1' and no_read = '0') THEN
                        IF (store_addr /= END_MEM_ADDR) THEN
                            store_addr := store_addr + 4;
                            updated_addr := store_addr+4;
                            no_read := '0';
                        ELSE
                            store_addr := START_MEM_ADDR;
                            updated_addr := store_addr;
                            no_read := '1';
                        END IF;
                    ELSE
                        store_addr := store_addr;
                        updated_addr := store_addr;
                        no_read := no_read;
                    END IF;
                    get_addr := get_addr;
                    returned_addr := store_addr;

                    -- after incrementing check to make sure you did not catch-up
                    -- to the other address counter.  If you did then this means
                    -- you do not have any more address spaces from SRAM to retrieve
                    IF updated_addr = get_addr THEN
                        empty_status := '0';
                    ELSE
                        empty_status := '1';
                    END IF;
                    full_status := '0';

                END IF;

            ELSE

                get_addr := get_addr;
                store_addr := store_addr;
                full_status := full_status;
                empty_status := empty_status;
                updated_addr := updated_addr;
                returned_addr := returned_addr;

            END IF;

            EMPTY_FLAG <= empty_status;
            FULL_FLAG <= full_status;
            OUT_ADDR <= returned_addr;
            GET_ADDR_PTR <= get_addr;
            STORE_ADDR_PTR <= store_addr;

        END IF;
    END PROCESS;
END ARCHITECTURE main_cntr_rtl;
```

E.9 Master Pre FIFO Write.VHD

This module is used in communicating with the Avalon Bus, the input Prestore and the output Prefetch FIFOs for one VOSQ for a particular sector. As long as there are packets within the prestore FIFO (PRE_FIFO_EMPTY = 0) and the prefetch FIFO is not full (POST_FIFO_FULL = 0), the module is designed to continue to perform read/write requests to the SDRAM controller. Addressing the SDRAM is obtained by first polling the Memory Address Generator peripheral. When writing to SDRAM, a 100 us delay is experienced at startup. In this case the SDRAM controller asserts the waitrequest on the Avalon Bus. This assertion happens 2 clock cycles after the WRITE is invoked. A delay timer is needed to hold-off checking the Avalon waitrequest signal when the WRITE is asserted.

```
-- Library Clause
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;


-- Entity Declaration
ENTITY Master_Pre_FIFO_Write IS
    PORT(
            CLK                     : IN   STD_LOGIC;
            RESET                   : IN STD_LOGIC;
            PRE_FIFO_EMPTY          : IN STD_LOGIC;-- Empty status flag from Prestore FIFO (active high)
            PRE_FIFO_AEMPTY         : IN STD_LOGIC;-- Almost Empty status flag from Prestore FIFO(active
high)
            POST_FIFO_FULL          : IN STD_LOGIC;-- Full status flag from Prefetch FIFO (active high)
            AV_WAITREQUEST          : IN STD_LOGIC; -- Allows Master to wait until data is available from
Avalon Bus (active high)
            AV_READDATAVALID: INSTD_LOGIC;-- Read data from Avalon Bus only when this is VALID
(active high)
            END_OF_PACKET           : IN STD_LOGIC;-- Used to flag when stream of packets is terminated
(active high)
            DELAY_FLAG              : IN STD_LOGIC;-- Status Flag to indicate the delay time has been reached.
            SRAM_WR_RQT_DELAY_FLAG  : INSTD_LOGIC;-- Status Flag to indicate the hold off time before
a waitrequest signal gets asserted from the SRAM for a WRITE request
            SRAM_RD_RQT_DELAY_FLAG  : INSTD_LOGIC;-- Status Flag to indicate the hold off time before
a waitrequest signal gets asserted from the SRAM for a READ request
            SRAM_RD_WR_DELAY_FLAG: INSTD_LOGIC;-- Status Flag to indicate the amount of delay
needed between a back-to-back read or write request from SRAM
            IN_DATA                 : IN STD_LOGIC_VECTOR (15 DOWNTO 0);-- Input data from Prestore
FIFO
            AV_READ_DATA            : IN STD_LOGIC_VECTOR (31 DOWNTO 0);-- Avalon Read Data Bus
Width
            NEXT_ADDR               : IN STD_LOGIC_VECTOR (31 DOWNTO 0);-- SDRAM next address
location for packet storage
            SRAM_IDLE_AREA_ADDR_DATA: INSTD_LOGIC_VECTOR (31 DOWNTO 0);-- SRAM idle
area address value
            SRAM_QUEUE_AREA_ADDR_DATA: INSTD_LOGIC_VECTOR (31 DOWNTO 0);-- SRAM
queue area address value
            SRAM_IDLE_AREA_ADDR: INSTD_LOGIC_VECTOR (31 DOWNTO 0);-- SRAM idle area
address
            SRAM_QUEUE_AREA_ADDR: INSTD_LOGIC_VECTOR (31 DOWNTO 0);-- SRAM queue area
address
            MASTER_ID                   : INSTD_LOGIC;-- Identifies whether this is the 'primary' master
```

```
        MEM_INIT_COMPL      : IN STD_LOGIC;-- Flag to represent when the 'primary' master is complete
with the SRAM address map initialization.
        SRAM_IDLE_AREA_DONE: INSTD_LOGIC;-- Flag to indicate when the idle address space has been
completed initialized with address values
        SRAM_QUEUE_AREA_DONE: INSTD_LOGIC;-- Flag to indicate when the queue address space has
been completed initialized with address values
        MASTER_MEM_INIT_DONE: OUTSTD_LOGIC;-- Flag to represent the completion of the SRAM
address map initialization
        AV_MEM_ADDR           : OUTSTD_LOGIC_VECTOR (31 DOWNTO 0);-- Memory address
        AV_WRITE_N          : OUTSTD_LOGIC;  -- Write enable bit to the Avalon Bus (active low)
        AV_WRITE_DATA       : OUTSTD_LOGIC_VECTOR (31 DOWNTO 0);-- Avalon Write Data Bus
Width
        AV_READ_N           : OUTSTD_LOGIC; -- Write enable bit to the Avalon Bus (active low)
        AV_FLUSH            : OUTSTD_LOGIC; -- Flush Avalon Bus of all pending data (active high)
        PRE_FIFO_RDREQ      : OUTSTD_LOGIC; -- Prestore FIFO Read Request (active high)
        POST_FIFO_WRREQ     : OUTSTD_LOGIC; -- Prefetch FIFO Write Request (active high)
        PACKET_CNT_EN       : OUTSTD_LOGIC;-- Streaming data counter enable (active high)
        PACKET_CNT_RST      : OUTSTD_LOGIC;-- Streaming data counter reset (active low)
        ADDR_LOAD           : OUTSTD_LOGIC; -- Preload for address counter in case packet size is large
and needs to be broken up (active high)
        DELAY_TIMER_EN      : OUT STD_LOGIC;-- Delay timer enable for WRITE cycle data capture for
SDRAM (active high)
        SRAM_WR_RQT_TIMER_EN: OUT STD_LOGIC;-- Delay timer enable for the SRAM write
waitrequest assertion(active high)
        SRAM_RD_RQT_TIMER_EN: OUT STD_LOGIC;-- Delay timer enable for the SRAM read
waitrequest assertion(active high)
        SRAM_RD_WR_TIMER_EN: OUT STD_LOGIC;-- Delay timer enable for the delay needed between
a SRAM read or a write(active high)
        SRAM_IDLE_AREA_INIT_EN: OUT STD_LOGIC;-- SRAM Address mapping initialization for the
idle address space (active high)
        SRAM_QUEUE_AREA_INIT_EN: OUT STD_LOGIC;-- SRAM Address mapping initialization for
the queue address space; this contains SDRAM address data(active high)
        SRAM_IDLE_AREA_INIT_RSTN: OUT STD_LOGIC;-- SRAM Address mapping initialization reset
signal for the idle address space (active low)
        SRAM_QUEUE_AREA_INIT_RSTN: OUT STD_LOGIC;-- SRAM Address mapping initialization
reset signal for the queue address space; this contains SDRAM address data(active low)
        Q_LGTH_CNT_EN       : OUTSTD_LOGIC;-- Queue length counter enable
        Q_LGTH_CTL          : OUTSTD_LOGIC;-- Queue length insert/remove control ('1'= insert; '0' =
remove)

        OUT_DATA            : OUTSTD_LOGIC_VECTOR (15 DOWNTO 0);-- Output data to the
Prefetch FIFO
        ADDR_LOAD_DATA      : OUTSTD_LOGIC_VECTOR (31 DOWNTO 0)-- Loaded Burst Address
Counter
    );

END ENTITY Master_Pre_FIFO_Write;

-- Architecture Body
ARCHITECTURE main_rtl OF Master_Pre_FIFO_Write IS

-- State Machine Declaration
```

```
--TYPE EXCHANGE_STATEIS
(INIT, REQ_EMPTY_ADDR, RQT_FOR_VOSQ_TAIL_ADDR, SET_IDLE_PTR_ADDR_CTL,
RQT_IDLE_PTR_ADDR, CAPTURE_AV_READ_IDLE_PTR_ADDR,
RQT_AND_EXTRACT_IDLE_PTR_ADDR, RQT_PREV_PACKET_PTR_ADDR,
UPDATE_PREV_PACKET_PTR_ADDR, UPDATE_VOSQ_TAIL_PTR_ADDR, SRAM_WRITE_RQT,
SRAM_WAIT_1, SRAM_READ_RQT, SRAM_WAIT_2, SRAM_WRITE_RQT2, SRAM_WAIT_3,
SRAM_READ_RQT2, SRAM_WAIT_4, REQ_FOR_ADDR, CAPTURE_AV_READ, PRELOAD_ADDR_CNTR,
PRELOAD_ADDR_CNTR_READ_DELAY, READ_RQT_FIFO, WRITE_SDRAM_DATA,
INCREMENT_ADDR, READ_SDRAM_DATA, CAPTURE_SDRAM_AV_READ, WRITE_TO_OUT_FIFO);

TYPEEXCHANGE_STATEIS(STARTUP_COND, ADDR_IDLE_MAP_WRITE_INIT,
ADDR_IDLE_MAP_WRITE_RST, ADDR_IDLE_MAP_READ_INIT,
ADDR_IDLE_MAP_AVALON_READ_INIT, ADDR_QUEUE_MAP_WRITE_INIT,
ADDR_QUEUE_MAP_WRITE_RST, ADDR_QUEUE_MAP_READ_INIT,
ADDR_QUEUE_MAP_AVALON_READ_INIT, TAIL_ADDR_MAP_WRITE_INIT,
TAIL_ADDR_MAP_READ_INIT, TAIL_ADDR_MAP_AVALON_READ_INIT,
HEAD_ADDR_MAP_WRITE_INIT, HEAD_ADDR_MAP_READ_INIT,
HEAD_ADDR_MAP_AVALON_READ_INIT, INIT, RQT_FOR_VOSQ_HEAD_ADDR,
CAPTURE_AV_READ_RQT_FOR_VOSQ_HEAD_ADDR, CAPTURE_AV_READ_VOSQ_TAIL_ADDR,
UPDATE_IDLE_PTR_ADDR_IN_EMPTY_LOC, RQT_PREV_PACKET_AT_HEAD_ADDR,
CAPTURE_AV_RQT_PREV_PACKET_AT_HEAD_ADDR, UPDATE_VOSQ_HEAD_PTR_ADDR,
RQT_FOR_VOSQ_TAIL_ADDR, SET_IDLE_PTR_ADDR_CTL, DELAY_RQT_IDLE_PTR_ADDR,
RQT_IDLE_PTR_ADDR, CAPTURE_IDLE_PTR_ADDR, RQT_AND_EXTRACT_IDLE_PTR_ADDR,
CAPTURE_AV_READ_IDLE_PTR_ADDR, RQT_PREV_PACKET_PTR_ADDR,
CAPTURE_AV_RQT_PREV_PKT_PTR_ADDR, UPDATE_PREV_PACKET_PTR_ADDR,
PREV_PACKET_UPDATE_TAIL_WRITE_RST, UPDATE_VOSQ_TAIL_PTR_ADDR,
RQT_SDRAM_IDLE_ADDR, CAPTURE_AV_RQT_SDRAM_IDLE_ADDR, PRELOAD_ADDR_CNTR,
PRELOAD_ADDR_CNTR_READ_DELAY, READ_RQT_FIFO, WRITE_SDRAM_DATA,
INCREMENT_ADDR,READ_SDRAM_DATA, CAPTURE_SDRAM_AV_READ, WRITE_TO_OUT_FIFO);

SIGNAL state, nextstate :EXCHANGE_STATE;

SIGNAL rd_wr_op, next_op:STD_LOGIC;  -- required for round-robin arbitration; '0' = write, '1' = read
SIGNAL first_pkt_flag:STD_LOGIC;  -- required to simplify the writing of the first packet

SIGNAL sdram_addr_data :STD_LOGIC_VECTOR (31 DOWNTO 0);
SIGNAL out_bus_packet_data:STD_LOGIC_VECTOR (31 DOWNTO 0);
SIGNAL prev_packet_addr:STD_LOGIC_VECTOR (31 DOWNTO 0);
SIGNAL head_tail_addr:STD_LOGIC_VECTOR (31 DOWNTO 0);
SIGNAL empty_avail_addr:STD_LOGIC_VECTOR (31 DOWNTO 0);
SIGNAL get_pointer_addr:STD_LOGIC_VECTOR (31 DOWNTO 0);


CONSTANT SRAM_AVALON_BUS_OFFSET:STD_LOGIC_VECTOR (15 DOWNTO 0) := x"0100";-- obtained
from SOPC builder
CONSTANT SDRAM_CTRL_ADDR:STD_LOGIC_VECTOR (31 DOWNTO 0) := x"00000000";
CONSTANT SRAM_CTRL_VOSQ_TAIL_ADDR:STD_LOGIC_VECTOR (31 DOWNTO 0) := x"01020004";
CONSTANT SRAM_CTRL_VOSQ_HEAD_ADDR:STD_LOGIC_VECTOR (31 DOWNTO 0) := x"01020000";
CONSTANT IDLE_ADDR_CNTR_SLAVE_ADDR:STD_LOGIC_VECTOR (31 DOWNTO 0) := x"01400000";


BEGIN
```

```
state_register: PROCESS (CLK,  RESET)
BEGIN
        -- Trigger on the positive edge of the clock.  Synchronous operation.
        -- Reset asynchronously to the initialization state if RESET is pulled high
        IF (RESET = '1') THEN

                state <= STARTUP_COND;
                rd_wr_op <= '0';  -- write takes precedence

        ELSIF (CLK'EVENT AND CLK = '1') THEN
                state <= nextstate;
                rd_wr_op <= next_op;

        END IF;
END PROCESS state_register;

-- all input variables go into sensitivity list as this represents the combinatorial
-- logic
rw_control:  PROCESS  (state,  PRE_FIFO_EMPTY,PRE_FIFO_AEMPTY,  POST_FIFO_FULL,
AV_WAITREQUEST,  AV_READDATAVALID,AV_READ_DATA,IN_DATA,  NEXT_ADDR,
END_OF_PACKET,  DELAY_FLAG,  SRAM_WR_RQT_DELAY_FLAG,  SRAM_RD_RQT_DELAY_FLAG,
SRAM_RD_WR_DELAY_FLAG,  sdram_addr_data,  out_bus_packet_data,  rd_wr_op,  prev_packet_addr,
head_tail_addr,  empty_avail_addr,  get_pointer_addr,  first_pkt_flag,  MEM_INIT_COMPL,  MASTER_ID,
SRAM_IDLE_AREA_ADDR_DATA,  SRAM_QUEUE_AREA_ADDR_DATA,  SRAM_IDLE_AREA_ADDR,
SRAM_QUEUE_AREA_ADDR, SRAM_IDLE_AREA_DONE, SRAM_QUEUE_AREA_DONE)
        BEGIN
        CASE state IS

                WHEN STARTUP_COND =>

                        AV_MEM_ADDR <= (others => 'Z');
                        sdram_addr_data <= (others => 'Z');
                        AV_WRITE_DATA <= (others => 'Z');
                        ADDR_LOAD_DATA <= (others => 'Z');
                        OUT_DATA <= (others => 'Z');

                        prev_packet_addr <= (others => 'Z');
                        head_tail_addr <= (others => 'Z');
                        empty_avail_addr <= (others => 'Z');
                        get_pointer_addr <= (others => 'Z');
                        first_pkt_flag <= '1';

                        SRAM_IDLE_AREA_INIT_EN <= '0';-- sram idle area initialization address
generator disabled
                        SRAM_QUEUE_AREA_INIT_EN <= '0';-- sram queue area initialization address
generator disabled
                        SRAM_IDLE_AREA_INIT_RSTN <= '0';-- sram idle area initialization address
generator reset disabled
                        SRAM_QUEUE_AREA_INIT_RSTN <= '0';-- sram queue area initialization address
generator reset disabled
                        SRAM_RD_WR_TIMER_EN <= '0';-- sram delta delay between read or a write
```

```
disabled
                    SRAM_WR_RQT_TIMER_EN <= '0';-- sram write delay disabled
                    SRAM_RD_RQT_TIMER_EN <= '0';-- sram read delay disabled
                    DELAY_TIMER_EN <= '0';-- sdram write delay disabled
                    Q_LGTH_CNT_EN <= '0';
                    Q_LGTH_CTL <= '1';
                    AV_READ_N <= '1';
                    AV_WRITE_N <= '1';
                    PRE_FIFO_RDREQ <= '0';
                    POST_FIFO_WRREQ <= '0';
                    AV_FLUSH <= '1';
                    PACKET_CNT_EN <= '0';
                    PACKET_CNT_RST <= '0';
                    ADDR_LOAD <= '0';
                    MASTER_MEM_INIT_DONE <= '0';
                    out_bus_packet_data <= (others => '0');

                    IF MASTER_ID = '1' THEN
                        nextstate <= ADDR_IDLE_MAP_WRITE_INIT;
                    ELSE
                        nextstate <= INIT;
                    END IF;


                WHEN ADDR_IDLE_MAP_WRITE_INIT =>

                    AV_MEM_ADDR <= SRAM_IDLE_AREA_ADDR;
                    sdram_addr_data <= (others => 'Z');
                    AV_WRITE_DATA <= SRAM_IDLE_AREA_ADDR_DATA;
                    ADDR_LOAD_DATA <= (others => 'Z');
                    OUT_DATA <= (others => 'Z');

                    prev_packet_addr <= (others => 'Z');
                    head_tail_addr <= (others => 'Z');
                    empty_avail_addr <= (others => 'Z');
                    get_pointer_addr <= (others => 'Z');
                    first_pkt_flag <= '1';

                    SRAM_QUEUE_AREA_INIT_EN <= '0';-- sram queue area initialization address
generator disabled
                    SRAM_QUEUE_AREA_INIT_RSTN <= '0';-- sram queue area initialization address
generator reset disabled
                    SRAM_RD_WR_TIMER_EN <= '0';-- sram delta delay between read or a write
disabled
                    SRAM_WR_RQT_TIMER_EN <= '1';-- sram write delay disabled
                    SRAM_RD_RQT_TIMER_EN <= '0';-- sram read delay disabled
                    DELAY_TIMER_EN <= '0';-- sdram write delay disabled
                    Q_LGTH_CNT_EN <= '0';
                    Q_LGTH_CTL <= '1';
                    AV_READ_N <= '1';
                    AV_WRITE_N <= '0';
                    PRE_FIFO_RDREQ <= '0';
```

```
                POST_FIFO_WRREQ <= '0';
                AV_FLUSH <= '0';
                PACKET_CNT_EN <= '0';
                PACKET_CNT_RST <= '0';
                ADDR_LOAD <= '0';
                MASTER_MEM_INIT_DONE <= '0';
                out_bus_packet_data <= (others => '0');
                next_op <= '0';  -- write takes precedence


                IF SRAM_WR_RQT_DELAY_FLAG = '1' THEN
                    IF AV_WAITREQUEST = '1' THEN
                        SRAM_IDLE_AREA_INIT_EN <= '0';-- sram idle area initialization
address generator
                        nextstate <= ADDR_IDLE_MAP_WRITE_INIT;
                    ELSE
                        IF (SRAM_IDLE_AREA_DONE = '1') THEN
                            SRAM_IDLE_AREA_INIT_EN <= '0';
                            nextstate <= ADDR_IDLE_MAP_WRITE_RST;
                        ELSE
                            SRAM_IDLE_AREA_INIT_EN <= '1';
                            nextstate <= ADDR_IDLE_MAP_WRITE_INIT;
                        END IF;
                    END IF;
                    SRAM_IDLE_AREA_INIT_RSTN <= '1';-- sram idle area initialization address
generator reset
                ELSE
                    SRAM_IDLE_AREA_INIT_RSTN <= '0';
                    SRAM_IDLE_AREA_INIT_EN <= '0';
                    nextstate <= ADDR_IDLE_MAP_WRITE_INIT;
                END IF;

            WHEN ADDR_IDLE_MAP_WRITE_RST =>

                AV_MEM_ADDR <= SRAM_IDLE_AREA_ADDR;
                sdram_addr_data <= (others => 'Z');
                AV_WRITE_DATA <= SRAM_IDLE_AREA_ADDR_DATA;
                ADDR_LOAD_DATA <= (others => 'Z');
                OUT_DATA <= (others => 'Z');

                prev_packet_addr <= (others => 'Z');
                head_tail_addr <= (others => 'Z');
                empty_avail_addr <= (others => 'Z');
                get_pointer_addr <= (others => 'Z');
                first_pkt_flag <= '1';

                SRAM_IDLE_AREA_INIT_RSTN <= '0';
                SRAM_IDLE_AREA_INIT_EN <= '0';
                SRAM_QUEUE_AREA_INIT_EN <= '0';-- sram queue area initialization address
generator disabled
                SRAM_QUEUE_AREA_INIT_RSTN <= '0';-- sram queue area initialization address
generator reset disabled
```

```
                SRAM_RD_WR_TIMER_EN <= '0';-- sram delta delay between read or a write
disabled
                SRAM_WR_RQT_TIMER_EN <= '0';-- sram write delay disabled
                SRAM_RD_RQT_TIMER_EN <= '0';-- sram read delay disabled
                DELAY_TIMER_EN <= '0';-- sdram write delay disabled
                Q_LGTH_CNT_EN <= '0';
                Q_LGTH_CTL <= '1';
                AV_READ_N <= '1';
                AV_WRITE_N <= '1';
                PRE_FIFO_RDREQ <= '0';
                POST_FIFO_WRREQ <= '0';
                AV_FLUSH <= '0';
                PACKET_CNT_EN <= '0';
                PACKET_CNT_RST <= '0';
                ADDR_LOAD <= '0';
                MASTER_MEM_INIT_DONE <= '0';
                out_bus_packet_data <= (others => '0');
                next_op <= '0';  -- write takes precedence

                nextstate <= ADDR_IDLE_MAP_READ_INIT;

        WHEN ADDR_IDLE_MAP_READ_INIT =>

                AV_MEM_ADDR <= SRAM_IDLE_AREA_ADDR;
                AV_WRITE_DATA <= (others => 'Z');
                ADDR_LOAD_DATA <= (others => 'Z');
                OUT_DATA <= (others => 'Z');

                sdram_addr_data <= (others => 'Z');
                prev_packet_addr <= (others => 'Z');
                head_tail_addr <= (others => 'Z');
                empty_avail_addr <= (others => 'Z');
                get_pointer_addr <= (others => 'Z');
                first_pkt_flag <= '1';

                SRAM_IDLE_AREA_INIT_RSTN <= '1';
                SRAM_IDLE_AREA_INIT_EN <= '0';-- sram idle area initialization address
generator disable
                SRAM_QUEUE_AREA_INIT_EN <= '0';-- sram queue area initialization address
generator disable
                SRAM_QUEUE_AREA_INIT_RSTN <= '0';-- sram queue area initialization address
generator reset disabled
                SRAM_RD_WR_TIMER_EN <= '0';-- sram delta delay between read or a write
disabled
                SRAM_WR_RQT_TIMER_EN <= '0';-- sram write delay disabled
                SRAM_RD_RQT_TIMER_EN <= '1';-- sram read delay disabled
                DELAY_TIMER_EN <= '0';-- sdram write delay disabled
                Q_LGTH_CNT_EN <= '0';
                Q_LGTH_CTL <= '1';
                AV_READ_N <= '0';
                AV_WRITE_N <= '1';
                PRE_FIFO_RDREQ <= '0';
```

```
                POST_FIFO_WRREQ <= '0';
                AV_FLUSH <= '0';
                PACKET_CNT_EN <= '0';
                PACKET_CNT_RST <= '0';
                ADDR_LOAD <= '0';
                MASTER_MEM_INIT_DONE <= '0';
                out_bus_packet_data <= (others => '0');


                next_op <= '0';  -- write takes precedence

                IF SRAM_RD_RQT_DELAY_FLAG = '1' THEN
                    IF AV_WAITREQUEST = '1' THEN
                        nextstate <= ADDR_IDLE_MAP_AVALON_READ_INIT;
                    ELSE
                        nextstate <= ADDR_IDLE_MAP_READ_INIT;
                    END IF;
                ELSE
                    nextstate <= ADDR_IDLE_MAP_READ_INIT;
                END IF;

            WHEN ADDR_IDLE_MAP_AVALON_READ_INIT =>

                AV_MEM_ADDR <= SRAM_IDLE_AREA_ADDR;
                AV_WRITE_DATA <= (others => 'Z');
                ADDR_LOAD_DATA <= (others => 'Z');
                OUT_DATA <= (others => 'Z');

                sdram_addr_data <= AV_READ_DATA;
                prev_packet_addr <= (others => 'Z');
                head_tail_addr <= (others => 'Z');
                empty_avail_addr <= (others => 'Z');
                get_pointer_addr <= (others => 'Z');
                first_pkt_flag <= '1';

                SRAM_IDLE_AREA_INIT_RSTN <= '1';
                SRAM_IDLE_AREA_INIT_EN <= '1';-- sram idle area initialization address
generator disable
                SRAM_QUEUE_AREA_INIT_EN <= '0';-- sram queue area initialization address
generator disable
                SRAM_QUEUE_AREA_INIT_RSTN <= '0';-- sram queue area initialization address
generator reset disabled
                SRAM_RD_WR_TIMER_EN <= '0';-- sram delta delay between read or a write
disabled
                SRAM_WR_RQT_TIMER_EN <= '0';-- sram write delay disabled
                SRAM_RD_RQT_TIMER_EN <= '0';-- sram read delay disabled
                DELAY_TIMER_EN <= '0';-- sdram write delay disabled
                Q_LGTH_CNT_EN <= '0';
                Q_LGTH_CTL <= '1';
                AV_READ_N <= '0';
                AV_WRITE_N <= '1';
                PRE_FIFO_RDREQ <= '0';
```

```
                POST_FIFO_WRREQ <= '0';
                AV_FLUSH <= '0';
                PACKET_CNT_EN <= '0';
                PACKET_CNT_RST <= '0';
                ADDR_LOAD <= '0';
                MASTER_MEM_INIT_DONE <= '0';
                out_bus_packet_data <= (others => '0');

                next_op <= '0';  -- write takes precedence
                IF (SRAM_IDLE_AREA_DONE = '1') THEN
                    nextstate <= ADDR_QUEUE_MAP_WRITE_INIT;
                ELSE
                    nextstate <= ADDR_IDLE_MAP_READ_INIT;
                END IF;


            WHEN ADDR_QUEUE_MAP_WRITE_INIT =>

                AV_MEM_ADDR <= SRAM_QUEUE_AREA_ADDR;
                sdram_addr_data <= (others => 'Z');
                AV_WRITE_DATA <= SRAM_QUEUE_AREA_ADDR_DATA;
                ADDR_LOAD_DATA <= (others => 'Z');
                OUT_DATA <= (others => 'Z');

                prev_packet_addr <= (others => 'Z');
                head_tail_addr <= (others => 'Z');
                empty_avail_addr <= (others => 'Z');
                get_pointer_addr <= (others => 'Z');
                first_pkt_flag <= '1';

                SRAM_IDLE_AREA_INIT_EN <= '0';--  sram  idle  area  initialization  address
generator disable
                SRAM_IDLE_AREA_INIT_RSTN <= '0';--  sram  idle  area  initialization  address
generator reset disabled
                SRAM_RD_WR_TIMER_EN <= '0';-- sram delta delay between read or a write
disabled
                SRAM_WR_RQT_TIMER_EN <= '1';-- sram write delay disabled
                SRAM_RD_RQT_TIMER_EN <= '0';-- sram read delay disabled
                DELAY_TIMER_EN <= '0';-- sdram write delay disabled
                Q_LGTH_CNT_EN <= '0';
                Q_LGTH_CTL <= '1';
                AV_READ_N <= '1';
                AV_WRITE_N <= '0';
                PRE_FIFO_RDREQ <= '0';
                POST_FIFO_WRREQ <= '0';
                AV_FLUSH <= '0';
                PACKET_CNT_EN <= '0';
                PACKET_CNT_RST <= '0';
                ADDR_LOAD <= '0';
                MASTER_MEM_INIT_DONE <= '0';
                out_bus_packet_data <= (others => '0');
                next_op <= '0';  -- write takes precedence
```

```
                    IF SRAM_WR_RQT_DELAY_FLAG = '1' THEN
                        IF AV_WAITREQUEST = '1' THEN
                            SRAM_QUEUE_AREA_INIT_EN <= '0';-- sram queue area initialization
address generator
                            nextstate <= ADDR_QUEUE_MAP_WRITE_INIT;
                        ELSE
                            IF (SRAM_QUEUE_AREA_DONE = '1') THEN
                                SRAM_QUEUE_AREA_INIT_EN <= '0';
                                nextstate <= ADDR_QUEUE_MAP_WRITE_RST;
                            ELSE
                                SRAM_QUEUE_AREA_INIT_EN <= '1';
                                nextstate <= ADDR_QUEUE_MAP_WRITE_INIT;
                            END IF;
                        END IF;
                        SRAM_QUEUE_AREA_INIT_RSTN <= '1';-- sram queue area initialization
address generator reset
                    ELSE
                        SRAM_QUEUE_AREA_INIT_RSTN <= '0';
                        SRAM_QUEUE_AREA_INIT_EN <= '0';
                        nextstate <= ADDR_QUEUE_MAP_WRITE_INIT;
                    END IF;


                WHEN ADDR_QUEUE_MAP_WRITE_RST =>

                    AV_MEM_ADDR <= SRAM_QUEUE_AREA_ADDR;
                    sdram_addr_data <= (others => 'Z');
                    AV_WRITE_DATA <= SRAM_QUEUE_AREA_ADDR_DATA;
                    ADDR_LOAD_DATA <= (others => 'Z');
                    OUT_DATA <= (others => 'Z');

                    prev_packet_addr <= (others => 'Z');
                    head_tail_addr <= (others => 'Z');
                    empty_avail_addr <= (others => 'Z');
                    get_pointer_addr <= (others => 'Z');
                    first_pkt_flag <= '1';

                    SRAM_IDLE_AREA_INIT_RSTN <= '0';
                    SRAM_IDLE_AREA_INIT_EN <= '0';
                    SRAM_QUEUE_AREA_INIT_EN <= '0';-- sram queue area initialization address
generator disabled
                    SRAM_QUEUE_AREA_INIT_RSTN <= '0';-- sram queue area initialization address
generator reset disabled
                    SRAM_RD_WR_TIMER_EN <= '0';-- sram delta delay between read or a write
disabled
                    SRAM_WR_RQT_TIMER_EN <= '0';-- sram write delay disabled
                    SRAM_RD_RQT_TIMER_EN <= '0';-- sram read delay disabled
                    DELAY_TIMER_EN <= '0';-- sdram write delay disabled
                    Q_LGTH_CNT_EN <= '0';
                    Q_LGTH_CTL <= '1';
                    AV_READ_N <= '1';
```

```
                AV_WRITE_N <= '1';
                PRE_FIFO_RDREQ <= '0';
                POST_FIFO_WRREQ <= '0';
                AV_FLUSH <= '0';
                PACKET_CNT_EN <= '0';
                PACKET_CNT_RST <= '0';
                ADDR_LOAD <= '0';
                MASTER_MEM_INIT_DONE <= '0';
                out_bus_packet_data <= (others => '0');
                next_op <= '0';   -- write takes precedence

                nextstate <= ADDR_QUEUE_MAP_READ_INIT;


        WHEN ADDR_QUEUE_MAP_READ_INIT =>

                AV_MEM_ADDR <= SRAM_QUEUE_AREA_ADDR;
                sdram_addr_data <= (others => 'Z');
                AV_WRITE_DATA <= (others => 'Z');
                ADDR_LOAD_DATA <= (others => 'Z');
                OUT_DATA <= (others => 'Z');

                prev_packet_addr <= (others => 'Z');
                head_tail_addr <= (others => 'Z');
                empty_avail_addr <= (others => 'Z');
                get_pointer_addr <= (others => 'Z');
                first_pkt_flag <= '1';

                SRAM_IDLE_AREA_INIT_EN <= '0';-- sram idle area initialization address
generator disable
                SRAM_IDLE_AREA_INIT_RSTN <= '0';-- sram idle area initialization address
generator reset disabled
                SRAM_QUEUE_AREA_INIT_EN <= '0';-- sram queue area initialization address
generator
                SRAM_QUEUE_AREA_INIT_RSTN <= '1';-- sram queue area initialization address
generator reset
                SRAM_RD_WR_TIMER_EN <= '0';-- sram delta delay between read or a write
disabled
                SRAM_WR_RQT_TIMER_EN <= '0';-- sram write delay disabled
                SRAM_RD_RQT_TIMER_EN <= '1';-- sram read delay disabled
                DELAY_TIMER_EN <= '0';-- sdram write delay disabled
                Q_LGTH_CNT_EN <= '0';
                Q_LGTH_CTL <= '1';
                AV_READ_N <= '0';
                AV_WRITE_N <= '1';
                PRE_FIFO_RDREQ <= '0';
                POST_FIFO_WRREQ <= '0';
                AV_FLUSH <= '0';
                PACKET_CNT_EN <= '0';
                PACKET_CNT_RST <= '0';
                ADDR_LOAD <= '0';
                MASTER_MEM_INIT_DONE <= '0';
```

```
                out_bus_packet_data <= (others => '0');

                next_op <= '0';  -- write takes precedence

                IF SRAM_RD_RQT_DELAY_FLAG = '1' THEN
                    IF AV_WAITREQUEST = '1' THEN
                        nextstate <= ADDR_QUEUE_MAP_AVALON_READ_INIT;
                    ELSE
                        nextstate <= ADDR_QUEUE_MAP_READ_INIT;
                    END IF;
                ELSE
                    nextstate <= ADDR_QUEUE_MAP_READ_INIT;
                END IF;

            WHEN ADDR_QUEUE_MAP_AVALON_READ_INIT =>

                AV_MEM_ADDR <= SRAM_QUEUE_AREA_ADDR;
                AV_WRITE_DATA <= (others => 'Z');
                ADDR_LOAD_DATA <= (others => 'Z');
                OUT_DATA <= (others => 'Z');

                sdram_addr_data <= AV_READ_DATA;
                prev_packet_addr <= (others => 'Z');
                head_tail_addr <= (others => 'Z');
                empty_avail_addr <= (others => 'Z');
                get_pointer_addr <= (others => 'Z');
                first_pkt_flag <= '1';

                SRAM_IDLE_AREA_INIT_RSTN <= '0';
                SRAM_IDLE_AREA_INIT_EN <= '0';-- sram idle area initialization address
generator disable
                SRAM_QUEUE_AREA_INIT_EN <= '1';-- sram queue area initialization address
generator disable
                SRAM_QUEUE_AREA_INIT_RSTN <= '1';-- sram queue area initialization address
generator reset disabled
                SRAM_RD_WR_TIMER_EN <= '0';-- sram delta delay between read or a write
disabled
                SRAM_WR_RQT_TIMER_EN <= '0';-- sram write delay disabled
                SRAM_RD_RQT_TIMER_EN <= '0';-- sram read delay disabled
                DELAY_TIMER_EN <= '0';-- sdram write delay disabled
                Q_LGTH_CNT_EN <= '0';
                Q_LGTH_CTL <= '1';
                AV_READ_N <= '0';
                AV_WRITE_N <= '1';
                PRE_FIFO_RDREQ <= '0';
                POST_FIFO_WRREQ <= '0';
                AV_FLUSH <= '0';
                PACKET_CNT_EN <= '0';
                PACKET_CNT_RST <= '0';
                ADDR_LOAD <= '0';
                MASTER_MEM_INIT_DONE <= '0';
                out_bus_packet_data <= (others => '0');
```

```
                next_op <= '0';   -- write takes precedence
                IF (SRAM_QUEUE_AREA_DONE = '1') THEN
                    nextstate <= TAIL_ADDR_MAP_WRITE_INIT;
                ELSE
                    nextstate <= ADDR_QUEUE_MAP_READ_INIT;
                END IF;


            WHEN TAIL_ADDR_MAP_WRITE_INIT =>

                AV_MEM_ADDR <= SRAM_CTRL_VOSQ_TAIL_ADDR;
                sdram_addr_data <= (others => 'Z');
                AV_WRITE_DATA <= SRAM_QUEUE_AREA_ADDR;
                ADDR_LOAD_DATA <= (others => 'Z');
                OUT_DATA <= (others => 'Z');

                prev_packet_addr <= (others => 'Z');
                head_tail_addr <= (others => 'Z');
                empty_avail_addr <= (others => 'Z');
                get_pointer_addr <= (others => 'Z');
                first_pkt_flag <= '1';

                SRAM_QUEUE_AREA_INIT_EN <= '0';-- sram queue area initialization address
generator
                SRAM_QUEUE_AREA_INIT_RSTN <= '0';-- sram queue area initialization address
generator reset
                SRAM_IDLE_AREA_INIT_EN <= '0';-- sram idle area initialization address
generator disable
                SRAM_IDLE_AREA_INIT_RSTN <= '0';-- sram idle area initialization address
generator reset disabled
                SRAM_RD_WR_TIMER_EN <= '0';-- sram delta delay between read or a write
disabled
                SRAM_WR_RQT_TIMER_EN <= '1';-- sram write delay disabled
                SRAM_RD_RQT_TIMER_EN <= '0';-- sram read delay disabled
                DELAY_TIMER_EN <= '0';-- sdram write delay disabled
                Q_LGTH_CNT_EN <= '0';
                Q_LGTH_CTL <= '1';
                AV_READ_N <= '1';
                AV_WRITE_N <= '0';
                PRE_FIFO_RDREQ <= '0';
                POST_FIFO_WRREQ <= '0';
                AV_FLUSH <= '0';
                PACKET_CNT_EN <= '0';
                PACKET_CNT_RST <= '0';
                ADDR_LOAD <= '0';
                MASTER_MEM_INIT_DONE <= '0';
                out_bus_packet_data <= (others => '0');
                next_op <= '0';   -- write takes precedence

                IF SRAM_WR_RQT_DELAY_FLAG = '1' THEN
                    IF AV_WAITREQUEST = '1' THEN
```

```
                nextstate <= TAIL_ADDR_MAP_WRITE_INIT;
            ELSE
                nextstate <= TAIL_ADDR_MAP_READ_INIT;
            END IF;
        ELSE
            nextstate <= TAIL_ADDR_MAP_WRITE_INIT;
        END IF;


    WHEN TAIL_ADDR_MAP_READ_INIT =>

            AV_MEM_ADDR <= SRAM_CTRL_VOSQ_TAIL_ADDR;
            sdram_addr_data <= (others => 'Z');
            AV_WRITE_DATA <= (others => 'Z');
            ADDR_LOAD_DATA <= (others => 'Z');
            OUT_DATA <= (others => 'Z');

            prev_packet_addr <= (others => 'Z');
            head_tail_addr <= (others => 'Z');
            empty_avail_addr <= (others => 'Z');
            get_pointer_addr <= (others => 'Z');
            first_pkt_flag <= '1';

            SRAM_QUEUE_AREA_INIT_RSTN <= '0';-- sram queue area initialization address
generator reset
            SRAM_QUEUE_AREA_INIT_EN <= '0';-- sram queue area initialization address
generator
            SRAM_IDLE_AREA_INIT_EN <= '0';-- sram idle area initialization address
generator disable
            SRAM_IDLE_AREA_INIT_RSTN <= '0';-- sram idle area initialization address
generator reset disabled
            SRAM_RD_WR_TIMER_EN <= '0';-- sram delta delay between read or a write
disabled
            SRAM_WR_RQT_TIMER_EN <= '0';-- sram write delay disabled
            SRAM_RD_RQT_TIMER_EN <= '1';-- sram read delay disabled
            DELAY_TIMER_EN <= '0';-- sdram write delay disabled
            Q_LGTH_CNT_EN <= '0';
            Q_LGTH_CTL <= '1';
            AV_READ_N <= '0';
            AV_WRITE_N <= '1';
            PRE_FIFO_RDREQ <= '0';
            POST_FIFO_WRREQ <= '0';
            AV_FLUSH <= '0';
            PACKET_CNT_EN <= '0';
            PACKET_CNT_RST <= '0';
            ADDR_LOAD <= '0';
            MASTER_MEM_INIT_DONE <= '0';
            out_bus_packet_data <= (others => '0');

            next_op <= '0';  -- write takes precedence
            IF SRAM_RD_RQT_DELAY_FLAG = '1' THEN
                    nextstate <= TAIL_ADDR_MAP_AVALON_READ_INIT;
```

```
                ELSE
                    nextstate <= TAIL_ADDR_MAP_READ_INIT;
                END IF;


            WHEN TAIL_ADDR_MAP_AVALON_READ_INIT =>

                AV_MEM_ADDR <= SRAM_CTRL_VOSQ_TAIL_ADDR;
                AV_WRITE_DATA <= (others => 'Z');
                ADDR_LOAD_DATA <= (others => 'Z');
                OUT_DATA <= (others => 'Z');

                sdram_addr_data <= AV_READ_DATA;
                prev_packet_addr <= (others => 'Z');
                head_tail_addr <= (others => 'Z');
                empty_avail_addr <= (others => 'Z');
                get_pointer_addr <= (others => 'Z');
                first_pkt_flag <= '1';

                SRAM_IDLE_AREA_INIT_RSTN <= '0';
                SRAM_IDLE_AREA_INIT_EN <= '0';-- sram idle area initialization address
generator disable
                SRAM_QUEUE_AREA_INIT_EN <= '0';-- sram queue area initialization address
generator disable
                SRAM_QUEUE_AREA_INIT_RSTN <= '0';-- sram queue area initialization address
generator reset disabled
                SRAM_RD_WR_TIMER_EN <= '0';-- sram delta delay between read or a write
disabled
                SRAM_WR_RQT_TIMER_EN <= '0';-- sram write delay disabled
                SRAM_RD_RQT_TIMER_EN <= '0';-- sram read delay disabled
                DELAY_TIMER_EN <= '0';-- sdram write delay disabled
                Q_LGTH_CNT_EN <= '0';
                Q_LGTH_CTL <= '1';
                AV_READ_N <= '0';
                AV_WRITE_N <= '1';
                PRE_FIFO_RDREQ <= '0';
                POST_FIFO_WRREQ <= '0';
                AV_FLUSH <= '0';
                PACKET_CNT_EN <= '0';
                PACKET_CNT_RST <= '0';
                ADDR_LOAD <= '0';
                MASTER_MEM_INIT_DONE <= '0';
                out_bus_packet_data <= (others => '0');

                next_op <= '0';  -- write takes precedence
                nextstate <= HEAD_ADDR_MAP_WRITE_INIT;


            WHEN HEAD_ADDR_MAP_WRITE_INIT =>

                AV_MEM_ADDR <= SRAM_CTRL_VOSQ_HEAD_ADDR;
                sdram_addr_data <= (others => 'Z');
```

```
                AV_WRITE_DATA <= SRAM_QUEUE_AREA_ADDR;
                ADDR_LOAD_DATA <= (others => 'Z');
                OUT_DATA <= (others => 'Z');

                prev_packet_addr <= (others => 'Z');
                head_tail_addr <= (others => 'Z');
                empty_avail_addr <= (others => 'Z');
                get_pointer_addr <= (others => 'Z');
                first_pkt_flag <= '1';

                SRAM_QUEUE_AREA_INIT_EN <= '0';-- sram queue area initialization address
generator
                SRAM_QUEUE_AREA_INIT_RSTN <= '0';-- sram queue area initialization address
generator reset
                SRAM_IDLE_AREA_INIT_EN <= '0';-- sram idle area initialization address
generator disable
                SRAM_IDLE_AREA_INIT_RSTN <= '0';-- sram idle area initialization address
generator reset disabled
                SRAM_RD_WR_TIMER_EN <= '0';-- sram delta delay between read or a write
disabled
                SRAM_WR_RQT_TIMER_EN <= '1';-- sram write delay disabled
                SRAM_RD_RQT_TIMER_EN <= '0';-- sram read delay disabled
                DELAY_TIMER_EN <= '0';-- sdram write delay disabled
                Q_LGTH_CNT_EN <= '0';
                Q_LGTH_CTL <= '1';
                AV_READ_N <= '1';
                AV_WRITE_N <= '0';
                PRE_FIFO_RDREQ <= '0';
                POST_FIFO_WRREQ <= '0';
                AV_FLUSH <= '0';
                PACKET_CNT_EN <= '0';
                PACKET_CNT_RST <= '0';
                ADDR_LOAD <= '0';
                MASTER_MEM_INIT_DONE <= '0';
                out_bus_packet_data <= (others => '0');
                next_op <= '0';  -- write takes precedence

                IF SRAM_WR_RQT_DELAY_FLAG = '1' THEN
                    IF AV_WAITREQUEST = '1' THEN
                        nextstate <= HEAD_ADDR_MAP_WRITE_INIT;
                    ELSE
                        nextstate <= HEAD_ADDR_MAP_READ_INIT;
                    END IF;
                ELSE
                    nextstate <= HEAD_ADDR_MAP_WRITE_INIT;
                END IF;


            WHEN HEAD_ADDR_MAP_READ_INIT =>

                AV_MEM_ADDR <= SRAM_CTRL_VOSQ_HEAD_ADDR;
                sdram_addr_data <= (others => 'Z');
```

```
                AV_WRITE_DATA <= (others => 'Z');
                ADDR_LOAD_DATA <= (others => 'Z');
                OUT_DATA <= (others => 'Z');

                prev_packet_addr <= (others => 'Z');
                head_tail_addr <= (others => 'Z');
                empty_avail_addr <= (others => 'Z');
                get_pointer_addr <= (others => 'Z');
                first_pkt_flag <= '1';

                SRAM_QUEUE_AREA_INIT_RSTN <= '0';-- sram queue area initialization address
generator reset
                SRAM_QUEUE_AREA_INIT_EN <= '0';-- sram queue area initialization address
generator
                SRAM_IDLE_AREA_INIT_EN <= '0';-- sram idle area initialization address
generator disable
                SRAM_IDLE_AREA_INIT_RSTN <= '0';-- sram idle area initialization address
generator reset disabled
                SRAM_RD_WR_TIMER_EN <= '0';-- sram delta delay between read or a write
disabled
                SRAM_WR_RQT_TIMER_EN <= '0';-- sram write delay disabled
                SRAM_RD_RQT_TIMER_EN <= '1';-- sram read delay disabled
                DELAY_TIMER_EN <= '0';-- sdram write delay disabled
                Q_LGTH_CNT_EN <= '0';
                Q_LGTH_CTL <= '1';
                AV_READ_N <= '0';
                AV_WRITE_N <= '1';
                PRE_FIFO_RDREQ <= '0';
                POST_FIFO_WRREQ <= '0';
                AV_FLUSH <= '0';
                PACKET_CNT_EN <= '0';
                PACKET_CNT_RST <= '0';
                ADDR_LOAD <= '0';
                MASTER_MEM_INIT_DONE <= '0';
                out_bus_packet_data <= (others => '0');

                next_op <= '0';  -- write takes precedence

                IF SRAM_RD_RQT_DELAY_FLAG = '1' THEN
                        nextstate <= HEAD_ADDR_MAP_AVALON_READ_INIT;
                ELSE
                    nextstate <= HEAD_ADDR_MAP_READ_INIT;
                END IF;


            WHEN HEAD_ADDR_MAP_AVALON_READ_INIT =>

                AV_MEM_ADDR <= SRAM_CTRL_VOSQ_HEAD_ADDR;
                AV_WRITE_DATA <= (others => 'Z');
                ADDR_LOAD_DATA <= (others => 'Z');
                OUT_DATA <= (others => 'Z');
```

```
                    sdram_addr_data <= AV_READ_DATA;
                    prev_packet_addr <= (others => 'Z');
                    head_tail_addr <= (others => 'Z');
                    empty_avail_addr <= (others => 'Z');
                    get_pointer_addr <= (others => 'Z');
                    first_pkt_flag <= '1';

                    SRAM_IDLE_AREA_INIT_RSTN <= '0';
                    SRAM_IDLE_AREA_INIT_EN <= '0';-- sram idle area initialization address
generator disable
                    SRAM_QUEUE_AREA_INIT_EN <= '0';-- sram queue area initialization address
generator disable
                    SRAM_QUEUE_AREA_INIT_RSTN <= '0';-- sram queue area initialization address
generator reset disabled
                    SRAM_RD_WR_TIMER_EN <= '0';-- sram delta delay between read or a write
disabled
                    SRAM_WR_RQT_TIMER_EN <= '0';-- sram write delay disabled
                    SRAM_RD_RQT_TIMER_EN <= '0';-- sram read delay disabled
                    DELAY_TIMER_EN <= '0';-- sdram write delay disabled
                    Q_LGTH_CNT_EN <= '0';
                    Q_LGTH_CTL <= '1';
                    AV_READ_N <= '0';
                    AV_WRITE_N <= '1';
                    PRE_FIFO_RDREQ <= '0';
                    POST_FIFO_WRREQ <= '0';
                    AV_FLUSH <= '0';
                    PACKET_CNT_EN <= '0';
                    PACKET_CNT_RST <= '0';
                    ADDR_LOAD <= '0';
                    MASTER_MEM_INIT_DONE <= '0';
                    out_bus_packet_data <= (others => '0');

                    next_op <= '0';  -- write takes precedence
                    nextstate <= INIT;


                WHEN INIT =>
                    AV_MEM_ADDR <= (others => 'Z');
                    sdram_addr_data <= SDRAM_CTRL_ADDR; -- use the SDRAM start address
location for first packet case
                    AV_WRITE_DATA <= (others => 'Z');
                    ADDR_LOAD_DATA <= (others => 'Z');
                    OUT_DATA <= (others => 'Z');

                    prev_packet_addr <= (others => 'Z');
                    head_tail_addr <= (others => 'Z');
                    empty_avail_addr <= (others => 'Z');
                    get_pointer_addr <= (others => 'Z');

                    SRAM_IDLE_AREA_INIT_EN <= '0';-- sram idle area initialization address
generator disable
                    SRAM_QUEUE_AREA_INIT_EN <= '0';-- sram queue area initialization address
```

```
generator disable
                    SRAM_IDLE_AREA_INIT_RSTN <= '0';-- sram idle area initialization address
generator reset disabled
                    SRAM_QUEUE_AREA_INIT_RSTN <= '0';-- sram queue area initialization address
generator reset disabled
                    SRAM_RD_WR_TIMER_EN <= '0';-- sram delta delay between read or a write
disabled
                    SRAM_WR_RQT_TIMER_EN <= '0';-- sram write delay disabled
                    SRAM_RD_RQT_TIMER_EN <= '0';-- sram read delay disabled
                    DELAY_TIMER_EN <= '0';-- sdram write delay disabled
                    Q_LGTH_CNT_EN <= '0';
                    Q_LGTH_CTL <= '1';
                    AV_READ_N <= '1';
                    AV_WRITE_N <= '1';
                    PRE_FIFO_RDREQ <= '0';
                    POST_FIFO_WRREQ <= '0';
                    AV_FLUSH <= '1';
                    PACKET_CNT_EN <= '0';
                    PACKET_CNT_RST <= '0';
                    ADDR_LOAD <= '0';
                    MASTER_MEM_INIT_DONE <= '1';
                    out_bus_packet_data <= (others => '0');

                    -- This multiplexer allows for the same VOSQ master to be used for all VOSQ's
                    -- However, it ensure that only one Master is tasked for the address mapping
                    -- on startup and others are simple queues who manage incoming data.
                    IF (MEM_INIT_COMPL = '1') THEN

                        IF (PRE_FIFO_EMPTY = '0' and rd_wr_op = '0') THEN

                            -- Only the main master peripheral is allowed to write the first packet
                            -- received. This allows to write to SDRAM directly without excercising
                            -- the normal state logic required to update the linked-list.
                            IF MASTER_ID = '1' and first_pkt_flag = '1' THEN
                                next_op <= '0';  -- keep to write mode to sdram at initialization
                                nextstate <= PRELOAD_ADDR_CNTR;
                            ELSE
                                next_op <= '1';  -- arbitrate to read sdram mode next time
                                nextstate <= RQT_FOR_VOSQ_TAIL_ADDR;
                            END IF;

                        ELSE
                            IF (POST_FIFO_FULL = '0' and rd_wr_op = '1') THEN
                                nextstate <= RQT_FOR_VOSQ_HEAD_ADDR;
                            ELSE
                                nextstate <= INIT;
                            END IF;
                            next_op <= '0';  -- for next loop go into write sdram mode
                        END IF;

                    ELSE
                        next_op <= '0';  -- write takes preference on initialization
```

```
                nextstate <= INIT;

            END IF;

----- State machine section for Reading a packet from a linked-list using SRAM and SDRAM ----------

        WHEN RQT_FOR_VOSQ_HEAD_ADDR =>

            AV_MEM_ADDR <= SRAM_CTRL_VOSQ_HEAD_ADDR;
            sdram_addr_data <= (others => 'Z');
            AV_WRITE_DATA <= (others => 'Z');
            ADDR_LOAD_DATA <= (others => 'Z');
            OUT_DATA <= (others => 'Z');

            prev_packet_addr <= (others => 'Z');
            head_tail_addr <= (others => 'Z');
            empty_avail_addr <= (others => 'Z');
            get_pointer_addr <= (others => 'Z');
            first_pkt_flag <= '0';

            SRAM_IDLE_AREA_INIT_EN <= '0';-- sram idle area initialization address
generator disable
            SRAM_QUEUE_AREA_INIT_EN <= '0';-- sram queue area initialization address
generator disable
            SRAM_IDLE_AREA_INIT_RSTN <= '0';-- sram idle area initialization address
generator reset disabled
            SRAM_QUEUE_AREA_INIT_RSTN <= '0';-- sram queue area initialization address
generator reset disabled
            SRAM_RD_WR_TIMER_EN <= '1';-- sram delta delay between read or a write
disabled
            SRAM_WR_RQT_TIMER_EN <= '0';-- sram write delay disabled
            SRAM_RD_RQT_TIMER_EN <= '0';-- sram read delay disabled
            DELAY_TIMER_EN <= '0';    -- sdram write delay disabled
            Q_LGTH_CNT_EN <= '0';
            Q_LGTH_CTL <= '1';
            AV_READ_N <= '0';
            AV_WRITE_N <= '1';
            PRE_FIFO_RDREQ <= '0';
            POST_FIFO_WRREQ <= '0';
            AV_FLUSH <= '0';
            PACKET_CNT_EN <= '0';
            PACKET_CNT_RST <= '1';
            ADDR_LOAD <= '0';
            MASTER_MEM_INIT_DONE <= '1';
            out_bus_packet_data <= (others => '0');


            next_op <= '0';  -- for next loop go into write sdram mode
            IF SRAM_RD_WR_DELAY_FLAG = '1' THEN
                    nextstate <= CAPTURE_AV_READ_RQT_FOR_VOSQ_HEAD_ADDR;
            ELSE
```

```
                nextstate <= RQT_FOR_VOSQ_HEAD_ADDR;
            END IF;


        WHEN CAPTURE_AV_READ_RQT_FOR_VOSQ_HEAD_ADDR =>

            AV_MEM_ADDR <= SRAM_CTRL_VOSQ_HEAD_ADDR;
            sdram_addr_data <= (others => 'Z');
            AV_WRITE_DATA <= (others => 'Z');
            ADDR_LOAD_DATA <= (others => 'Z');
            OUT_DATA <= (others => 'Z');

            prev_packet_addr <= (others => 'Z');
            head_tail_addr <= AV_READ_DATA;
            empty_avail_addr <= (others => 'Z');
            get_pointer_addr <= (others => 'Z');
            first_pkt_flag <= '0';

            SRAM_IDLE_AREA_INIT_EN <= '0';-- sram idle area initialization address
generator disable
            SRAM_QUEUE_AREA_INIT_EN <= '0';-- sram queue area initialization address
generator disable
            SRAM_IDLE_AREA_INIT_RSTN <= '0';-- sram idle area initialization address
generator reset disabled
            SRAM_QUEUE_AREA_INIT_RSTN <= '0';-- sram queue area initialization address
generator reset disabled
            SRAM_RD_WR_TIMER_EN <= '0';-- sram delta delay between read or a write
disabled
            SRAM_WR_RQT_TIMER_EN <= '0';-- sram write delay disabled
            SRAM_RD_RQT_TIMER_EN <= '0';-- sram read delay disabled
            DELAY_TIMER_EN <= '0';    -- sdram write delay disabled
            Q_LGTH_CNT_EN <= '0';
            Q_LGTH_CTL <= '1';
            AV_READ_N <= '0';
            AV_WRITE_N <= '1';
            PRE_FIFO_RDREQ <= '0';
            POST_FIFO_WRREQ <= '0';
            AV_FLUSH <= '0';
            PACKET_CNT_EN <= '0';
            PACKET_CNT_RST <= '1';
            ADDR_LOAD <= '0';
            MASTER_MEM_INIT_DONE <= '1';
            out_bus_packet_data <= (others => '0');


            next_op <= '0';  -- for next loop go into write sdram mode
            nextstate <= SET_IDLE_PTR_ADDR_CTL;


        WHEN UPDATE_IDLE_PTR_ADDR_IN_EMPTY_LOC =>

            AV_MEM_ADDR <= empty_avail_addr;
```

```
                sdram_addr_data <= (others => 'Z');
                AV_WRITE_DATA <= head_tail_addr;
                ADDR_LOAD_DATA <= (others => 'Z');
                OUT_DATA <= (others => 'Z');

                prev_packet_addr <= (others => 'Z');
                get_pointer_addr <= (others => 'Z');
                first_pkt_flag <= '0';

                SRAM_IDLE_AREA_INIT_EN <= '0';-- sram idle area initialization address
generator disable
                SRAM_QUEUE_AREA_INIT_EN <= '0';-- sram queue area initialization address
generator disable
                SRAM_IDLE_AREA_INIT_RSTN <= '0';-- sram idle area initialization address
generator reset disabled
                SRAM_QUEUE_AREA_INIT_RSTN <= '0';-- sram queue area initialization address
generator reset disabled
                SRAM_RD_WR_TIMER_EN <= '0';-- sram delta delay between read or a write
disabled
                SRAM_WR_RQT_TIMER_EN <= '1';-- sram write delay disabled
                SRAM_RD_RQT_TIMER_EN <= '0';-- sram read delay disabled
                DELAY_TIMER_EN <= '0';  -- sdram write delay triggered
                Q_LGTH_CNT_EN <= '0';
                Q_LGTH_CTL <= '1';
                AV_READ_N <= '1';
                AV_WRITE_N <= '0';
                PRE_FIFO_RDREQ <= '0';
                POST_FIFO_WRREQ <= '0';
                AV_FLUSH <= '0';
                PACKET_CNT_EN <= '0';
                PACKET_CNT_RST <= '1';
                ADDR_LOAD <= '0';
                MASTER_MEM_INIT_DONE <= '1';
                out_bus_packet_data <= (others => '0');

                next_op <= '0';  -- for next loop go into write packet mode
                IF SRAM_WR_RQT_DELAY_FLAG = '1' THEN
                    IF AV_WAITREQUEST = '1' THEN
                        nextstate <= UPDATE_IDLE_PTR_ADDR_IN_EMPTY_LOC;
                    ELSE
                        nextstate <= RQT_PREV_PACKET_AT_HEAD_ADDR;
                    END IF;
                ELSE
                    nextstate <= UPDATE_IDLE_PTR_ADDR_IN_EMPTY_LOC;
                END IF;

            WHEN RQT_PREV_PACKET_AT_HEAD_ADDR =>

                AV_MEM_ADDR <= head_tail_addr;
                sdram_addr_data <= (others => 'Z');
                AV_WRITE_DATA <= (others => 'Z');
                ADDR_LOAD_DATA <= (others => 'Z');
```

```
                OUT_DATA <= (others => 'Z');

                prev_packet_addr <= (others => 'Z');
                empty_avail_addr <= (others => 'Z');
                get_pointer_addr <= (others => 'Z');
                first_pkt_flag <= '0';

                SRAM_IDLE_AREA_INIT_EN <= '0';-- sram idle area initialization address
generator disable
                SRAM_QUEUE_AREA_INIT_EN <= '0';-- sram queue area initialization address
generator disable
                SRAM_IDLE_AREA_INIT_RSTN <= '0';-- sram idle area initialization address
generator reset disabled
                SRAM_QUEUE_AREA_INIT_RSTN <= '0';-- sram queue area initialization address
generator reset disabled
                SRAM_RD_WR_TIMER_EN <= '0';-- sram delta delay between read or a write
disabled
                SRAM_WR_RQT_TIMER_EN <= '0';-- sram write delay disabled
                SRAM_RD_RQT_TIMER_EN <= '1';-- sram read delay disabled
                DELAY_TIMER_EN <= '0';    -- sdram write delay disabled
                Q_LGTH_CNT_EN <= '0';
                Q_LGTH_CTL <= '1';
                AV_READ_N <= '0';
                AV_WRITE_N <= '1';
                PRE_FIFO_RDREQ <= '0';
                POST_FIFO_WRREQ <= '0';
                AV_FLUSH <= '0';
                PACKET_CNT_EN <= '0';
                PACKET_CNT_RST <= '1';
                ADDR_LOAD <= '0';
                MASTER_MEM_INIT_DONE <= '1';
                out_bus_packet_data <= (others => '0');


                next_op <= '0';  -- for next loop go into write packet mode

                IF SRAM_RD_RQT_DELAY_FLAG = '1' THEN
                        nextstate <= CAPTURE_AV_RQT_PREV_PACKET_AT_HEAD_ADDR;
                ELSE
                        nextstate <= RQT_PREV_PACKET_AT_HEAD_ADDR;
                END IF;

        WHEN CAPTURE_AV_RQT_PREV_PACKET_AT_HEAD_ADDR =>

                AV_MEM_ADDR <= head_tail_addr;
                sdram_addr_data <= (others => 'Z');
                AV_WRITE_DATA <= (others => 'Z');
                ADDR_LOAD_DATA <= (others => 'Z');
                OUT_DATA <= (others => 'Z');

                prev_packet_addr <= AV_READ_DATA;
                empty_avail_addr <= empty_avail_addr;
```

```
                    get_pointer_addr <= (others => 'Z');
                    first_pkt_flag <= '0';

                    SRAM_IDLE_AREA_INIT_EN <= '0';-- sram idle area initialization address
generator disable
                    SRAM_QUEUE_AREA_INIT_EN <= '0';-- sram queue area initialization address
generator disable
                    SRAM_IDLE_AREA_INIT_RSTN <= '0';-- sram idle area initialization address
generator reset disabled
                    SRAM_QUEUE_AREA_INIT_RSTN <= '0';-- sram queue area initialization address
generator reset disabled
                    SRAM_RD_WR_TIMER_EN <= '0';-- sram delta delay between read or a write
disabled
                    SRAM_WR_RQT_TIMER_EN <= '0';-- sram write delay disabled
                    SRAM_RD_RQT_TIMER_EN <= '0';-- sram read delay disabled
                    DELAY_TIMER_EN <= '0';   -- sdram write delay disabled
                    Q_LGTH_CNT_EN <= '0';
                    Q_LGTH_CTL <= '1';
                    AV_READ_N <= '0';
                    AV_WRITE_N <= '1';
                    PRE_FIFO_RDREQ <= '0';
                    POST_FIFO_WRREQ <= '0';
                    AV_FLUSH <= '0';
                    PACKET_CNT_EN <= '0';
                    PACKET_CNT_RST <= '1';
                    ADDR_LOAD <= '0';
                    MASTER_MEM_INIT_DONE <= '1';
                    out_bus_packet_data <= (others => '0');


                    next_op <= '0';  -- for next loop go into write packet mode
                    nextstate <= UPDATE_VOSQ_HEAD_PTR_ADDR;


                WHEN UPDATE_VOSQ_HEAD_PTR_ADDR =>

                    AV_MEM_ADDR <= SRAM_CTRL_VOSQ_HEAD_ADDR;
                    sdram_addr_data <= prev_packet_addr;
                    AV_WRITE_DATA <= (SRAM_AVALON_BUS_OFFSET & prev_packet_addr(15
DOWNTO 0));
                    ADDR_LOAD_DATA <= (others => 'Z');
                    OUT_DATA <= (others => 'Z');

                    head_tail_addr <= (others => 'Z');
                    empty_avail_addr <= (others => 'Z');
                    get_pointer_addr <= (others => 'Z');
                    first_pkt_flag <= '0';

                    SRAM_IDLE_AREA_INIT_EN <= '0';-- sram idle area initialization address
generator disable
                    SRAM_QUEUE_AREA_INIT_EN <= '0';-- sram queue area initialization address
generator disable
```

```
                    SRAM_IDLE_AREA_INIT_RSTN <= '0';-- sram idle area initialization address
generator reset disabled
                    SRAM_QUEUE_AREA_INIT_RSTN <= '0';-- sram queue area initialization address
generator reset disabled
                    SRAM_RD_WR_TIMER_EN <= '0';-- sram delta delay between read or a write
disabled
                    SRAM_WR_RQT_TIMER_EN <= '1';-- sram write delay disabled
                    SRAM_RD_RQT_TIMER_EN <= '0';-- sram read delay disabled
                    DELAY_TIMER_EN <= '0';  -- sdram write delay triggered
                    Q_LGTH_CNT_EN <= '0';
                    Q_LGTH_CTL <= '1';
                    AV_READ_N <= '1';
                    AV_WRITE_N <= '0';
                    PRE_FIFO_RDREQ <= '0';
                    POST_FIFO_WRREQ <= '0';
                    AV_FLUSH <= '0';
                    PACKET_CNT_EN <= '0';
                    PACKET_CNT_RST <= '1';
                    ADDR_LOAD <= '0';
                    MASTER_MEM_INIT_DONE <= '1';
                    out_bus_packet_data <= (others => '0');

                    next_op <= '0';  -- for next loop go into write packet mode
                    IF SRAM_WR_RQT_DELAY_FLAG = '1' THEN
                        IF AV_WAITREQUEST = '1' THEN
                            nextstate <= UPDATE_VOSQ_HEAD_PTR_ADDR;
                        ELSE
                            nextstate <= PRELOAD_ADDR_CNTR;
                        END IF;
                    ELSE
                        nextstate <= UPDATE_VOSQ_HEAD_PTR_ADDR;
                    END IF;


----- End of State machine section for Reading a packet from a linked-list using SRAM and SDRAM ----------


----- State machine section for Writing a packet to a linked-list using SRAM and SDRAM ----------
                WHEN RQT_FOR_VOSQ_TAIL_ADDR =>

                    AV_MEM_ADDR <= SRAM_CTRL_VOSQ_TAIL_ADDR;
                    sdram_addr_data <= (others => 'Z');
                    AV_WRITE_DATA <= (others => 'Z');
                    ADDR_LOAD_DATA <= (others => 'Z');
                    OUT_DATA <= (others => 'Z');

                    prev_packet_addr <= (others => 'Z');
                    head_tail_addr <= (others => 'Z');
                    empty_avail_addr <= (others => 'Z');
                    get_pointer_addr <= (others => 'Z');
                    first_pkt_flag <= '0';
```

```
                SRAM_IDLE_AREA_INIT_EN <= '0';-- sram idle area initialization address
generator disable
                SRAM_QUEUE_AREA_INIT_EN <= '0';-- sram queue area initialization address
generator disable
                SRAM_IDLE_AREA_INIT_RSTN <= '0';-- sram idle area initialization address
generator reset disabled
                SRAM_QUEUE_AREA_INIT_RSTN <= '0';-- sram queue area initialization address
generator reset disabled
                SRAM_RD_WR_TIMER_EN <= '1';-- sram delta delay between read or a write
disabled
                SRAM_WR_RQT_TIMER_EN <= '0';-- sram write delay disabled
                SRAM_RD_RQT_TIMER_EN <= '0';-- sram read delay disabled
                DELAY_TIMER_EN <= '0';    -- sdram write delay disabled
                Q_LGTH_CNT_EN <= '0';
                Q_LGTH_CTL <= '1';
                AV_READ_N <= '0';
                AV_WRITE_N <= '1';
                PRE_FIFO_RDREQ <= '0';
                POST_FIFO_WRREQ <= '0';
                AV_FLUSH <= '0';
                PACKET_CNT_EN <= '0';
                PACKET_CNT_RST <= '1';
                ADDR_LOAD <= '0';
                MASTER_MEM_INIT_DONE <= '1';
                out_bus_packet_data <= (others => '0');


                next_op <= '1';  -- for next loop go into read sdram mode

                -- Needed to insert this delay timer as this state requires 3xTclk
                -- latency before waitrequest signal is asserted. Do not know why it
                -- is different then the other SRAM reads before.
                IF SRAM_RD_WR_DELAY_FLAG = '1' THEN
                        nextstate <= CAPTURE_AV_READ_VOSQ_TAIL_ADDR;
                ELSE
                        nextstate <= RQT_FOR_VOSQ_TAIL_ADDR;
                END IF;


        WHEN CAPTURE_AV_READ_VOSQ_TAIL_ADDR =>

                AV_MEM_ADDR <= SRAM_CTRL_VOSQ_TAIL_ADDR;
                sdram_addr_data <= (others => 'Z');
                AV_WRITE_DATA <= (others => 'Z');
                ADDR_LOAD_DATA <= (others => 'Z');
                OUT_DATA <= (others => 'Z');

                prev_packet_addr <= (others => 'Z');
                head_tail_addr <= AV_READ_DATA;
                empty_avail_addr <= (others => 'Z');
                get_pointer_addr <= (others => 'Z');
```

```
                    first_pkt_flag <= '0';

                    SRAM_IDLE_AREA_INIT_EN <= '0';-- sram idle area initialization address
generator disable
                    SRAM_QUEUE_AREA_INIT_EN <= '0';-- sram queue area initialization address
generator disable
                    SRAM_IDLE_AREA_INIT_RSTN <= '0';-- sram idle area initialization address
generator reset disabled
                    SRAM_QUEUE_AREA_INIT_RSTN <= '0';-- sram queue area initialization address
generator reset disabled
                    SRAM_RD_WR_TIMER_EN <= '0';-- sram delta delay between read or a write
disabled
                    SRAM_WR_RQT_TIMER_EN <= '0';-- sram write delay disabled
                    SRAM_RD_RQT_TIMER_EN <= '0';-- sram read delay disabled
                    DELAY_TIMER_EN <= '0';   -- sdram write delay disabled
                    Q_LGTH_CNT_EN <= '0';
                    Q_LGTH_CTL <= '1';
                    AV_READ_N <= '0';
                    AV_WRITE_N <= '1';
                    PRE_FIFO_RDREQ <= '0';
                    POST_FIFO_WRREQ <= '0';
                    AV_FLUSH <= '0';
                    PACKET_CNT_EN <= '0';
                    PACKET_CNT_RST <= '1';
                    ADDR_LOAD <= '0';
                    MASTER_MEM_INIT_DONE <= '1';
                    out_bus_packet_data <= (others => '0');


                    next_op <= '1';  -- for next loop go into read sdram mode
                    nextstate <= SET_IDLE_PTR_ADDR_CTL;


                WHEN SET_IDLE_PTR_ADDR_CTL =>

                    AV_MEM_ADDR <= IDLE_ADDR_CNTR_SLAVE_ADDR;
                    sdram_addr_data <= (others => 'Z');
                    ADDR_LOAD_DATA <= (others => 'Z');
                    OUT_DATA <= (others => 'Z');

                    prev_packet_addr <= (others => 'Z');
                    head_tail_addr <= head_tail_addr;
                    empty_avail_addr <= (others => 'Z');
                    get_pointer_addr <= (others => 'Z');
                    first_pkt_flag <= '0';

                    SRAM_IDLE_AREA_INIT_EN <= '0';-- sram idle area initialization address
generator disable
                    SRAM_QUEUE_AREA_INIT_EN <= '0';-- sram queue area initialization address
generator disable
                    SRAM_IDLE_AREA_INIT_RSTN <= '0';-- sram idle area initialization address
generator reset disabled
```

```
                    SRAM_QUEUE_AREA_INIT_RSTN <= '0';-- sram queue area initialization address
generator reset disabled
                    SRAM_RD_WR_TIMER_EN <= '0';-- sram delta delay between read or a write
disabled
                    SRAM_WR_RQT_TIMER_EN <= '1';-- sram write delay disabled
                    SRAM_RD_RQT_TIMER_EN <= '0';-- sram read delay disabled
                    DELAY_TIMER_EN <= '0';  -- sdram write delay triggered
                    Q_LGTH_CNT_EN <= '0';
                    Q_LGTH_CTL <= '1';
                    AV_READ_N <= '1';
                    AV_WRITE_N <= '0';
                    PRE_FIFO_RDREQ <= '0';
                    POST_FIFO_WRREQ <= '0';
                    AV_FLUSH <= '0';
                    PACKET_CNT_EN <= '0';
                    PACKET_CNT_RST <= '1';
                    ADDR_LOAD <= '0';
                    MASTER_MEM_INIT_DONE <= '1';
                    out_bus_packet_data <= (others => '0');

                    IF rd_wr_op = '1' THEN-- current state is in write packet mode; used for arbitration
                        AV_WRITE_DATA <= x"00000001";-- address[0] represents "get_store_ctl" bit
for the free space counter
                        next_op <= '1';-- retrieve an address
                    ELSE        -- current state is in read packet mode; used for arbitration
                        AV_WRITE_DATA <= x"00000000";-- address[0] represents "get_store_ctl" bit
for the free space counter
                        next_op <= '0';-- return an address
                    END IF;

                    IF SRAM_WR_RQT_DELAY_FLAG = '1' THEN
                            nextstate <= DELAY_RQT_IDLE_PTR_ADDR;
                    ELSE
                        nextstate <= SET_IDLE_PTR_ADDR_CTL;
                    END IF;


                WHEN DELAY_RQT_IDLE_PTR_ADDR =>

                    AV_MEM_ADDR <= IDLE_ADDR_CNTR_SLAVE_ADDR;
                    sdram_addr_data <= (others => 'Z');
                    AV_WRITE_DATA <= (others => 'Z');
                    ADDR_LOAD_DATA <= (others => 'Z');
                    OUT_DATA <= (others => 'Z');

                    prev_packet_addr <= (others => 'Z');
                    head_tail_addr <= head_tail_addr;
                    empty_avail_addr <= (others => 'Z');
                    get_pointer_addr <= (others => 'Z');
                    first_pkt_flag <= '0';

                    SRAM_IDLE_AREA_INIT_EN <= '0';-- sram idle area initialization address
```

```
generator disable
                SRAM_QUEUE_AREA_INIT_EN <= '0';-- sram queue area initialization address
generator disable
                SRAM_IDLE_AREA_INIT_RSTN <= '0';-- sram idle area initialization address
generator reset disabled
                SRAM_QUEUE_AREA_INIT_RSTN <= '0';-- sram queue area initialization address
generator reset disabled
                SRAM_RD_WR_TIMER_EN <= '0';-- sram delta delay between read or a write
disabled
                SRAM_WR_RQT_TIMER_EN <= '0';-- sram write delay disabled
                SRAM_RD_RQT_TIMER_EN <= '0';-- sram read delay disabled
                DELAY_TIMER_EN <= '0';  -- sdram write delay triggered
                Q_LGTH_CNT_EN <= '0';
                Q_LGTH_CTL <= '1';
                AV_READ_N <= '1';
                AV_WRITE_N <= '1';
                PRE_FIFO_RDREQ <= '0';
                POST_FIFO_WRREQ <= '0';
                AV_FLUSH <= '0';
                PACKET_CNT_EN <= '0';
                PACKET_CNT_RST <= '1';
                ADDR_LOAD <= '0';
                MASTER_MEM_INIT_DONE <= '1';
                out_bus_packet_data <= (others => '0');

                IF rd_wr_op = '1' THEN-- current state is in write packet mode; used for arbitration
                    next_op <= '1';
                ELSE        -- current state is in read packet mode; used for arbitration
                    next_op <= '0';
                END IF;

                nextstate <= RQT_IDLE_PTR_ADDR;


            WHEN RQT_IDLE_PTR_ADDR =>

                AV_MEM_ADDR <= IDLE_ADDR_CNTR_SLAVE_ADDR;
                sdram_addr_data <= (others => 'Z');
                AV_WRITE_DATA <= (others => 'Z');
                ADDR_LOAD_DATA <= (others => 'Z');
                OUT_DATA <= (others => 'Z');

                prev_packet_addr <= (others => 'Z');
                head_tail_addr <= head_tail_addr;
                empty_avail_addr <= (others => 'Z');
                get_pointer_addr <= (others => 'Z');
                first_pkt_flag <= '0';

                SRAM_IDLE_AREA_INIT_EN <= '0';-- sram idle area initialization address
generator disable
                SRAM_QUEUE_AREA_INIT_EN <= '0';-- sram queue area initialization address
```

```
generator disable
                SRAM_IDLE_AREA_INIT_RSTN <= '0';-- sram idle area initialization address
generator reset disabled
                SRAM_QUEUE_AREA_INIT_RSTN <= '0';-- sram queue area initialization address
generator reset disabled
                SRAM_RD_WR_TIMER_EN <= '0';-- sram delta delay between read or a write
disabled
                SRAM_WR_RQT_TIMER_EN <= '0';-- sram write delay disabled
                SRAM_RD_RQT_TIMER_EN <= '1';-- sram read delay disabled
                DELAY_TIMER_EN <= '0';  -- sdram write delay triggered
                Q_LGTH_CNT_EN <= '0';
                Q_LGTH_CTL <= '1';
                AV_READ_N <= '0';
                AV_WRITE_N <= '1';
                PRE_FIFO_RDREQ <= '0';
                POST_FIFO_WRREQ <= '0';
                AV_FLUSH <= '0';
                PACKET_CNT_EN <= '0';
                PACKET_CNT_RST <= '1';
                ADDR_LOAD <= '0';
                MASTER_MEM_INIT_DONE <= '1';
                out_bus_packet_data <= (others => '0');

                IF rd_wr_op = '1' THEN-- current state is in write packet mode; used for arbitration
                    next_op <= '1';
                ELSE        -- current state is in read packet mode; used for arbitration
                    next_op <= '0';
                END IF;

                -- Requires a hold-off of 1xTclk delay for the waitrequest to be asserted
                -- by the slave peripheral. But since there are multiple states before
                -- the data is ready by slave, instead a 2 x Tclk delay. The Write delay timer
                -- can not reset fast enough before entering this state so this was implemented.
                -- No checks are put in place since the data is captured on 3 x Tclk where
                -- waitrequest is deasserted
                IF SRAM_RD_RQT_DELAY_FLAG = '1' THEN
                        nextstate <= CAPTURE_IDLE_PTR_ADDR;
                ELSE
                    nextstate <= RQT_IDLE_PTR_ADDR;
                END IF;

            WHEN CAPTURE_IDLE_PTR_ADDR =>

                AV_MEM_ADDR <= IDLE_ADDR_CNTR_SLAVE_ADDR;
                sdram_addr_data <= (others => 'Z');
                AV_WRITE_DATA <= (others => 'Z');
                ADDR_LOAD_DATA <= (others => 'Z');
                OUT_DATA <= (others => 'Z');

                prev_packet_addr <= (others => 'Z');
                head_tail_addr <= head_tail_addr;
                empty_avail_addr <= AV_READ_DATA;
```

```
                    get_pointer_addr <= (others => 'Z');
                    first_pkt_flag <= '0';

                    SRAM_IDLE_AREA_INIT_EN <= '0';-- sram idle area initialization address
generator disable
                    SRAM_QUEUE_AREA_INIT_EN <= '0';-- sram queue area initialization address
generator disable
                    SRAM_IDLE_AREA_INIT_RSTN <= '0';-- sram idle area initialization address
generator reset disabled
                    SRAM_QUEUE_AREA_INIT_RSTN <= '0';-- sram queue area initialization address
generator reset disabled
                    SRAM_RD_WR_TIMER_EN <= '0';-- sram delta delay between read or a write
disabled
                    SRAM_WR_RQT_TIMER_EN <= '0';-- sram write delay disabled
                    SRAM_RD_RQT_TIMER_EN <= '0';-- sram read delay disabled
                    DELAY_TIMER_EN <= '0';  -- sdram write delay triggered
                    Q_LGTH_CNT_EN <= '0';
                    Q_LGTH_CTL <= '1';
                    AV_READ_N <= '0';
                    AV_WRITE_N <= '1';
                    PRE_FIFO_RDREQ <= '0';
                    POST_FIFO_WRREQ <= '0';
                    AV_FLUSH <= '0';
                    PACKET_CNT_EN <= '0';
                    PACKET_CNT_RST <= '1';
                    ADDR_LOAD <= '0';
                    MASTER_MEM_INIT_DONE <= '1';
                    out_bus_packet_data <= (others => '0');

                    IF AV_WAITREQUEST = '1' THEN
                        IF rd_wr_op = '1' THEN-- current state is in write packet mode; used for
arbitration
                            next_op <= '1';
                        ELSE    -- current state is in read packet mode; used for arbitration
                            next_op <= '0';
                        END IF;
                        nextstate <= CAPTURE_IDLE_PTR_ADDR;
                    ELSE
                        IF rd_wr_op = '1' THEN-- current state is in write packet mode; used for
arbitration
                            next_op <= '1';
                            nextstate <= RQT_AND_EXTRACT_IDLE_PTR_ADDR;
                        ELSE    -- current state is in read packet mode; used for arbitration
                            next_op <= '0';
                            nextstate <= UPDATE_IDLE_PTR_ADDR_IN_EMPTY_LOC;
                        END IF;
                    END IF;


                WHEN RQT_AND_EXTRACT_IDLE_PTR_ADDR =>

                    AV_MEM_ADDR <= empty_avail_addr;
```

```
sdram_addr_data <= (others => 'Z');
AV_WRITE_DATA <= (others => 'Z');
ADDR_LOAD_DATA <= (others => 'Z');
OUT_DATA <= (others => 'Z');

prev_packet_addr <= (others => 'Z');
head_tail_addr <= head_tail_addr;
get_pointer_addr <= (others => 'Z');
first_pkt_flag <= '0';

SRAM_IDLE_AREA_INIT_EN <= '0';-- sram idle area initialization address
generator disable

SRAM_QUEUE_AREA_INIT_EN <= '0';-- sram queue area initialization address
generator disable

SRAM_IDLE_AREA_INIT_RSTN <= '0';-- sram idle area initialization address
generator reset disabled

SRAM_QUEUE_AREA_INIT_RSTN <= '0';-- sram queue area initialization address
generator reset disabled

SRAM_RD_WR_TIMER_EN <= '1';-- sram delta delay between read or a write
disabled

SRAM_WR_RQT_TIMER_EN <= '0';-- sram write delay disabled
SRAM_RD_RQT_TIMER_EN <= '0';-- sram read delay disabled
DELAY_TIMER_EN <= '0';   -- sdram write delay disabled
Q_LGTH_CNT_EN <= '0';
Q_LGTH_CTL <= '1';
AV_READ_N <= '0';
AV_WRITE_N <= '1';
PRE_FIFO_RDREQ <= '0';
POST_FIFO_WRREQ <= '0';
AV_FLUSH <= '0';
PACKET_CNT_EN <= '0';
PACKET_CNT_RST <= '1';
ADDR_LOAD <= '0';
MASTER_MEM_INIT_DONE <= '1';
out_bus_packet_data <= (others => '0');


next_op <= '1';  -- for next loop go into read sdram mode

-- Needed to insert this delay timer as this state requires 3xTclk
-- latency before waitrequest signal is asserted. Do not know why it
-- is different then the other SRAM reads before.
IF SRAM_RD_WR_DELAY_FLAG = '1' THEN
        nextstate <= CAPTURE_AV_READ_IDLE_PTR_ADDR;
ELSE
        nextstate <= RQT_AND_EXTRACT_IDLE_PTR_ADDR;
END IF;


WHEN CAPTURE_AV_READ_IDLE_PTR_ADDR =>

        AV_MEM_ADDR <= empty_avail_addr;
```

```
                sdram_addr_data <= (others => 'Z');
                AV_WRITE_DATA <= (others => 'Z');
                ADDR_LOAD_DATA <= (others => 'Z');
                OUT_DATA <= (others => 'Z');

                prev_packet_addr <= (others => 'Z');
                head_tail_addr <= head_tail_addr;
                get_pointer_addr <= AV_READ_DATA;
                first_pkt_flag <= '0';

                SRAM_IDLE_AREA_INIT_EN <= '0';-- sram idle area initialization address
generator disable
                SRAM_QUEUE_AREA_INIT_EN <= '0';-- sram queue area initialization address
generator disable
                SRAM_IDLE_AREA_INIT_RSTN <= '0';-- sram idle area initialization address
generator reset disabled
                SRAM_QUEUE_AREA_INIT_RSTN <= '0';-- sram queue area initialization address
generator reset disabled
                SRAM_RD_WR_TIMER_EN <= '0';-- sram delta delay between read or a write
disabled
                SRAM_WR_RQT_TIMER_EN <= '0';-- sram write delay disabled
                SRAM_RD_RQT_TIMER_EN <= '0';-- sram read delay disabled
                DELAY_TIMER_EN <= '0';    -- sdram write delay disabled
                Q_LGTH_CNT_EN <= '0';
                Q_LGTH_CTL <= '1';
                AV_READ_N <= '0';
                AV_WRITE_N <= '1';
                PRE_FIFO_RDREQ <= '0';
                POST_FIFO_WRREQ <= '0';
                AV_FLUSH <= '0';
                PACKET_CNT_EN <= '0';
                PACKET_CNT_RST <= '1';
                ADDR_LOAD <= '0';
                MASTER_MEM_INIT_DONE <= '1';
                out_bus_packet_data <= (others => '0');


                next_op <= '1';  -- for next loop go into read sdram mode
                nextstate <= RQT_PREV_PACKET_PTR_ADDR;

        WHEN RQT_PREV_PACKET_PTR_ADDR =>

                AV_MEM_ADDR <= head_tail_addr;
                sdram_addr_data <= (others => 'Z');
                AV_WRITE_DATA <= (others => 'Z');
                ADDR_LOAD_DATA <= (others => 'Z');
                OUT_DATA <= (others => 'Z');

                prev_packet_addr <= (others => 'Z');
                empty_avail_addr <= empty_avail_addr;
                get_pointer_addr <= get_pointer_addr;
                first_pkt_flag <= '0';
```

```
                SRAM_IDLE_AREA_INIT_EN <= '0';-- sram idle area initialization address
generator disable
                SRAM_QUEUE_AREA_INIT_EN <= '0';-- sram queue area initialization address
generator disable
                SRAM_IDLE_AREA_INIT_RSTN <= '0';-- sram idle area initialization address
generator reset disabled
                SRAM_QUEUE_AREA_INIT_RSTN <= '0';-- sram queue area initialization address
generator reset disabled
                SRAM_RD_WR_TIMER_EN <= '0';-- sram delta delay between read or a write
disabled
                SRAM_WR_RQT_TIMER_EN <= '0';-- sram write delay disabled
                SRAM_RD_RQT_TIMER_EN <= '1';-- sram read delay disabled
                DELAY_TIMER_EN <= '0';   -- sdram write delay disabled
                Q_LGTH_CNT_EN <= '0';
                Q_LGTH_CTL <= '1';
                AV_READ_N <= '0';
                AV_WRITE_N <= '1';
                PRE_FIFO_RDREQ <= '0';
                POST_FIFO_WRREQ <= '0';
                AV_FLUSH <= '0';
                PACKET_CNT_EN <= '0';
                PACKET_CNT_RST <= '1';
                ADDR_LOAD <= '0';
                MASTER_MEM_INIT_DONE <= '1';
                out_bus_packet_data <= (others => '0');


                next_op <= '1';  -- for next loop go into read sdram mode

                IF SRAM_RD_RQT_DELAY_FLAG = '1' THEN
                        nextstate <= CAPTURE_AV_RQT_PREV_PKT_PTR_ADDR;
                ELSE
                        nextstate <= RQT_PREV_PACKET_PTR_ADDR;
                END IF;


        WHEN CAPTURE_AV_RQT_PREV_PKT_PTR_ADDR =>

                AV_MEM_ADDR <= head_tail_addr;
                sdram_addr_data <= (others => 'Z');
                AV_WRITE_DATA <= (others => 'Z');
                ADDR_LOAD_DATA <= (others => 'Z');
                OUT_DATA <= (others => 'Z');

                prev_packet_addr <= AV_READ_DATA;
                empty_avail_addr <= empty_avail_addr;
                get_pointer_addr <= get_pointer_addr;
                first_pkt_flag <= '0';

                SRAM_IDLE_AREA_INIT_EN <= '0';-- sram idle area initialization address
generator disable
```

```
                SRAM_QUEUE_AREA_INIT_EN <= '0';-- sram queue area initialization address
generator disable
                SRAM_IDLE_AREA_INIT_RSTN <= '0';-- sram idle area initialization address
generator reset disabled
                SRAM_QUEUE_AREA_INIT_RSTN <= '0';-- sram queue area initialization address
generator reset disabled
                SRAM_RD_WR_TIMER_EN <= '0';-- sram delta delay between read or a write
disabled
                SRAM_WR_RQT_TIMER_EN <= '0';-- sram write delay disabled
                SRAM_RD_RQT_TIMER_EN <= '0';-- sram read delay disabled
                DELAY_TIMER_EN <= '0';    -- sdram write delay disabled
                Q_LGTH_CNT_EN <= '0';
                Q_LGTH_CTL <= '1';
                AV_READ_N <= '0';
                AV_WRITE_N <= '1';
                PRE_FIFO_RDREQ <= '0';
                POST_FIFO_WRREQ <= '0';
                AV_FLUSH <= '0';
                PACKET_CNT_EN <= '0';
                PACKET_CNT_RST <= '1';
                ADDR_LOAD <= '0';
                MASTER_MEM_INIT_DONE <= '1';
                out_bus_packet_data <= (others => '0');


                next_op <= '1';  -- for next loop go into read sdram mode
                nextstate <= UPDATE_PREV_PACKET_PTR_ADDR;


        WHEN UPDATE_PREV_PACKET_PTR_ADDR =>

                AV_MEM_ADDR <= head_tail_addr;
                sdram_addr_data <= (others => 'Z');
                AV_WRITE_DATA <= (prev_packet_addr(31 DOWNTO 16) & get_pointer_addr(15
DOWNTO 0));
                ADDR_LOAD_DATA <= (others => 'Z');
                OUT_DATA <= (others => 'Z');

                empty_avail_addr <= empty_avail_addr;
                first_pkt_flag <= '0';

                SRAM_IDLE_AREA_INIT_EN <= '0';-- sram idle area initialization address
generator disable
                SRAM_QUEUE_AREA_INIT_EN <= '0';-- sram queue area initialization address
generator disable
                SRAM_IDLE_AREA_INIT_RSTN <= '0';-- sram idle area initialization address
generator reset disabled
                SRAM_QUEUE_AREA_INIT_RSTN <= '0';-- sram queue area initialization address
generator reset disabled
                SRAM_RD_WR_TIMER_EN <= '0';-- sram delta delay between read or a write
disabled
                SRAM_WR_RQT_TIMER_EN <= '1';-- sram write delay disabled
```

```
SRAM_RD_RQT_TIMER_EN <= '0';-- sram read delay disabled
DELAY_TIMER_EN <= '0';  -- sdram write delay triggered
Q_LGTH_CNT_EN <= '0';
Q_LGTH_CTL <= '1';
AV_READ_N <= '1';
AV_WRITE_N <= '0';
PRE_FIFO_RDREQ <= '0';
POST_FIFO_WRREQ <= '0';
AV_FLUSH <= '0';
PACKET_CNT_EN <= '0';
PACKET_CNT_RST <= '1';
ADDR_LOAD <= '0';
MASTER_MEM_INIT_DONE <= '1';
out_bus_packet_data <= (others => '0');

next_op <= '1';  -- for next loop go into read sdram mode
IF SRAM_WR_RQT_DELAY_FLAG = '1' THEN
    IF AV_WAITREQUEST = '1' THEN
        nextstate <= UPDATE_PREV_PACKET_PTR_ADDR;
    ELSE
        nextstate <= PREV_PACKET_UPDATE_TAIL_WRITE_RST;
    END IF;
ELSE
    nextstate <= UPDATE_PREV_PACKET_PTR_ADDR;
END IF;


WHEN PREV_PACKET_UPDATE_TAIL_WRITE_RST =>

AV_MEM_ADDR <= SRAM_CTRL_VOSQ_TAIL_ADDR;
sdram_addr_data <= (others => 'Z');
AV_WRITE_DATA <= get_pointer_addr;
ADDR_LOAD_DATA <= (others => 'Z');
OUT_DATA <= (others => 'Z');

prev_packet_addr <= (others => 'Z');
head_tail_addr <= (others => 'Z');
empty_avail_addr <= (others => 'Z');
first_pkt_flag <= '0';

SRAM_IDLE_AREA_INIT_EN <= '0';-- sram idle area initialization address
generator disable
SRAM_QUEUE_AREA_INIT_EN <= '0';-- sram queue area initialization address
generator disable
SRAM_IDLE_AREA_INIT_RSTN <= '0';-- sram idle area initialization address
generator reset disabled
SRAM_QUEUE_AREA_INIT_RSTN <= '0';-- sram queue area initialization address
generator reset disabled
SRAM_RD_WR_TIMER_EN <= '0';-- sram delta delay between read or a write
disabled
SRAM_WR_RQT_TIMER_EN <= '0';-- sram write delay disabled
SRAM_RD_RQT_TIMER_EN <= '0';-- sram read delay disabled
```

```
                DELAY_TIMER_EN <= '0';  -- sdram write delay triggered
                Q_LGTH_CNT_EN <= '0';
                Q_LGTH_CTL <= '1';
                AV_READ_N <= '1';
                AV_WRITE_N <= '1';
                PRE_FIFO_RDREQ <= '0';
                POST_FIFO_WRREQ <= '0';
                AV_FLUSH <= '0';
                PACKET_CNT_EN <= '0';
                PACKET_CNT_RST <= '1';
                ADDR_LOAD <= '0';
                MASTER_MEM_INIT_DONE <= '1';
                out_bus_packet_data <= (others => '0');

                next_op <= '1';  -- for next loop go into read sdram mode
                nextstate <= UPDATE_VOSQ_TAIL_PTR_ADDR;


        WHEN UPDATE_VOSQ_TAIL_PTR_ADDR =>

                AV_MEM_ADDR <= SRAM_CTRL_VOSQ_TAIL_ADDR;
                sdram_addr_data <= (others => 'Z');
                AV_WRITE_DATA <= get_pointer_addr;
                ADDR_LOAD_DATA <= (others => 'Z');
                OUT_DATA <= (others => 'Z');

                prev_packet_addr <= (others => 'Z');
                head_tail_addr <= (others => 'Z');
                empty_avail_addr <= (others => 'Z');
                first_pkt_flag <= '0';

                SRAM_IDLE_AREA_INIT_EN <= '0';-- sram idle area initialization address
generator disable
                SRAM_QUEUE_AREA_INIT_EN <= '0';-- sram queue area initialization address
generator disable
                SRAM_IDLE_AREA_INIT_RSTN <= '0';-- sram idle area initialization address
generator reset disabled
                SRAM_QUEUE_AREA_INIT_RSTN <= '0';-- sram queue area initialization address
generator reset disabled
                SRAM_RD_WR_TIMER_EN <= '0';-- sram delta delay between read or a write
disabled
                SRAM_WR_RQT_TIMER_EN <= '1';-- sram write delay disabled
                SRAM_RD_RQT_TIMER_EN <= '0';-- sram read delay disabled
                DELAY_TIMER_EN <= '0';  -- sdram write delay triggered
                Q_LGTH_CNT_EN <= '0';
                Q_LGTH_CTL <= '1';
                AV_READ_N <= '1';
                AV_WRITE_N <= '0';
                PRE_FIFO_RDREQ <= '0';
                POST_FIFO_WRREQ <= '0';
                AV_FLUSH <= '0';
                PACKET_CNT_EN <= '0';
```

```
                PACKET_CNT_RST <= '1';
                ADDR_LOAD <= '0';
                MASTER_MEM_INIT_DONE <= '1';
                out_bus_packet_data <= (others => '0');

                next_op <= '1';  -- for next loop go into read sdram mode
                IF SRAM_WR_RQT_DELAY_FLAG = '1' THEN
                    IF AV_WAITREQUEST = '1' THEN
                        nextstate <= UPDATE_VOSQ_TAIL_PTR_ADDR;
                    ELSE
                        nextstate <= RQT_SDRAM_IDLE_ADDR;
                    END IF;
                ELSE
                    nextstate <= UPDATE_VOSQ_TAIL_PTR_ADDR;
                END IF;

            WHEN RQT_SDRAM_IDLE_ADDR =>

                AV_MEM_ADDR <= get_pointer_addr;
                sdram_addr_data <= (others => 'Z');
                AV_WRITE_DATA <= (others => 'Z');
                ADDR_LOAD_DATA <= (others => 'Z');
                OUT_DATA <= (others => 'Z');

                prev_packet_addr <= (others => 'Z');
                head_tail_addr <= (others => 'Z');
                empty_avail_addr <= (others => 'Z');
                first_pkt_flag <= '0';

                SRAM_IDLE_AREA_INIT_EN <= '0';-- sram idle area initialization address
generator disable
                SRAM_QUEUE_AREA_INIT_EN <= '0';-- sram queue area initialization address
generator disable
                SRAM_IDLE_AREA_INIT_RSTN <= '0';-- sram idle area initialization address
generator reset disabled
                SRAM_QUEUE_AREA_INIT_RSTN <= '0';-- sram queue area initialization address
generator reset disabled
                SRAM_RD_WR_TIMER_EN <= '0';-- sram delta delay between read or a write
disabled
                SRAM_WR_RQT_TIMER_EN <= '0';-- sram write delay disabled
                SRAM_RD_RQT_TIMER_EN <= '1';-- sram read delay disabled
                DELAY_TIMER_EN <= '0';   -- sdram write delay disabled
                Q_LGTH_CNT_EN <= '0';
                Q_LGTH_CTL <= '1';
                AV_READ_N <= '0';
                AV_WRITE_N <= '1';
                PRE_FIFO_RDREQ <= '0';
                POST_FIFO_WRREQ <= '0';
                AV_FLUSH <= '0';
                PACKET_CNT_EN <= '0';
                PACKET_CNT_RST <= '1';
                ADDR_LOAD <= '0';
```

```
                MASTER_MEM_INIT_DONE <= '1';
                out_bus_packet_data <= (others => '0');


                next_op <= '1';  -- for next loop go into read sdram mode

                IF SRAM_RD_RQT_DELAY_FLAG = '1' THEN
                        nextstate <= CAPTURE_AV_RQT_SDRAM_IDLE_ADDR;
                ELSE
                        nextstate <= RQT_SDRAM_IDLE_ADDR;
                END IF;


        WHEN CAPTURE_AV_RQT_SDRAM_IDLE_ADDR =>

                AV_MEM_ADDR <= get_pointer_addr;
                sdram_addr_data <= AV_READ_DATA;-- Only the MSB 16 bits contains the
SDRAM address
                AV_WRITE_DATA <= (others => 'Z');
                ADDR_LOAD_DATA <= (others => 'Z');
                OUT_DATA <= (others => 'Z');

                prev_packet_addr <= (others => 'Z');
                head_tail_addr <= (others => 'Z');
                empty_avail_addr <= (others => 'Z');
                first_pkt_flag <= '0';

                SRAM_IDLE_AREA_INIT_EN <= '0';-- sram idle area initialization address
generator disable
                SRAM_QUEUE_AREA_INIT_EN <= '0';-- sram queue area initialization address
generator disable
                SRAM_IDLE_AREA_INIT_RSTN <= '0';-- sram idle area initialization address
generator reset disabled
                SRAM_QUEUE_AREA_INIT_RSTN <= '0';-- sram queue area initialization address
generator reset disabled
                SRAM_RD_WR_TIMER_EN <= '0';-- sram delta delay between read or a write
disabled
                SRAM_WR_RQT_TIMER_EN <= '0';-- sram write delay disabled
                SRAM_RD_RQT_TIMER_EN <= '0';-- sram read delay disabled
                DELAY_TIMER_EN <= '0';   -- sdram write delay disabled
                Q_LGTH_CNT_EN <= '0';
                Q_LGTH_CTL <= '1';
                AV_READ_N <= '0';
                AV_WRITE_N <= '1';
                PRE_FIFO_RDREQ <= '0';
                POST_FIFO_WRREQ <= '0';
                AV_FLUSH <= '0';
                PACKET_CNT_EN <= '0';
                PACKET_CNT_RST <= '1';
                ADDR_LOAD <= '0';
                MASTER_MEM_INIT_DONE <= '1';
                out_bus_packet_data <= (others => '0');
```

```
                    next_op <= '1';  -- for next loop go into read sdram mode

                    nextstate <= PRELOAD_ADDR_CNTR;


----- End of State machine section for Writing a packet to a linked-list using SRAM and SDRAM ----------


                WHEN PRELOAD_ADDR_CNTR =>

                    AV_MEM_ADDR <= (others => 'Z');
                    ADDR_LOAD_DATA <= ("0000000000000000" & sdram_addr_data(31 DOWNTO
16));  -- obtained from reading MSB of SRAM linked-list ptr's
                    AV_WRITE_DATA <= (others => 'Z');
                    OUT_DATA <= (others => 'Z');

                    prev_packet_addr <= (others => 'Z');
                    head_tail_addr <= (others => 'Z');
                    empty_avail_addr <= (others => 'Z');
                    get_pointer_addr <= (others => 'Z');

                    SRAM_IDLE_AREA_INIT_EN <= '0';-- sram idle area initialization address
generator disable
                    SRAM_QUEUE_AREA_INIT_EN <= '0';-- sram queue area initialization address
generator disable
                    SRAM_IDLE_AREA_INIT_RSTN <= '0';-- sram idle area initialization address
generator reset disabled
                    SRAM_QUEUE_AREA_INIT_RSTN <= '0';-- sram queue area initialization address
generator reset disabled
                    SRAM_RD_WR_TIMER_EN <= '0';-- sram delta delay between read or a write
disabled
                    SRAM_WR_RQT_TIMER_EN <= '0';-- sram write delay disabled
                    SRAM_RD_RQT_TIMER_EN <= '0';-- sram read delay disabled
                    DELAY_TIMER_EN <= '0';
                    Q_LGTH_CNT_EN <= '0';
                    Q_LGTH_CTL <= '1';
                    AV_READ_N <= '1';
                    AV_WRITE_N <= '1';
                    PRE_FIFO_RDREQ <= '0';
                    POST_FIFO_WRREQ <= '0';
                    AV_FLUSH <= '0';
                    PACKET_CNT_EN <= '0';
                    PACKET_CNT_RST <= '1';
                    ADDR_LOAD <= '1';
                    MASTER_MEM_INIT_DONE <= '1';
                    out_bus_packet_data <= (others => '0');

                    -- On initialization continue to write the first packet and then enter normal
                    -- routine once you get to the INIT state of staying in write mode then arbitrate to
                    -- READ mode after.
```

```
                    IF MASTER_ID = '1' and first_pkt_flag = '1' THEN
                        next_op <= '0';  -- keep to write mode to sdram at initialization
                        nextstate <= READ_RQT_FIFO;
                    ELSE
                        IF rd_wr_op = '1' THEN
                            nextstate <= READ_RQT_FIFO;
                            next_op <= '1';  -- keep to write mode to sdram at initialization
                        ELSE
                            nextstate <= PRELOAD_ADDR_CNTR_READ_DELAY;
                            next_op <= '0';
                        END IF;
                    END IF;

                WHEN PRELOAD_ADDR_CNTR_READ_DELAY =>

                    AV_MEM_ADDR <= (others => 'Z');
                    AV_WRITE_DATA <= (others => 'Z');
                    ADDR_LOAD_DATA <= ("0000000000000000" & sdram_addr_data(31 DOWNTO
16));
                    OUT_DATA <= (others => 'Z');

                    prev_packet_addr <= (others => 'Z');
                    head_tail_addr <= (others => 'Z');
                    empty_avail_addr <= (others => 'Z');
                    get_pointer_addr <= (others => 'Z');
                    first_pkt_flag <= '0';

                    SRAM_IDLE_AREA_INIT_EN <= '0';--  sram  idle  area  initialization  address
generator disable
                    SRAM_QUEUE_AREA_INIT_EN <= '0';-- sram queue area initialization address
generator disable
                    SRAM_IDLE_AREA_INIT_RSTN <= '0';-- sram idle area initialization address
generator reset disabled
                    SRAM_QUEUE_AREA_INIT_RSTN <= '0';-- sram queue area initialization address
generator reset disabled
                    SRAM_RD_WR_TIMER_EN <= '0';-- sram delta delay between read or a write
disabled
                    SRAM_WR_RQT_TIMER_EN <= '0';-- sram write delay disabled
                    SRAM_RD_RQT_TIMER_EN <= '0';-- sram read delay disabled
                    DELAY_TIMER_EN <= '0';
                    Q_LGTH_CNT_EN <= '0';
                    Q_LGTH_CTL <= '1';
                    AV_READ_N <= '1';
                    AV_WRITE_N <= '1';
                    PRE_FIFO_RDREQ <= '0';
                    POST_FIFO_WRREQ <= '0';
                    AV_FLUSH <= '0';
                    PACKET_CNT_EN <= '0';
                    PACKET_CNT_RST <= '1';
                    ADDR_LOAD <= '0';
                    MASTER_MEM_INIT_DONE <= '1';
                    out_bus_packet_data <= (others => '0');
```

```
                nextstate <= READ_SDRAM_DATA;
                next_op <= '0';


        WHEN READ_RQT_FIFO =>

                AV_MEM_ADDR <= (others => 'Z');
                sdram_addr_data <= (others => 'Z');
                AV_WRITE_DATA <= (others => 'Z');
                ADDR_LOAD_DATA <= (others => 'Z');
                OUT_DATA <= (others => 'Z');

                prev_packet_addr <= (others => 'Z');
                head_tail_addr <= (others => 'Z');
                empty_avail_addr <= (others => 'Z');
                get_pointer_addr <= (others => 'Z');

                SRAM_IDLE_AREA_INIT_EN <= '0';-- sram idle area initialization address
generator disable

                SRAM_QUEUE_AREA_INIT_EN <= '0';-- sram queue area initialization address
generator disable

                SRAM_IDLE_AREA_INIT_RSTN <= '0';-- sram idle area initialization address
generator reset disabled

                SRAM_QUEUE_AREA_INIT_RSTN <= '0';-- sram queue area initialization address
generator reset disabled

                SRAM_RD_WR_TIMER_EN <= '0';-- sram delta delay between read or a write
disabled

                SRAM_WR_RQT_TIMER_EN <= '0';-- sram write delay disabled
                SRAM_RD_RQT_TIMER_EN <= '0';-- sram read delay disabled
                DELAY_TIMER_EN <= '0';
                Q_LGTH_CNT_EN <= '0';
                Q_LGTH_CTL <= '1';
                AV_READ_N <= '1';
                AV_WRITE_N <= '1';
                PRE_FIFO_RDREQ <= '1';
                POST_FIFO_WRREQ <= '0';
                AV_FLUSH <= '0';
                PACKET_CNT_EN <= '0';
                PACKET_CNT_RST <= '1';
                ADDR_LOAD <= '0';
                MASTER_MEM_INIT_DONE <= '1';
                out_bus_packet_data <= ("0000000000000000" & IN_DATA);

                -- On initialization continue to write the first packet and then enter normal
                -- routine once you get to the INIT state of staying in write mode then arbitrate to
                -- READ mode after.
                IF MASTER_ID = '1' and first_pkt_flag = '1' THEN
                    next_op <= '0';  -- keep to write mode to sdram at initialization
                ELSE
                    IF rd_wr_op = '1' THEN
                        next_op <= '1';  -- keep to write mode to sdram at initialization
```

```
            ELSE
                next_op <= '0';
            END IF;
        END IF;
        nextstate <= WRITE_SDRAM_DATA;

    WHEN WRITE_SDRAM_DATA =>

        AV_MEM_ADDR <= NEXT_ADDR;       -- Obtain current address for burst
        sdram_addr_data <= (others => 'Z');
        AV_WRITE_DATA <= out_bus_packet_data;
        ADDR_LOAD_DATA <= (others => 'Z');
        OUT_DATA <= (others => 'Z');

        prev_packet_addr <= (others => 'Z');
        head_tail_addr <= (others => 'Z');
        empty_avail_addr <= (others => 'Z');
        get_pointer_addr <= (others => 'Z');

        SRAM_IDLE_AREA_INIT_EN <= '0';-- sram idle area initialization address
generator disable
        SRAM_QUEUE_AREA_INIT_EN <= '0';-- sram queue area initialization address
generator disable
        SRAM_IDLE_AREA_INIT_RSTN <= '0';-- sram idle area initialization address
generator reset disabled
        SRAM_QUEUE_AREA_INIT_RSTN <= '0';-- sram queue area initialization address
generator reset disabled
        SRAM_RD_WR_TIMER_EN <= '0';-- sram delta delay between read or a write
disabled
        SRAM_WR_RQT_TIMER_EN <= '0';-- sram write delay disabled
        SRAM_RD_RQT_TIMER_EN <= '0';-- sram read delay disabled
        DELAY_TIMER_EN <= '1';  -- Trigger the write timer
        Q_LGTH_CTL <= '1';
        AV_READ_N <= '1';
        AV_WRITE_N <= '0';
        PRE_FIFO_RDREQ <= '0';
        POST_FIFO_WRREQ <= '0';
        AV_FLUSH <= '0';
        PACKET_CNT_EN <= '0';
        PACKET_CNT_RST <= '1';
        ADDR_LOAD <= '0';
        MASTER_MEM_INIT_DONE <= '1';

        -- On initialization continue to write the first packet and then enter normal
        -- routine once you get to the INIT state of staying in write mode then arbitrate to
        -- READ mode after.
        IF MASTER_ID = '1' and first_pkt_flag = '1' THEN
            next_op <= '0';  -- keep to write mode to sdram at initialization
            IF END_OF_PACKET = '1' THEN
                first_pkt_flag <= '0';
            ELSE
                first_pkt_flag <= '1';
```

```
            END IF;
        ELSE
            IF rd_wr_op = '1' THEN
                next_op <= '1';  -- keep to write mode to sdram at initialization
            ELSE
                next_op <= '0';
            END IF;
            first_pkt_flag <= '0';
        END IF;

        -- Waitrequest is pulled high once the write is invoked because there is a short delay
        -- when the waitrequest is asserted. This is not instantaneous on the same clock period.
        IF DELAY_FLAG = '1' THEN
            IF AV_WAITREQUEST = '1' THEN
                Q_LGTH_CNT_EN <= '0';
                nextstate <= WRITE_SDRAM_DATA;
            ELSE
                IF END_OF_PACKET = '1' THEN
                    Q_LGTH_CNT_EN <= '1';
                    nextstate <= INIT;
                ELSE
                    Q_LGTH_CNT_EN <= '0';
                nextstate <= INCREMENT_ADDR;
                END IF;
            END IF;
        ELSE
            Q_LGTH_CNT_EN <= '0';
            nextstate <= WRITE_SDRAM_DATA;
        END IF;

    WHEN INCREMENT_ADDR =>

        AV_MEM_ADDR <= (others => 'Z');
        sdram_addr_data <= (others => 'Z');
        AV_WRITE_DATA <= (others => 'Z');
        ADDR_LOAD_DATA <= (others => 'Z');
        OUT_DATA <= (others => 'Z');

        prev_packet_addr <= (others => 'Z');
        head_tail_addr <= (others => 'Z');
        empty_avail_addr <= (others => 'Z');
        get_pointer_addr <= (others => 'Z');
        out_bus_packet_data <= (others => '0');

        SRAM_IDLE_AREA_INIT_EN <= '0';-- sram idle area initialization address
generator disable
        SRAM_QUEUE_AREA_INIT_EN <= '0';-- sram queue area initialization address
generator disable
        SRAM_IDLE_AREA_INIT_RSTN <= '0';-- sram idle area initialization address
generator reset disabled
        SRAM_QUEUE_AREA_INIT_RSTN <= '0';-- sram queue area initialization address
generator reset disabled
```

```
                SRAM_RD_WR_TIMER_EN <= '0';-- sram delta delay between read or a write
disabled
                SRAM_WR_RQT_TIMER_EN <= '0';-- sram write delay disabled
                SRAM_RD_RQT_TIMER_EN <= '0';-- sram read delay disabled
                DELAY_TIMER_EN <= '0';
                Q_LGTH_CNT_EN <= '0';
                Q_LGTH_CTL <= '1';
                AV_READ_N <= '1';
                AV_WRITE_N <= '1';
                PRE_FIFO_RDREQ <= '0';
                POST_FIFO_WRREQ <= '0';
                AV_FLUSH <= '0';
                PACKET_CNT_EN <= '1';
                PACKET_CNT_RST <= '1';
                ADDR_LOAD <= '0';
                MASTER_MEM_INIT_DONE <= '1';


                -- On initialization continue to write the first packet and then enter normal
                -- routine once you get to the INIT state of staying in write mode then arbitrate to
                -- READ mode after.
                IF MASTER_ID = '1' and first_pkt_flag = '1' THEN
                    next_op <= '0';  -- keep to write mode to sdram at initialization
                    first_pkt_flag <= '1';
                    nextstate <= READ_RQT_FIFO;
                ELSE
                    IF rd_wr_op = '1' THEN
                        next_op <= '1';  -- keep to write mode to sdram at initialization
                        nextstate <= READ_RQT_FIFO;
                    ELSE
                        next_op <= '0';
                        nextstate <= PRELOAD_ADDR_CNTR_READ_DELAY;
                    END IF;
                    first_pkt_flag <= '0';
                END IF;


            WHEN READ_SDRAM_DATA =>

                AV_MEM_ADDR <= NEXT_ADDR;       -- Obtain current address for burst
                sdram_addr_data <= (others => 'Z');
                AV_WRITE_DATA <= (others => 'Z');
                ADDR_LOAD_DATA <= (others => 'Z');
                OUT_DATA <=(others => 'Z');

                prev_packet_addr <= (others => 'Z');
                head_tail_addr <= (others => 'Z');
                empty_avail_addr <= (others => 'Z');
                get_pointer_addr <= (others => 'Z');
                first_pkt_flag <= '0';

                SRAM_IDLE_AREA_INIT_EN <= '0';--  sram  idle  area  initialization  address
```

```
generator disable
                    SRAM_QUEUE_AREA_INIT_EN <= '0';-- sram queue area initialization address
generator disable
                    SRAM_IDLE_AREA_INIT_RSTN <= '0';-- sram idle area initialization address
generator reset disabled
                    SRAM_QUEUE_AREA_INIT_RSTN <= '0';-- sram queue area initialization address
generator reset disabled
                    SRAM_RD_WR_TIMER_EN <= '0';-- sram delta delay between read or a write
disabled
                    SRAM_WR_RQT_TIMER_EN <= '0';-- sram write delay disabled
                    SRAM_RD_RQT_TIMER_EN <= '0';-- sram read delay disabled
                    DELAY_TIMER_EN <= '1';
                    Q_LGTH_CTL <= '0';
                    AV_READ_N <= '0';
                    AV_WRITE_N <= '1';
                    PRE_FIFO_RDREQ <= '0';
                    POST_FIFO_WRREQ <= '0';
                    AV_FLUSH <= '0';
                    PACKET_CNT_EN <= '0';
                    PACKET_CNT_RST <= '1';
                    ADDR_LOAD <= '0';
                    MASTER_MEM_INIT_DONE <= '1';
                    out_bus_packet_data <= (others => '0');

                    IF rd_wr_op = '1' THEN
                        next_op <= '1';
                    ELSE
                        next_op <= '0';
                    END IF;

                    -- There is a delay until the waitrequest is pulled high prior to the read
                    -- As a result a delay is introduced to wait until the waitrequest is asserted.
                    IF DELAY_FLAG = '1' THEN
                        IF AV_WAITREQUEST = '1' THEN
                            Q_LGTH_CNT_EN <= '0';
                            nextstate <= READ_SDRAM_DATA;
                        ELSE
                            IF AV_READDATAVALID = '1' THEN
                                Q_LGTH_CNT_EN <= '0';
                                nextstate <= CAPTURE_SDRAM_AV_READ;
                            ELSE
                                Q_LGTH_CNT_EN <= '0';
                                nextstate <= READ_SDRAM_DATA;
                            END IF;
                        END IF;
                    ELSE
                        Q_LGTH_CNT_EN <= '0';
                        nextstate <= READ_SDRAM_DATA;
                    END IF;


                WHEN CAPTURE_SDRAM_AV_READ =>
```

```
                    AV_MEM_ADDR <= NEXT_ADDR;
                    sdram_addr_data <= AV_READ_DATA;
                    AV_WRITE_DATA <= (others => 'Z');
                    ADDR_LOAD_DATA <= (others => 'Z');
                    OUT_DATA <=(others => 'Z');

                    prev_packet_addr <= (others => 'Z');
                    head_tail_addr <= (others => 'Z');
                    empty_avail_addr <= (others => 'Z');
                    get_pointer_addr <= (others => 'Z');
                    first_pkt_flag <= '0';

                    SRAM_IDLE_AREA_INIT_EN <= '0';-- sram idle area initialization address
generator disable
                    SRAM_QUEUE_AREA_INIT_EN <= '0';-- sram queue area initialization address
generator disable
                    SRAM_IDLE_AREA_INIT_RSTN <= '0';-- sram idle area initialization address
generator reset disabled
                    SRAM_QUEUE_AREA_INIT_RSTN <= '0';-- sram queue area initialization address
generator reset disabled
                    SRAM_RD_WR_TIMER_EN <= '0';-- sram delta delay between read or a write
disabled
                    SRAM_WR_RQT_TIMER_EN <= '0';-- sram write delay disabled
                    SRAM_RD_RQT_TIMER_EN <= '0';-- sram read delay disabled
                    DELAY_TIMER_EN <= '0';
                    Q_LGTH_CNT_EN <= '0';
                    Q_LGTH_CTL <= '0';
                    AV_READ_N <= '0';
                    AV_WRITE_N <= '1';
                    PRE_FIFO_RDREQ <= '0';
                    POST_FIFO_WRREQ <= '0';
                    AV_FLUSH <= '0';
                    PACKET_CNT_EN <= '0';
                    PACKET_CNT_RST <= '1';
                    ADDR_LOAD <= '0';
                    MASTER_MEM_INIT_DONE <= '1';
                    out_bus_packet_data <= (others => '0');

                    IF rd_wr_op = '1' THEN
                        next_op <= '1';
                    ELSE
                        next_op <= '0';
                    END IF;

                    IF AV_WAITREQUEST = '1' THEN
                        nextstate <= CAPTURE_SDRAM_AV_READ;
                    ELSE
                        nextstate <= WRITE_TO_OUT_FIFO;
                    END IF;

                WHEN WRITE_TO_OUT_FIFO =>
```

```
AV_MEM_ADDR <= (others => 'Z');
AV_WRITE_DATA <= (others => 'Z');
ADDR_LOAD_DATA <= (others => 'Z');
OUT_DATA <= sdram_addr_data(15 DOWNTO 0);  -- output bus to FIFO designed
to be 16 bits

prev_packet_addr <= (others => 'Z');
head_tail_addr <= (others => 'Z');
empty_avail_addr <= (others => 'Z');
get_pointer_addr <= (others => 'Z');
first_pkt_flag <= '0';

SRAM_IDLE_AREA_INIT_EN <= '0';-- sram idle area initialization address
generator disable
SRAM_QUEUE_AREA_INIT_EN <= '0';-- sram queue area initialization address
generator disable
SRAM_IDLE_AREA_INIT_RSTN <= '0';-- sram idle area initialization address
generator reset disabled
SRAM_QUEUE_AREA_INIT_RSTN <= '0';-- sram queue area initialization address
generator reset disabled
SRAM_RD_WR_TIMER_EN <= '0';-- sram delta delay between read or a write
disabled
SRAM_WR_RQT_TIMER_EN <= '0';-- sram write delay disabled
SRAM_RD_RQT_TIMER_EN <= '0';-- sram read delay disabled
DELAY_TIMER_EN <= '0';
Q_LGTH_CTL <= '0';
AV_READ_N <= '1';
AV_WRITE_N <= '1';
PRE_FIFO_RDREQ <= '0';
POST_FIFO_WRREQ <= '1';
AV_FLUSH <= '0';
PACKET_CNT_EN <= '0';
PACKET_CNT_RST <= '1';
ADDR_LOAD <= '0';
MASTER_MEM_INIT_DONE <= '1';
out_bus_packet_data <= (others => '0');

IF rd_wr_op = '1' THEN
    next_op <= '1';
ELSE
    next_op <= '0';
END IF;

IF END_OF_PACKET = '1' THEN
    Q_LGTH_CNT_EN <= '1';
    nextstate <= INIT;
ELSE
    Q_LGTH_CNT_EN <= '0';
    nextstate <= INCREMENT_ADDR;
END IF;
```

```
        END CASE;
    END PROCESS rw_control;

END ARCHITECTURE main_rtl;
```

E.10 Ext Mem Tristate Bridge.VHD

This module is a slave control peripheral used to communicate with the Avalon Bus and the the external asynchronous SRAM memory. The objective of this is to allow the external SRAM to provide and to receive data from the Avalon Bus that would be supplied to it. The state machine is derived from the Avalon Specification document (chapiter 7 - Tristate Transfers)

```
-- Library Clause
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
use ieee.std_logic_arith.all;

-- Entity Declaration
ENTITY EXT_MEM_TRISTATE_BRIDGE IS
    PORT(
            CLK                 : IN STD_LOGIC;  -- System clock
            RESET               : IN STD_LOGIC;  -- Reset state machine (active high)
            AV_READ_N           : IN STD_LOGIC;  -- Read Request from the Avalon Bus (active low)
            AV_WRITE_N          : IN STD_LOGIC;  -- Write Request from the Avalon Bus (active low)
          AV_CHIPSELECT_N : IN STD_LOGIC;  -- Avalon Address & Read Request Decode Signal from the
Avalon Bus (active high)
            AV_ADDRESS : IN STD_LOGIC_VECTOR (19 DOWNTO 0);  -- Address from the Avalon Bus
            IN_DATA       : IN STD_LOGIC_VECTOR (31 DOWNTO 0);  -- Data from the Avalon Bus

            AV_WAITREQUEST : OUT  STD_LOGIC;  -- Stalls the Avalon Bus until slave is ready to transmit to
Avalon Bus  (active high)
            OUT_DATA          : OUT  STD_LOGIC_VECTOR (31 DOWNTO 0);  -- Data to the Avalon Bus
            MEM_ADDRESS     : OUT  STD_LOGIC_VECTOR (17 DOWNTO 0);  -- Address to the off-chip
memory
            MEM_DATA      : INOUT STD_LOGIC_VECTOR (31 DOWNTO 0);  -- Data to the off-chip memory
            MEM_BYTEENABLE_N : OUT  STD_LOGIC_VECTOR (3 DOWNTO 0);  -- Memory Byte enable
(BHE_n_2 = 3, BLE_n_2 = 2, BHE_n_1 = 1, BLE_n_1 = 0)
            MEM_OE_N       : OUT  STD_LOGIC;  -- Memory tristate output enable
            MEM_CS_N       : OUT  STD_LOGIC;  -- Memory chipselect
            MEM_WE_N       : OUT  STD_LOGIC  -- Memory write enable
);

END ENTITY EXT_MEM_TRISTATE_BRIDGE;

ARCHITECTURE main_rtl OF EXT_MEM_TRISTATE_BRIDGE IS
-- Signal declarations.
signal sram_data_oe   : std_logic;
signal r_av_write_data : std_logic_vector(31 downto 0);
```

```
signal sram_access    : std_logic;
signal r_sram_access  : std_logic;
signal sram_data_in   : std_logic_vector(31 downto 0);
signal be_settings    : std_logic_vector(3 downto 0);

BEGIN


-------------------------------------------------------------------------------
-- SRAM control
-------------------------------------------------------------------------------
sram_controller : process (clk, reset)
    VARIABLE set_hold_delay  :INTEGER RANGE 0 to 15;
begin
    if (reset = '1') then

            MEM_CS_N      <= '1';
            MEM_WE_N      <= '1';
            MEM_OE_N      <= '1';
            MEM_ADDRESS   <= (others => '0');
            AV_WAITREQUEST <= '0';
            sram_data_in  <= (others => '0');
            sram_data_oe  <= '0';
            r_av_write_data <= (others => '0');
            sram_access   <= '0';
            r_sram_access <= '0';
            be_settings       <= "1111";
            set_hold_delay:= 0;

    elsif clk'event and clk = '1' then   -- rising clock edge

            r_av_write_data <= IN_DATA;
            sram_data_in  <= MEM_DATA;
            MEM_ADDRESS   <= AV_ADDRESS(17 downto 0);
            MEM_CS_N      <= '1';
            MEM_WE_N      <= '1';
            MEM_OE_N      <= '1';
            sram_data_oe  <= '0';
            AV_WAITREQUEST <= '0';
            sram_access   <= '0';
            r_sram_access <= '0';
            be_settings       <= "0000";


            if (AV_CHIPSELECT_N = '0') then --(AV_ADDRESS(31 downto 24) = x"01" and AV_ADDRESS(23
downto 22) = "00" and AV_CHIPSELECT_N = '0') then
                    MEM_CS_N      <= '0';
                    sram_data_oe  <= not AV_WRITE_N;
                    sram_access   <= not sram_access;
                    r_sram_access <= sram_access;
                    AV_WAITREQUEST <= '1';
```

```
                    if (AV_READ_N = '0') then
                        if (sram_access = '0') then
                        MEM_OE_N      <= AV_READ_N;
                    end if;
                end if;

                if ((sram_access = '1' and r_sram_access = '0')) then
                        AV_WAITREQUEST <= '0';
                        MEM_WE_N      <= AV_WRITE_N;
                end if;

    end if;
end if;
end process sram_controller;

-------------------------------------------------------------------------------
-- Async Assignments.
-------------------------------------------------------------------------------
MEM_BYTEENABLE_N <= be_settings;
MEM_DATA <= r_av_write_data when (sram_data_oe = '1') else (others => 'Z');
OUT_DATA <= sram_data_in;

END ARCHITECTURE main_rtl;
```

E.11 VOSQ Queue Length.VHD

This module counts the total number of packets entered within a particular VOSQ. Any packet entered in SDRAM, or extracted from the SDRAM is used in this measure. A 17-bit counter will be used which is based on the total SRAM size / 2. An overflow is identified when the total number of packets reaches a threshold. Here the threshold = NUM_OF_PACKETS.

```
-- Library Clause
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;


-- Entity Declaration
ENTITY vosq_queue_length IS
    GENERIC(
            NUM_OF_PACKETS:POSITIVE := 131072;
            OVERFLOW_THR:POSITIVE := 131072);
    PORT(
            CLK             :IN STD_LOGIC;
            RST             :IN STD_LOGIC;
            CNT_EN          :IN STD_LOGIC;
            INS_REM_CTL     :IN STD_LOGIC;

            OVERFLOW_FLAG:OUT STD_LOGIC;
```

```
        TOTAL_PACKETS:OUT INTEGER RANGE 0 to (NUM_OF_PACKETS-1)
    );
END ENTITY vosq_queue_length;

ARCHITECTURE main_cntr_rtl OF vosq_queue_length IS

BEGIN

    PROCESS (CLK, RST)
        VARIABLE count  :INTEGER RANGE 0 to (NUM_OF_PACKETS-1);
    BEGIN

        IF (RST = '0') THEN
            count := 0;
        ELSIF (CLK'EVENT AND CLK = '1') THEN
            IF (CNT_EN = '1' AND (count /= (NUM_OF_PACKETS-1) OR count >= 0)) THEN
                IF (INS_REM_CTL = '1') THEN
                    count := count + 1;
                ELSE
                    count := count - 1;
                END IF;
            ELSE
                count := count;
            END IF;
        END IF;
        TOTAL_PACKETS <= count;

        -- Identify the overflow condition.
        IF count >= (OVERFLOW_THR-1) THEN
            OVERFLOW_FLAG <= '1';
        ELSE
            OVERFLOW_FLAG <= '0';
        END IF ;

    END PROCESS;
END ARCHITECTURE main_cntr_rtl;
```

E.12 Tb SOPC Memory RW VHDL.VHD

This module is the test bench file that is used within Modelsim (by Cadence). The test bench file has all the libraries required from Altera SOPC builder which automatically generated the glue logic required to connect both the Master Input Logic and the slave peripherals to the Avalon Bus. The code generation was done using Altera Quartus development simulation software. The VHDL module that represents the entire linked-list memory manager is identified as SOPC_SDRAM_RW.VHD which also consists of the SDRAM memory controller that is also integrated. Other modules included in the test bench that were extracted from the Altera library are both the Prestore and Prefetch FIFOs. These VHDL files were automatically generated and the top-level entities were imported as components into this file. Both the SRAM and SDRAM memory model was included into test bench for connectivity with the linked-list memory manager.

```
-- Library Clause
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE std.textio.ALL;
USE ieee.std_logic_unsigned.ALL;
USE ieee.std_logic_arith.ALL;
USE ieee.vital_timing.ALL;
USE ieee.vital_primitives.ALL;

LIBRARY lpm;
USE lpm.lpm_components.all;

LIBRARY altera_mf;
USE altera_mf.altera_mf_components.all;

LIBRARY work;

LIBRARY FMF;
USE FMF.gen_utils.ALL;
USE FMF.conversions.ALL;


-- Entity Declaration
ENTITY tb_SOPC_Memory_RW_VHDL IS
END ENTITY tb_SOPC_Memory_RW_VHDL;

ARCHITECTURE main_behvrl OF tb_SOPC_Memory_RW_VHDL IS

-- Declaration of signals & components for test bench
  COMPONENT idt71V416 IS
    GENERIC (
      -- tipd delays: interconnect path delays
      tipd_OENeg       : VitalDelayType01 := VitalZeroDelay01;
      tipd_WENeg       : VitalDelayType01 := VitalZeroDelay01;
      tipd_CENeg       : VitalDelayType01 := VitalZeroDelay01;
      tipd_BHENeg      : VitalDelayType01 := VitalZeroDelay01;
      tipd_BLENeg      : VitalDelayType01 := VitalZeroDelay01;
      tipd_D0          : VitalDelayType01 := VitalZeroDelay01;
      tipd_D1          : VitalDelayType01 := VitalZeroDelay01;
      tipd_D2          : VitalDelayType01 := VitalZeroDelay01;
      tipd_D3          : VitalDelayType01 := VitalZeroDelay01;
      tipd_D4          : VitalDelayType01 := VitalZeroDelay01;
      tipd_D5          : VitalDelayType01 := VitalZeroDelay01;
      tipd_D6          : VitalDelayType01 := VitalZeroDelay01;
      tipd_D7          : VitalDelayType01 := VitalZeroDelay01;
      tipd_D8          : VitalDelayType01 := VitalZeroDelay01;
      tipd_D9          : VitalDelayType01 := VitalZeroDelay01;
      tipd_D10         : VitalDelayType01 := VitalZeroDelay01;
      tipd_D11         : VitalDelayType01 := VitalZeroDelay01;
      tipd_D12         : VitalDelayType01 := VitalZeroDelay01;
      tipd_D13         : VitalDelayType01 := VitalZeroDelay01;
```

```
    tipd_D14        : VitalDelayType01 := VitalZeroDelay01;
    tipd_D15        : VitalDelayType01 := VitalZeroDelay01;
    tipd_A0         : VitalDelayType01 := VitalZeroDelay01;
    tipd_A1         : VitalDelayType01 := VitalZeroDelay01;
    tipd_A2         : VitalDelayType01 := VitalZeroDelay01;
    tipd_A3         : VitalDelayType01 := VitalZeroDelay01;
    tipd_A4         : VitalDelayType01 := VitalZeroDelay01;
    tipd_A5         : VitalDelayType01 := VitalZeroDelay01;
    tipd_A6         : VitalDelayType01 := VitalZeroDelay01;
    tipd_A7         : VitalDelayType01 := VitalZeroDelay01;
    tipd_A8         : VitalDelayType01 := VitalZeroDelay01;
    tipd_A9         : VitalDelayType01 := VitalZeroDelay01;
    tipd_A10        : VitalDelayType01 := VitalZeroDelay01;
    tipd_A11        : VitalDelayType01 := VitalZeroDelay01;
    tipd_A12        : VitalDelayType01 := VitalZeroDelay01;
    tipd_A13        : VitalDelayType01 := VitalZeroDelay01;
    tipd_A14        : VitalDelayType01 := VitalZeroDelay01;
    tipd_A15        : VitalDelayType01 := VitalZeroDelay01;
    tipd_A16        : VitalDelayType01 := VitalZeroDelay01;
    tipd_A17        : VitalDelayType01 := VitalZeroDelay01;
    -- tpd delays
    tpd_BLENeg_D0     : VitalDelayType01Z := UnitDelay01Z;
    tpd_OENeg_D0      : VitalDelayType01Z := UnitDelay01Z;
    tpd_CENeg_D0      : VitalDelayType01Z := UnitDelay01Z;
    tpd_A0_D0       : VitalDelayType01  := UnitDelay01;
    -- tpw values: pulse widths
    tpw_WENeg_negedge  : VitalDelayType   := UnitDelay;
    -- tsetup values: setup times
    tsetup_BLENeg_WENeg : VitalDelayType   := UnitDelay;
    tsetup_D0_WENeg    : VitalDelayType   := UnitDelay;
    tsetup_D0_CENeg    : VitalDelayType   := UnitDelay;
    -- thold values: hold times
    thold_D0_WENeg     : VitalDelayType   := UnitDelay;
    thold_D0_CENeg     : VitalDelayType   := UnitDelay;
    -- generic control parameters
    InstancePath      : STRING   := DefaultInstancePath;
    TimingChecksOn    : BOOLEAN  := DefaultTimingChecks;
    MsgOn             : BOOLEAN  := DefaultMsgOn;
    XOn               : BOOLEAN  := DefaultXOn;
    SeverityMode      : SEVERITY_LEVEL := WARNING;
    -- For FMF SDF technology file usage
    TimingModel       : STRING   := DefaultTimingModel
  );
  PORT(
    A0          : IN   std_ulogic := 'U';
    A1          : IN   std_ulogic := 'U';
    A2          : IN   std_ulogic := 'U';
    A3          : IN   std_ulogic := 'U';
    A4          : IN   std_ulogic := 'U';
    A5          : IN   std_ulogic := 'U';
    A6          : IN   std_ulogic := 'U';
    A7          : IN   std_ulogic := 'U';
```

```
    A8          : IN   std_ulogic := 'U';
    A9          : IN   std_ulogic := 'U';
    A10         : IN   std_ulogic := 'U';
    A11         : IN   std_ulogic := 'U';
    A12         : IN   std_ulogic := 'U';
    A13         : IN   std_ulogic := 'U';
    A14         : IN   std_ulogic := 'U';
    A15         : IN   std_ulogic := 'U';
    A16         : IN   std_ulogic := 'U';
    A17         : IN   std_ulogic := 'U';
    D0          : INOUT std_ulogic := 'U';
    D1          : INOUT std_ulogic := 'U';
    D2          : INOUT std_ulogic := 'U';
    D3          : INOUT std_ulogic := 'U';
    D4          : INOUT std_ulogic := 'U';
    D5          : INOUT std_ulogic := 'U';
    D6          : INOUT std_ulogic := 'U';
    D7          : INOUT std_ulogic := 'U';
    D8          : INOUT std_ulogic := 'U';
    D9          : INOUT std_ulogic := 'U';
    D10         : INOUT std_ulogic := 'U';
    D11         : INOUT std_ulogic := 'U';
    D12         : INOUT std_ulogic := 'U';
    D13         : INOUT std_ulogic := 'U';
    D14         : INOUT std_ulogic := 'U';
    D15         : INOUT std_ulogic := 'U';
    BHENeg      : IN   std_ulogic := 'U';
    BLENeg      : IN   std_ulogic := 'U';
    OENeg       : IN   std_ulogic := 'U';
    WENeg       : IN   std_ulogic := 'U';
    CENeg       : IN   std_ulogic := 'U'
  );
END COMPONENT idt71V416;


COMPONENT mt48lc4m32b2 IS
  GENERIC (
    -- Timing Parameters for -75 (PC133) and CL = 3
    tAC     : TIME   :=  5.4 ns;
    tHZ     : TIME   :=  5.4 ns;
    tOH     : TIME   :=  2.7 ns;
    tMRD    : INTEGER :=  2;        -- 2 Clk Cycles
    tRAS    : TIME   := 44.0 ns;
    tRC     : TIME   := 66.0 ns;
    tRCD    : TIME   := 20.0 ns;
    tRFC    : TIME   := 66.0 ns;
    tRP     : TIME   := 20.0 ns;
    tRRD    : TIME   := 15.0 ns;
    tWRa    : TIME   :=  7.5 ns;    -- Auto precharge
    tWRm    : TIME   := 15.0 ns;    -- Manual Precharge

    tAH     : TIME   :=  0.8 ns;
```

```
    tAS     : TIME   := 1.5 ns;
    tCH     : TIME   := 2.5 ns;
    tCL     : TIME   := 2.5 ns;
    tCK     : TIME   := 7.5 ns;
    tDH     : TIME   := 0.8 ns;
    tDS     : TIME   := 1.5 ns;
    tCKH    : TIME   := 0.8 ns;
    tCKS    : TIME   := 1.5 ns;
    tCMH    : TIME   := 0.8 ns;
    tCMS    : TIME   := 1.5 ns;

    addr_bits : INTEGER := 12;
    data_bits : INTEGER := 32;
    col_bits  : INTEGER := 8
  );
  PORT (
   Dq     : INOUT STD_LOGIC_VECTOR (data_bits - 1 DOWNTO 0) := (OTHERS => 'Z');
    Addr   : IN  STD_LOGIC_VECTOR (addr_bits - 1 DOWNTO 0) := (OTHERS => '0');
    Ba     : IN  STD_LOGIC_VECTOR (1 DOWNTO 0) := "00";
    Clk    : IN  STD_LOGIC := '0';
    Cke    : IN  STD_LOGIC := '1';
    Cs_n   : IN  STD_LOGIC := '1';
    Ras_n  : IN  STD_LOGIC := '1';
    Cas_n  : IN  STD_LOGIC := '1';
    We_n   : IN  STD_LOGIC := '1';
    Dqm    : IN  STD_LOGIC_VECTOR (3 DOWNTO 0) := "0000"
  );
END COMPONENT mt48lc4m32b2;

COMPONENT SOPC_SDRAM_RW IS
  PORT (
    -- 1) global signals:
    clk : IN STD_LOGIC;
    reset_n : IN STD_LOGIC;

    -- the_ext_sram_interface
    mem_address_from_the_ext_sram_interface : OUT STD_LOGIC_VECTOR (17 DOWNTO 0);
    mem_byteenable_n_from_the_ext_sram_interface : OUT STD_LOGIC_VECTOR (3 DOWNTO 0);
    mem_cs_n_from_the_ext_sram_interface : OUT STD_LOGIC;
    mem_data_to_and_from_the_ext_sram_interface : INOUT STD_LOGIC_VECTOR (31 DOWNTO 0);
    mem_oe_n_from_the_ext_sram_interface : OUT STD_LOGIC;
    mem_we_n_from_the_ext_sram_interface : OUT STD_LOGIC;

    -- the_sdram_mt48lc2m32b2
    zs_addr_from_the_sdram_mt48lc4m32b2 : OUT STD_LOGIC_VECTOR (11 DOWNTO 0);
    zs_ba_from_the_sdram_mt48lc4m32b2 : OUT STD_LOGIC_VECTOR (1 DOWNTO 0);
    zs_cas_n_from_the_sdram_mt48lc4m32b2 : OUT STD_LOGIC;
    zs_cke_from_the_sdram_mt48lc4m32b2 : OUT STD_LOGIC;
    zs_cs_n_from_the_sdram_mt48lc4m32b2 : OUT STD_LOGIC;
    zs_dq_to_and_from_the_sdram_mt48lc4m32b2 : INOUT STD_LOGIC_VECTOR (31 DOWNTO 0);
    zs_dqm_from_the_sdram_mt48lc4m32b2 : OUT STD_LOGIC_VECTOR (3 DOWNTO 0);
    zs_ras_n_from_the_sdram_mt48lc4m32b2 : OUT STD_LOGIC;
```

```
        zs_we_n_from_the_sdram_mt48lc4m32b2 : OUT STD_LOGIC;

        -- the_vosq0_addr_generator
        ext_get_store_ctl_from_the_vosq0_addr_generator : OUT STD_LOGIC;
        ext_ptr_en_from_the_vosq0_addr_generator : OUT STD_LOGIC;
        ptr_addr_data_to_the_vosq0_addr_generator : IN STD_LOGIC_VECTOR (31 DOWNTO 0);

        -- the_vosq0_fifo_logic
        addr_load_data_from_the_vosq0_fifo_logic : OUT STD_LOGIC_VECTOR (31 DOWNTO 0);
        addr_load_from_the_vosq0_fifo_logic : OUT STD_LOGIC;
        delay_flag_to_the_vosq0_fifo_logic : IN STD_LOGIC;
        delay_timer_en_from_the_vosq0_fifo_logic : OUT STD_LOGIC;
        end_of_packet_to_the_vosq0_fifo_logic : IN STD_LOGIC;
        in_data_to_the_vosq0_fifo_logic : IN STD_LOGIC_VECTOR (31 DOWNTO 0);
        master_id_to_the_vosq0_fifo_logic : IN STD_LOGIC;
        master_mem_init_done_from_the_vosq0_fifo_logic : OUT STD_LOGIC;
        mem_init_compl_to_the_vosq0_fifo_logic : IN STD_LOGIC;
        next_addr_to_the_vosq0_fifo_logic : IN STD_LOGIC_VECTOR (31 DOWNTO 0);
        out_data_from_the_vosq0_fifo_logic : OUT STD_LOGIC_VECTOR (31 DOWNTO 0);
        packet_cnt_en_from_the_vosq0_fifo_logic : OUT STD_LOGIC;
        packet_cnt_rst_from_the_vosq0_fifo_logic : OUT STD_LOGIC;
        post_fifo_full_to_the_vosq0_fifo_logic : IN STD_LOGIC;
        post_fifo_wrreq_from_the_vosq0_fifo_logic : OUT STD_LOGIC;
        pre_fifo_aempty_to_the_vosq0_fifo_logic : IN STD_LOGIC;
        pre_fifo_empty_to_the_vosq0_fifo_logic : IN STD_LOGIC;
        pre_fifo_rdreq_from_the_vosq0_fifo_logic : OUT STD_LOGIC;
        q_lgth_cnt_en_from_the_vosq0_fifo_logic : OUT STD_LOGIC;
        q_lgth_ctl_from_the_vosq0_fifo_logic : OUT STD_LOGIC;
        sram_idle_area_addr_data_to_the_vosq0_fifo_logic : IN STD_LOGIC_VECTOR (31 DOWNTO 0);
        sram_idle_area_addr_to_the_vosq0_fifo_logic : IN STD_LOGIC_VECTOR (31 DOWNTO 0);
        sram_idle_area_done_to_the_vosq0_fifo_logic : IN STD_LOGIC;
        sram_idle_area_init_en_from_the_vosq0_fifo_logic : OUT STD_LOGIC;
        sram_idle_area_init_rstn_from_the_vosq0_fifo_logic : OUT STD_LOGIC;
        sram_queue_area_addr_data_to_the_vosq0_fifo_logic : IN STD_LOGIC_VECTOR (31 DOWNTO 0);
        sram_queue_area_addr_to_the_vosq0_fifo_logic : IN STD_LOGIC_VECTOR (31 DOWNTO 0);
        sram_queue_area_done_to_the_vosq0_fifo_logic : IN STD_LOGIC;
        sram_queue_area_init_en_from_the_vosq0_fifo_logic : OUT STD_LOGIC;
        sram_queue_area_init_rstn_from_the_vosq0_fifo_logic : OUT STD_LOGIC;
        sram_rd_rqt_delay_flag_to_the_vosq0_fifo_logic : IN STD_LOGIC;
        sram_rd_rqt_timer_en_from_the_vosq0_fifo_logic : OUT STD_LOGIC;
        sram_rd_wr_delay_flag_to_the_vosq0_fifo_logic : IN STD_LOGIC;
        sram_rd_wr_timer_en_from_the_vosq0_fifo_logic : OUT STD_LOGIC;
        sram_wr_rqt_delay_flag_to_the_vosq0_fifo_logic : IN STD_LOGIC;
        sram_wr_rqt_timer_en_from_the_vosq0_fifo_logic : OUT STD_LOGIC

    );
END COMPONENT SOPC_SDRAM_RW;

  COMPONENT packet_counter_VHDL IS
        GENERIC(
        DATAWIDTH:INTEGER:= 2
    );
```

```
        PORT(
                CLK             :IN STD_LOGIC;
                RQT_EN          :IN STD_LOGIC;
                RST             :IN STD_LOGIC;

                PKT_END_FLAG:OUT STD_LOGIC;
                PACKET_NUMBER:OUT INTEGER RANGE 0 TO (DATAWIDTH -1)
        );
    END COMPONENT packet_counter_VHDL;

  COMPONENT SDRAM_Packet_Addr_Counter IS
      PORT(
            CLK             :IN STD_LOGIC;
            RQT_EN          :IN STD_LOGIC;
            RST             :IN STD_LOGIC;
            PRE_LOAD:IN STD_LOGIC;
            LOADED_ADDR:IN STD_LOGIC_VECTOR(31 DOWNTO 0);

            OUT_ADDR:OUT STD_LOGIC_VECTOR(31 DOWNTO 0)
      );
  END COMPONENT SDRAM_Packet_Addr_Counter;

COMPONENT vosq0_prestore_fifo IS
      PORT
      (
            data            :IN STD_LOGIC_VECTOR (31 DOWNTO 0);
            wrreq           :IN STD_LOGIC;
            rdreq           :IN STD_LOGIC;
            rdclk           :IN STD_LOGIC;
            wrclk           :IN STD_LOGIC;
            aclr            :IN STD_LOGIC;
            q               :OUT STD_LOGIC_VECTOR (31 DOWNTO 0);
            rdempty         :OUT STD_LOGIC ;
            wrfull          :OUT STD_LOGIC
      );
  END COMPONENT vosq0_prestore_fifo;

  COMPONENT packet_generator IS
      GENERIC(
            NUM_OF_WORDS:POSITIVE := 8);
      PORT(
            CLK             :IN STD_LOGIC;
            RQT_EN          :IN STD_LOGIC;
            RST             :IN STD_LOGIC;

            OUT_ADDR:OUT INTEGER RANGE 0 TO (NUM_OF_WORDS-1)
      );
  END COMPONENT packet_generator;

  COMPONENT Write_SDRAM_Ctrlr_Delay IS
    GENERIC(
            WRITE_DELAY:POSITIVE :=2
```

```
            );
    PORT(
            CLK                     :IN STD_LOGIC;
            TIMER_EN_N :IN STD_LOGIC;

            CNT_OUT         :OUT INTEGER RANGE 0 to 15;
            DELAY_FLAG:OUT STD_LOGIC
          );
END COMPONENT Write_SDRAM_Ctrlr_Delay;

COMPONENT SRAM_READ_DELAY IS
    GENERIC(
            READ_DELAY:POSITIVE :=1
            );
    PORT(
            CLK                     :IN STD_LOGIC;
            TIMER_EN_N :IN STD_LOGIC;

            CNT_OUT         :OUT INTEGER RANGE 0 to 15;
            DELAY_FLAG:OUT STD_LOGIC
    );
END COMPONENT SRAM_READ_DELAY;

COMPONENT SDRAM_Memory_Filler IS
    PORT(
            CLK                 :IN STD_LOGIC;
            MEM_CNT_EN          :IN STD_LOGIC;
            RST_N               :IN STD_LOGIC;

            MEM_SEQ_COMPL_FLAG:OUT STD_LOGIC;
            MEM_FILL_ADDR   :OUT STD_LOGIC_VECTOR(31 DOWNTO 0);
            OUT_ADDR            :OUT STD_LOGIC_VECTOR(31 DOWNTO 0)
    );
END COMPONENT SDRAM_Memory_Filler;

COMPONENT SRAM_Memory_Filler IS
    PORT(
            CLK                 :IN STD_LOGIC;
            MEM_CNT_EN          :IN STD_LOGIC;
            RST_N               :IN STD_LOGIC;

            MEM_SEQ_COMPL_FLAG:OUT STD_LOGIC;
            MEM_FILL_ADDR   :OUT STD_LOGIC_VECTOR(31 DOWNTO 0);
            OUT_ADDR            :OUT STD_LOGIC_VECTOR(31 DOWNTO 0)
    );
END COMPONENT SRAM_Memory_Filler;


COMPONENT vosq_queue_length IS
    GENERIC(
            NUM_OF_PACKETS:POSITIVE := 131072;
            OVERFLOW_THR:POSITIVE := 131072);
```

```
    PORT(
            CLK                     :IN STD_LOGIC;
            RST                     :IN STD_LOGIC;
            CNT_EN              :IN STD_LOGIC;
            INS_REM_CTL:IN STD_LOGIC;

            OVERFLOW_FLAG:OUT STD_LOGIC;
            TOTAL_PACKETS:OUT INTEGER RANGE 0 to (NUM_OF_PACKETS-1)
    );
END COMPONENT vosq_queue_length;

COMPONENT free_memory_addr_counter IS
    GENERIC (NUM_OF_BITS : POSITIVE :=32);
    PORT(
            CLK                 :IN STD_LOGIC;
            RST_N               :IN STD_LOGIC;
            PTR_EN              :IN STD_LOGIC;
            GET_STORE_CTL:IN STD_LOGIC;

            EMPTY_FLAG:OUT STD_LOGIC;
            FULL_FLAG   :OUT STD_LOGIC;
            GET_ADDR_PTR  :OUT  STD_LOGIC_VECTOR (NUM_OF_BITS-1 DOWNTO 0);
            STORE_ADDR_PTR:OUT  STD_LOGIC_VECTOR (NUM_OF_BITS-1 DOWNTO 0);
            OUT_ADDR      :OUT  STD_LOGIC_VECTOR (NUM_OF_BITS-1 DOWNTO 0)
    );
END COMPONENT free_memory_addr_counter;


-- Define constants
constant CLK_PERIOD1 :time := 20 ns;
constant DELAY :time := 15 ns;
constant FIFO_SIZE :integer := 65536;
constant NO_OF_PKT_FGMTS :integer := 2;
constant VOSQ0_TOTAL_NO_OF_PKTS :integer := 131072;
constant VOSQ0_PKT_OVERFLOW_THR :integer := 131072;
constant VOSQ_IDLE_ADDR_N0_OF_BITS :integer := 32;

-- Common Inputs to the VOSQ0 packet counter model
signal vosq0_pkt_cnt_clk, vosq0_pkt_cnt_en, vosq0_pkt_cnt_rstn :std_logic;
signal vosq0_pkt_cnt_end_flag :std_logic;

-- Outputs from the VOSQ0 packet counter model
signal vosq0_pkt_cnt_output :integer range 0 to 1;

-- Common Inputs to the SDRAM Address Generator model
signal sdram_addr_gen_clk, sdram_addr_gen_en, sdram_addr_gen_rst, sdram_addr_gen_preload :std_logic;
signal sdram_addr_gen_loaded_addr :std_logic_vector(31 downto 0);

-- Outputs from the SDRAM Address Generator model
signal sdram_addr_gen_out_addr :std_logic_vector(31 downto 0);

-- Common Inputs to VOSQ0 packet monitor or comparator between generator & fifo
```

```
-- output
signal vosq0_comp_clk:std_logic;

-- Output from the VOSQ0 packet monitor
   signal    vosq0_comp_out :std_logic;

-- Common Inputs to VOSQ0 prestore fifo
   signal vosq0_pre_fifo_data:std_logic_vector(15 DOWNTO 0);
   signal    vosq0_pre_fifo_wrreq:std_logic;
   signal vosq0_pre_fifo_rdreq:std_logic;
   signal vosq0_pre_fifo_rdclk:std_logic;
   signal vosq0_pre_fifo_wrclk:std_logic;
   signal vosq0_pre_fifo_aclr:std_logic;

-- Outputs for the VOSQ0 prestore fifo
   signal vosq0_pre_fifo_q:std_logic_vector(31 DOWNTO 0);
   signal vosq0_pre_fifo_rdempty:std_logic;
   signal vosq0_pre_fifo_wrfull:std_logic;

-- Common Inputs to VOSQ0 prefetch fifo
   signal vosq0_post_fifo_data:std_logic_vector(31 DOWNTO 0);
   signal    vosq0_post_fifo_wrreq:std_logic;
   signal vosq0_post_fifo_rdreq:std_logic;
   signal vosq0_post_fifo_rdclk:std_logic;
   signal vosq0_post_fifo_wrclk:std_logic;
   signal vosq0_post_fifo_aclr:std_logic;

-- Outputs for the VOSQ0 prefetch fifo
   signal vosq0_post_fifo_q:std_logic_vector(31 DOWNTO 0);
   signal vosq0_post_fifo_rdempty:std_logic;
   signal vosq0_post_fifo_wrfull:std_logic;


-- Common Inputs to the Packet Generator
signal pkt_gen_clk, pkt_gen_en, pkt_gen_rst :std_logic;


-- Outputs for the Packet Generator
signal pkt_gen_dout:integer range 0 to (FIFO_SIZE-1);

-- Common Inputs to the Write SDRAM Delay Timer
signal write_delay_clk  :std_logic;
signal write_delay_timer_en_n :std_logic;

-- Common Outputs to the Write SDRAM Delay Timer
signal write_delay_cnt_out :integer range 0 to 15;
signal write_delay_flag:std_logic;

-- Common Inputs to the SRAM Read and Write Delay Request Timer
signal sram_wr_rqt_delay_clk  :std_logic;
signal sram_wr_rqt_delay_timer_en :std_logic;
signal sram_rd_rqt_delay_clk  :std_logic;
```

```
signal sram_rd_rqt_delay_timer_en :std_logic;

-- Common Outputs to the SRAM Read and Write Delay Request Timer
signal sram_wr_rqt_delay_cnt_out :integer range 0 to 15;
signal sram_wr_rqt_delay_flag:std_logic;
signal sram_rd_rqt_delay_cnt_out :integer range 0 to 15;
signal sram_rd_rqt_delay_flag:std_logic;

-- Common Inputs to the SRAM Read and Write Back-to-Back Delay Timer
signal sram_rd_wr_delay_clk  :std_logic;
signal sram_rd_wr_delay_timer_en :std_logic;

-- Common Outputs to the SRAM Read and Write Back-to-Back Delay Timer
signal sram_rd_wr_delay_cnt_out :integer range 0 to 15;
signal sram_rd_wr_delay_flag:std_logic;

-- Common Inputs to the vosq0 packet queue length counter
   signal vosq0_queue_lgth_clk     :std_logic;
   signal vosq0_queue_lgth_rst     :std_logic;
   signal vosq0_queue_lgth_cnt_en  :std_logic;
   signal vosq0_queue_lgth_ctl     :std_logic;

-- Common Outputs to the vosq0 packet queue length counter
   signal vosq0_queue_lgth_ovfl    :std_logic;
   signal vosq0_queue_lgth_pkt_cnt  :integer range 0 to (VOSQ0_TOTAL_NO_OF_PKTS-1);

-- Common Inputs to the vosq idle address generator
   signal vosq_idle_addr_gen_clk          :std_logic;
   signal vosq_idle_addr_gen_rstn         :std_logic;
   signal vosq_idle_addr_gen_ptr_en       :std_logic;
   signal vosq_idle_addr_gen_get_store_ctl  :std_logic;

-- Common Outputs to the vosq idle address generator
   signal vosq_idle_addr_gen_empty_flag    :std_logic;
   signal vosq_idle_addr_gen_full_flag     :std_logic;
   signal vosq_idle_addr_gen_get_addr_ptr  :std_logic_vector (VOSQ_IDLE_ADDR_N0_OF_BITS-1 DOWNTO
0);
   signal    vosq_idle_addr_gen_store_addr_ptr     :std_logic_vector     (VOSQ_IDLE_ADDR_N0_OF_BITS-1
DOWNTO 0);
   signal vosq_idle_addr_gen_out_addr_ptr  :std_logic_vector (VOSQ_IDLE_ADDR_N0_OF_BITS-1 DOWNTO
0);


-- Common Inputs to the SRAM Queue Filler Address Generator
   signal sram_queue_filler_clk          :std_logic;
   signal sram_queue_filler_mem_cnt_en     :std_logic;
   signal sram_queue_filler_rst_n        :std_logic;

-- Common Outputs to the SRAM Queue Filler Address Generator
   signal sram_queue_filler_mem_seq_compl_flag  :std_logic;
   signal sram_queue_filler_mem_fill_addr      :std_logic_vector(31 DOWNTO 0);
   signal sram_queue_filler_out_addr           :std_logic_vector(31 DOWNTO 0);
```

```
-- Common Inputs to the SRAM Idle Filler Address Generator
   signal sram_idle_filler_clk          :std_logic;
   signal sram_idle_filler_mem_cnt_en   :std_logic;
   signal sram_idle_filler_rst_n        :std_logic;

-- Common Outputs to the SRAM Idle Filler Address Generator
   signal sram_idle_filler_mem_seq_compl_flag  :std_logic;
   signal sram_idle_filler_mem_fill_addr       :std_logic_vector(31 DOWNTO 0);
   signal sram_idle_filler_out_addr            :std_logic_vector(31 DOWNTO 0);


-- Common Inputs to the SDRAM memory model
signal sdram_clk  :std_logic;
signal sdram_addr :std_logic_vector(11 DOWNTO 0);
signal sdram_ba   :std_logic_vector(1 DOWNTO 0);
signal sdram_cke  :std_logic;
signal sdram_csn  :std_logic;
signal sdram_rasn :std_logic;
signal sdram_casn :std_logic;
signal sdram_wen  :std_logic;
signal sdram_dqm  :std_logic_vector(3 DOWNTO 0);

-- Common Outputs to the SDRAM memory model
signal sdram_dq   :std_logic_vector(31 DOWNTO 0);

-- Common Inputs to the SRAM memory model
--**use To_StdULogicVector (std_logic_vector) to convert to U-logic**
signal sram1_addr       :std_logic_vector(17 DOWNTO 0);
signal sram1_din        :std_logic_vector(15 DOWNTO 0);
signal sram1_BHENeg     :std_logic;
signal sram1_BLENeg     :std_logic;
signal sram1_OENeg      :std_logic;
signal sram1_WENeg      :std_logic;
signal sram1_CENeg      :std_logic;
signal sram2_addr       :std_logic_vector(17 DOWNTO 0);
signal sram2_din        :std_logic_vector(15 DOWNTO 0);
signal sram2_BHENeg     :std_logic;
signal sram2_BLENeg     :std_logic;
signal sram2_OENeg      :std_logic;
signal sram2_WENeg      :std_logic;
signal sram2_CENeg      :std_logic;
signal sram_BE_n        :std_logic_vector(3 DOWNTO 0);
signal sram_32bit_data_in  :std_logic_vector(31 DOWNTO 0);
signal sram_32bit_data_out :std_logic_vector(31 DOWNTO 0);


-- Common Inputs to the SOPC design
signal sopc_vosq0_rstn                :std_logic;
signal sopc_vosq0_clk                 :std_logic;
signal sopc_vosq0_in_data             :std_logic_vector(31 DOWNTO 0);
signal sopc_vosq0_prefifo_rdempty     :std_logic;
```

```
signal sopc_vosq0_prefifo_aempty          :std_logic;
signal sopc_vosq0_next_addr               :std_logic_vector(31 DOWNTO 0);
signal sopc_vosq0_pkt_blk_end             :std_logic;
signal sopc_vosq0_delay_flag              :std_logic;
signal sopc_vosq0_sram_wr_rqt_delay_flag  :std_logic;
signal sopc_vosq0_sram_rd_rqt_delay_flag  :std_logic;
signal sopc_vosq0_sram_rd_wr_delay_flag   :std_logic;
signal sopc_vosq0_postfifo_wrfull         :std_logic;
signal sopc_vosq0_master_id               :std_logic;
signal sopc_vosq0_master_mem_init_done_input :std_logic;

-- Common Outputs to the SOPC design
signal sopc_vosq0_prefifo_rdreq           :std_logic;
signal sopc_vosq0_pkt_cnt_en              :std_logic;
signal sopc_vosq0_pkt_cnt_rst             :std_logic;
signal sopc_vosq0_addr_load_data          :std_logic_vector(31 DOWNTO 0);
signal sopc_vosq0_addr_load               :std_logic;
signal sopc_vosq0_delay_timer_en          :std_logic;
signal sopc_vosq0_sram_wr_rqt_delay_timer_en  :std_logic;
signal sopc_vosq0_sram_rd_rqt_delay_timer_en  :std_logic;
signal sopc_vosq0_sram_rd_wr_delay_timer_en   :std_logic;
signal sopc_vosq0_q_lgth_cnt_en           :std_logic;
signal sopc_vosq0_q_lgth_ctl              :std_logic;
signal sopc_vosq0_q_lgth_rst              :std_logic;
signal sopc_vosq0_out_data                :std_logic_vector(31 DOWNTO 0);
signal sopc_vosq0_postfifo_wrreq          :std_logic;
signal sopc_vosq0_master_mem_init_done    :std_logic;

signal sopc_vosq0_be_n_to_sram            :std_logic_vector(3 DOWNTO 0);
signal sopc_vosq0_ext_mem_address         :std_logic_vector(17 DOWNTO 0);
signal sopc_vosq0_ext_mem_data            :std_logic_vector(31 DOWNTO 0);
signal sopc_vosq0_oe_n_to_sram            :std_logic;
signal sopc_vosq0_cs_n_to_sram            :std_logic;
signal sopc_vosq0_we_n_to_sram            :std_logic;

-- Global clock
signal global_clk :std_logic;

BEGIN

-- Assign Global Clock to interfaces
sdram_clk <= global_clk; -- (PLL Output #5 from FPGA)
vosq0_pkt_cnt_clk <= global_clk;
vosq0_comp_clk <= global_clk;
vosq0_pre_fifo_rdclk <= global_clk;
vosq0_pre_fifo_wrclk <= global_clk;
pkt_gen_clk <= global_clk;
sdram_addr_gen_clk <= global_clk;
sopc_vosq0_clk <= global_clk;
write_delay_clk <= global_clk;
vosq0_queue_lgth_clk <= global_clk;
  vosq0_post_fifo_rdclk<= global_clk;
```

```
    vosq0_post_fifo_wrclk <= global_clk;
  sram_wr_rqt_delay_clk <= global_clk;
  sram_rd_rqt_delay_clk <= global_clk;
  sram_rd_wr_delay_clk <= global_clk;
  vosq_idle_addr_gen_clk <= global_clk;
    sram_queue_filler_clk <= global_clk;
    sram_idle_filler_clk <= global_clk;

  -- Global reset conditions for modules
vosq0_pre_fifo_aclr <= '1', '0' after (CLK_PERIOD1 * 5);  -- vosq0 prestore fifo
  pkt_gen_rst <= '1', '0' after (CLK_PERIOD1 * 5);  -- packet generator
  sdram_addr_gen_rst <= '1', '0' after (CLK_PERIOD1 * 5);  -- sdram address generator
  sopc_vosq0_rstn <= '0', '1' after (CLK_PERIOD1 * 5);  -- VOSQ0 master peripheral
  vosq0_queue_lgth_rst <= '0', '1' after (CLK_PERIOD1 * 5);  -- VOSQ0 queue length counter
  vosq0_post_fifo_aclr <= '1', '0' after (CLK_PERIOD1 * 5);  -- vosq0 prefetch fifo
  vosq_idle_addr_gen_rstn <= '0', '1' after (CLK_PERIOD1 * 5);  -- idle address generator

  -- Global clock generation (differnet duty cycles are possble)
  clkgen: PROCESS
  BEGIN
    global_clk <= '0';
    wait for (CLK_PERIOD1/2);
    global_clk <= '1';
    wait for (CLK_PERIOD1/2);

  END PROCESS clkgen;


  stimulus1: PROCESS

BEGIN
      wait until (global_clk'event) and (global_clk = '1');

    vosq0_pre_fifo_wrreq <= '0';
    pkt_gen_en <= '0';

     -- Allow the VOSQ0 Master Peripheral to be the controlling Master
     sopc_vosq0_master_id <= '1';

     if vosq0_post_fifo_rdempty /= '1' then
       vosq0_post_fifo_rdreq <= '1';
     else
       vosq0_post_fifo_rdreq <= '0';
     end if;

    -- Delay for a period of 5 clock cycles
     for k in 0 to 5 loop
       wait until (global_clk'event) and (global_clk = '1');
     end loop;

     -- enable the packet generator to fill prestore FIFO
     -- enable the prestore FIFO to write packets
```

```
    pkt_gen_en <= '1' after DELAY;
    vosq0_pre_fifo_wrreq <= '1' after DELAY;

    wait;  -- suspend the process indefinitely & don't re-execute the process.


END PROCESS stimulus1;


packet_gen0: packet_generator
     GENERIC MAP(
    NUM_OF_WORDS=> FIFO_SIZE)
     PORT MAP(
              CLK => pkt_gen_clk,
              RQT_EN => pkt_gen_en,
              RST => pkt_gen_rst,
              OUT_ADDR => pkt_gen_dout
     );


packet_mon0: packet_comparator_VHDL
      GENERIC MAP(
              DATAWIDTH=> 16)
      PORT MAP(
     CLK => vosq0_comp_clk,
              RD_EN => vosq0_pre_fifo_rdreq,
    FIFO_OUTPUT => vosq0_pre_fifo_q,
              COMP_OUT => vosq0_comp_out
      );

write_fifo_vosq0: vosq0_prestore_fifo
      PORT MAP(
              data          => vosq0_pre_fifo_data,
              wrreq         => vosq0_pre_fifo_wrreq,
              rdreq         => vosq0_pre_fifo_rdreq,
              rdclk         => vosq0_pre_fifo_rdclk,
       wrclk          => vosq0_pre_fifo_wrclk,
              aclr          => vosq0_pre_fifo_aclr,
              q             => vosq0_pre_fifo_q,
              rdempty => vosq0_pre_fifo_rdempty,
              wrfull        => vosq0_pre_fifo_wrfull
      );

vosq0_pre_fifo_data <= conv_std_logic_vector(pkt_gen_dout, 16);

read_fifo_vosq0: vosq0_prestore_fifo
      PORT MAP(
              data          => vosq0_post_fifo_data,
              wrreq         => vosq0_post_fifo_wrreq,
              rdreq         => vosq0_post_fifo_rdreq,
              rdclk         => vosq0_post_fifo_rdclk,
       wrclk          => vosq0_post_fifo_wrclk,
```

```
            aclr            => vosq0_post_fifo_aclr,
            q               => vosq0_post_fifo_q,
            rdempty => vosq0_post_fifo_rdempty,
            wrfull          => vosq0_post_fifo_wrfull
    );


sdram_addr_cnt0: SDRAM_Packet_Addr_Counter
    PORT MAP(
            CLK       => sdram_addr_gen_clk,
            RQT_EN    => sdram_addr_gen_en,
            RST       => sdram_addr_gen_rst,
            PRE_LOAD  => sdram_addr_gen_preload,
            LOADED_ADDR=> sdram_addr_gen_loaded_addr,
            OUT_ADDR  => sdram_addr_gen_out_addr
    );

  vosq0_pkt_frag_cntr0: packet_counter_VHDL
          GENERIC MAP(
      DATAWIDTH=> NO_OF_PKT_FGMTS)
          PORT MAP(
                  CLK                 => vosq0_pkt_cnt_clk,
                  RQT_EN          => vosq0_pkt_cnt_en,
                  RST                 => vosq0_pkt_cnt_rstn,
                  PKT_END_FLAG  => vosq0_pkt_cnt_end_flag,
                  PACKET_NUMBER => vosq0_pkt_cnt_output
          );

vosq0_write_sdram_timer0: Write_SDRAM_Ctrlr_Delay
  GENERIC MAP(
            WRITE_DELAY=> 2)
    PORT MAP(
            CLK       => write_delay_clk,
            TIMER_EN_N  => write_delay_timer_en_n,
            CNT_OUT     => write_delay_cnt_out,
            DELAY_FLAG  => write_delay_flag
            );

vosq0_sram_wr_rqt_timer0: SRAM_READ_DELAY
    GENERIC MAP(
            READ_DELAY => 1)
    PORT MAP(
            CLK       => sram_wr_rqt_delay_clk,
            TIMER_EN_N  => sram_wr_rqt_delay_timer_en,

            CNT_OUT     => sram_wr_rqt_delay_cnt_out,
            DELAY_FLAG  => sram_wr_rqt_delay_flag
            );

vosq0_sram_rd_rqt_timer0: SRAM_READ_DELAY
    GENERIC MAP(
            READ_DELAY => 2)
```

```
    PORT MAP(
            CLK         => sram_rd_rqt_delay_clk,
            TIMER_EN_N  => sram_rd_rqt_delay_timer_en,

            CNT_OUT     => sram_rd_rqt_delay_cnt_out,
            DELAY_FLAG  => sram_rd_rqt_delay_flag
            );

vosq0_sram_rd_wr_timer0: SRAM_READ_DELAY
    GENERIC MAP(
            READ_DELAY => 3)
    PORT MAP(
            CLK         => sram_rd_wr_delay_clk,
            TIMER_EN_N  => sram_rd_wr_delay_timer_en,

            CNT_OUT     => sram_rd_wr_delay_cnt_out,
            DELAY_FLAG  => sram_rd_wr_delay_flag
            );

sram_queue_filler0: SDRAM_Memory_Filler
    PORT MAP(
            CLK         => sram_queue_filler_clk,
            MEM_CNT_EN  => sram_queue_filler_mem_cnt_en,
            RST_N       => sram_queue_filler_rst_n,

            MEM_SEQ_COMPL_FLAG=> sram_queue_filler_mem_seq_compl_flag,
            MEM_FILL_ADDR   => sram_queue_filler_mem_fill_addr,
            OUT_ADDR            => sram_queue_filler_out_addr
    );

sram_idle_filler0: SRAM_Memory_Filler
    PORT MAP(
            CLK         => sram_idle_filler_clk,
            MEM_CNT_EN  => sram_idle_filler_mem_cnt_en,
            RST_N       => sram_idle_filler_rst_n,

            MEM_SEQ_COMPL_FLAG=> sram_idle_filler_mem_seq_compl_flag,
            MEM_FILL_ADDR   => sram_idle_filler_mem_fill_addr,
            OUT_ADDR            => sram_idle_filler_out_addr
    );


vosq0_queue_lgth0: vosq_queue_length
    GENERIC MAP(
            NUM_OF_PACKETS=> VOSQ0_TOTAL_NO_OF_PKTS,
            OVERFLOW_THR=> VOSQ0_PKT_OVERFLOW_THR)
    PORT MAP(
            CLK => vosq0_queue_lgth_clk,
            RST => vosq0_queue_lgth_rst,
            CNT_EN => vosq0_queue_lgth_cnt_en,
            INS_REM_CTL => vosq0_queue_lgth_ctl,
```

```
            OVERFLOW_FLAG => vosq0_queue_lgth_ovfl,
            TOTAL_PACKETS => vosq0_queue_lgth_pkt_cnt
    );

vosq_idle_addr: free_memory_addr_counter
    GENERIC MAP(NUM_OF_BITS => 32)
    PORT MAP(
            CLK   => vosq_idle_addr_gen_clk,
            RST_N => vosq_idle_addr_gen_rstn,
            PTR_EN=> vosq_idle_addr_gen_ptr_en,
            GET_STORE_CTL=> vosq_idle_addr_gen_get_store_ctl,

            EMPTY_FLAG=> vosq_idle_addr_gen_empty_flag,
            FULL_FLAG=> vosq_idle_addr_gen_full_flag,
            GET_ADDR_PTR => vosq_idle_addr_gen_get_addr_ptr,
            STORE_ADDR_PTR=> vosq_idle_addr_gen_store_addr_ptr,
            OUT_ADDR => vosq_idle_addr_gen_out_addr_ptr
    );


sdram0: mt48lc4m32b2
  GENERIC MAP(
    -- Timing Parameters for -75 (PC133) and CL = 3
    tAC => 5.4 ns,
    tHZ => 5.4 ns,
    tOH => 2.7 ns,
    tMRD => 2,        -- 2 Clk Cycles
    tRAS => 44.0 ns,
    tRC => 66.0 ns,
    tRCD => 20.0 ns,
    tRFC => 66.0 ns,
    tRP => 20.0 ns,
    tRRD => 15.0 ns,
    tWRa => 7.5 ns,    -- Auto precharge
    tWRm => 15.0 ns,   -- Manual Precharge

    tAH => 0.8 ns,
    tAS => 1.5 ns,
    tCH  => 2.5 ns,
    tCL => 2.5 ns,
    tCK => 7.5 ns,
    tDH => 0.8 ns,
    tDS => 1.5 ns,
    tCKH => 0.8 ns,
    tCKS => 1.5 ns,
    tCMH => 0.8 ns,
    tCMS => 1.5 ns,

    addr_bits => 12,
    data_bits => 32,
    col_bits => 8
  )
```

```
    PORT MAP(
      Dq  => sdram_dq,
--      Dq_out => sdram_dq,
--      Dq_dir => sdram_dqdir,
      Addr => sdram_addr,
      Ba => sdram_ba,
      Clk => sdram_clk,
      Cke => sdram_cke,
      Cs_n => sdram_csn,
      Ras_n => sdram_rasn,
      Cas_n => sdram_casn,
      We_n => sdram_wen,
      Dqm => sdram_dqm
    );


  sram0: idt71V416
    GENERIC MAP(
      -- tipd delays: interconnect path delays
      tipd_OENeg => VitalZeroDelay01,
      tipd_WENeg => VitalZeroDelay01,
      tipd_CENeg => VitalZeroDelay01,
      tipd_BHENeg => VitalZeroDelay01,
      tipd_BLENeg => VitalZeroDelay01,
      tipd_D0 => VitalZeroDelay01,
      tipd_D2 => VitalZeroDelay01,
      tipd_D3 => VitalZeroDelay01,
      tipd_D4 => VitalZeroDelay01,
      tipd_D5 => VitalZeroDelay01,
      tipd_D6 => VitalZeroDelay01,
      tipd_D7 => VitalZeroDelay01,
      tipd_D8 => VitalZeroDelay01,
      tipd_D9 => VitalZeroDelay01,
      tipd_D10 => VitalZeroDelay01,
      tipd_D11 => VitalZeroDelay01,
      tipd_D12 => VitalZeroDelay01,
      tipd_D13 => VitalZeroDelay01,
      tipd_D14 => VitalZeroDelay01,
      tipd_D15 => VitalZeroDelay01,
      tipd_A0 => VitalZeroDelay01,
      tipd_A1 => VitalZeroDelay01,
      tipd_A2 => VitalZeroDelay01,
      tipd_A3 => VitalZeroDelay01,
      tipd_A4 => VitalZeroDelay01,
      tipd_A5 => VitalZeroDelay01,
      tipd_A6 => VitalZeroDelay01,
      tipd_A7 => VitalZeroDelay01,
      tipd_A8 => VitalZeroDelay01,
      tipd_A9 => VitalZeroDelay01,
      tipd_A10 => VitalZeroDelay01,
      tipd_A11 => VitalZeroDelay01,
      tipd_A12 => VitalZeroDelay01,
```

```
    tipd_A13 => VitalZeroDelay01,
    tipd_A14 => VitalZeroDelay01,
    tipd_A15 => VitalZeroDelay01,
    tipd_A16 => VitalZeroDelay01,
    tipd_A17 => VitalZeroDelay01,
    -- tpd delays
    tpd_BLENeg_D0 => UnitDelay01Z,
    tpd_OENeg_D0  => UnitDelay01Z,
    tpd_CENeg_D0  => UnitDelay01Z,
    tpd_A0_D0 => UnitDelay01,
    -- tpw values: pulse widths
    tpw_WENeg_negedge => UnitDelay,
    -- tsetup values: setup times
    tsetup_BLENeg_WENeg => UnitDelay,
    tsetup_D0_WENeg => UnitDelay,
    tsetup_D0_CENeg => UnitDelay,
    -- thold values: hold times
    thold_D0_WENeg => UnitDelay,
    thold_D0_CENeg => UnitDelay,
    -- generic control parameters
    InstancePath => DefaultInstancePath,
    TimingChecksOn => DefaultTimingChecks,
    MsgOn => DefaultMsgOn,
    XOn => DefaultXOn,
    SeverityMode => WARNING,
    -- For FMF SDF technology file usage
    TimingModel => DefaultTimingModel
  )
  PORT MAP(
    A0 => sram1_addr(0),
    A1 => sram1_addr(1),
    A2 => sram1_addr(2),
    A3 => sram1_addr(3),
    A4 => sram1_addr(4),
    A5 => sram1_addr(5),
    A6 => sram1_addr(6),
    A7 => sram1_addr(7),
    A8 => sram1_addr(8),
    A9 => sram1_addr(9),
    A10 => sram1_addr(10),
    A11 => sram1_addr(11),
    A12 => sram1_addr(12),
    A13 => sram1_addr(13),
    A14 => sram1_addr(14),
    A15 => sram1_addr(15),
    A16 => sram1_addr(16),
    A17 => sram1_addr(17),
    D0 => sopc_vosq0_ext_mem_data(0),
    D1 => sopc_vosq0_ext_mem_data(1),
    D2 => sopc_vosq0_ext_mem_data(2),
    D3 => sopc_vosq0_ext_mem_data(3),
    D4 => sopc_vosq0_ext_mem_data(4),
```

```
      D5 => sopc_vosq0_ext_mem_data(5),
      D6 => sopc_vosq0_ext_mem_data(6),
      D7 => sopc_vosq0_ext_mem_data(7),
      D8 => sopc_vosq0_ext_mem_data(8),
      D9 => sopc_vosq0_ext_mem_data(9),
      D10 => sopc_vosq0_ext_mem_data(10),
      D11 => sopc_vosq0_ext_mem_data(11),
      D12 => sopc_vosq0_ext_mem_data(12),
      D13 => sopc_vosq0_ext_mem_data(13),
      D14 => sopc_vosq0_ext_mem_data(14),
      D15 => sopc_vosq0_ext_mem_data(15),
      BHENeg => sram1_BHENeg,
      BLENeg => sram1_BLENeg,
      OENeg => sram1_OENeg,
      WENeg => sram1_WENeg,
      CENeg => sram1_CENEg
    );

  sram1: idt71V416
    GENERIC MAP(
      -- tipd delays: interconnect path delays
      tipd_OENeg => VitalZeroDelay01,
      tipd_WENeg => VitalZeroDelay01,
      tipd_CENeg => VitalZeroDelay01,
      tipd_BHENeg => VitalZeroDelay01,
      tipd_BLENeg => VitalZeroDelay01,
      tipd_D0 => VitalZeroDelay01,
      tipd_D1 => VitalZeroDelay01,
      tipd_D2 => VitalZeroDelay01,
      tipd_D3 => VitalZeroDelay01,
      tipd_D4 => VitalZeroDelay01,
      tipd_D5 => VitalZeroDelay01,
      tipd_D6 => VitalZeroDelay01,
      tipd_D7 => VitalZeroDelay01,
      tipd_D8 => VitalZeroDelay01,
      tipd_D9 => VitalZeroDelay01,
      tipd_D10 => VitalZeroDelay01,
      tipd_D11 => VitalZeroDelay01,
      tipd_D12 => VitalZeroDelay01,
      tipd_D13 => VitalZeroDelay01,
      tipd_D14 => VitalZeroDelay01,
      tipd_D15 => VitalZeroDelay01,
      tipd_A0 => VitalZeroDelay01,
      tipd_A1 => VitalZeroDelay01,
      tipd_A2 => VitalZeroDelay01,
      tipd_A3 => VitalZeroDelay01,
      tipd_A4 => VitalZeroDelay01,
      tipd_A5 => VitalZeroDelay01,
      tipd_A6 => VitalZeroDelay01,
      tipd_A7 => VitalZeroDelay01,
      tipd_A8 => VitalZeroDelay01,
      tipd_A9 => VitalZeroDelay01,
```

```
    tipd_A10 => VitalZeroDelay01,
    tipd_A11 => VitalZeroDelay01,
    tipd_A12 => VitalZeroDelay01,
    tipd_A13 => VitalZeroDelay01,
    tipd_A14 => VitalZeroDelay01,
    tipd_A15 => VitalZeroDelay01,
    tipd_A16 => VitalZeroDelay01,
    tipd_A17 => VitalZeroDelay01,
    -- tpd delays
    tpd_BLENeg_D0 => UnitDelay01Z,
    tpd_OENeg_D0  => UnitDelay01Z,
    tpd_CENeg_D0  => UnitDelay01Z,
    tpd_A0_D0 => UnitDelay01,
    -- tpw values: pulse widths
    tpw_WENeg_negedge => UnitDelay,
    -- tsetup values: setup times
    tsetup_BLENeg_WENeg => UnitDelay,
    tsetup_D0_WENeg => UnitDelay,
    tsetup_D0_CENeg => UnitDelay,
    -- thold values: hold times
    thold_D0_WENeg => UnitDelay,
    thold_D0_CENeg => UnitDelay,
    -- generic control parameters
    InstancePath => DefaultInstancePath,
    TimingChecksOn => DefaultTimingChecks,
    MsgOn => DefaultMsgOn,
    XOn => DefaultXOn,
    SeverityMode => WARNING,
    -- For FMF SDF technology file usage
    TimingModel => DefaultTimingModel
  )
  PORT MAP(
    A0 => sram2_addr(0),
    A1 => sram2_addr(1),
    A2 => sram2_addr(2),
    A3 => sram2_addr(3),
    A4 => sram2_addr(4),
    A5 => sram2_addr(5),
    A6 => sram2_addr(6),
    A7 => sram2_addr(7),
    A8 => sram2_addr(8),
    A9 => sram2_addr(9),
    A10 => sram2_addr(10),
    A11 => sram2_addr(11),
    A12 => sram2_addr(12),
    A13 => sram2_addr(13),
    A14 => sram2_addr(14),
    A15 => sram2_addr(15),
    A16 => sram2_addr(16),
    A17 => sram2_addr(17),
    D0 => sopc_vosq0_ext_mem_data(16),
    D1 => sopc_vosq0_ext_mem_data(17),
```

```
      D2 => sopc_vosq0_ext_mem_data(18),
      D3 => sopc_vosq0_ext_mem_data(19),
      D4 => sopc_vosq0_ext_mem_data(20),
      D5 => sopc_vosq0_ext_mem_data(21),
      D6 => sopc_vosq0_ext_mem_data(22),
      D7 => sopc_vosq0_ext_mem_data(23),
      D8 => sopc_vosq0_ext_mem_data(24),
      D9 => sopc_vosq0_ext_mem_data(25),
      D10 => sopc_vosq0_ext_mem_data(26),
      D11 => sopc_vosq0_ext_mem_data(27),
      D12 => sopc_vosq0_ext_mem_data(28),
      D13 => sopc_vosq0_ext_mem_data(29),
      D14 => sopc_vosq0_ext_mem_data(30),
      D15 => sopc_vosq0_ext_mem_data(31),
      BHENeg => sram2_BHENeg,
      BLENeg => sram2_BLENeg,
      OENeg => sram2_OENeg,
      WENeg => sram2_WENeg,
      CENeg => sram2_CENEg
    );

-- Connection as per the h/w schematic of the EP1S10 StratixI NIOS Development Board
-- and as per the recommendations of the Avalon Spec concerning the connection
-- of external peripherals for address byte offsets.
sram2_addr <= sram1_addr;
sram2_BHENeg <= sram_BE_n(3);
sram2_BLENeg <= sram_BE_n(2);
sram2_OENeg <= sram1_OENeg;
sram2_WENeg <= sram1_WENeg;
sram2_CENeg <= sram1_CENeg;
sram1_BHENeg <= sram_BE_n(1);
sram1_BLENeg <= sram_BE_n(0);

sram_BE_n <= sopc_vosq0_be_n_to_sram;
sram1_addr <= sopc_vosq0_ext_mem_address;
sram1_OENeg <= sopc_vosq0_oe_n_to_sram;
sram1_CENeg <= sopc_vosq0_cs_n_to_sram;
sram1_WENeg <= sopc_vosq0_we_n_to_sram;


sopc_sdram_rw0: SOPC_SDRAM_RW
  PORT MAP(
    clk => sopc_vosq0_clk,
    reset_n => sopc_vosq0_rstn,

    -- the_ext_sram_interface
    mem_address_from_the_ext_sram_interface => sopc_vosq0_ext_mem_address,
    mem_byteenable_n_from_the_ext_sram_interface => sopc_vosq0_be_n_to_sram,
    mem_cs_n_from_the_ext_sram_interface => sopc_vosq0_cs_n_to_sram,
    mem_data_to_and_from_the_ext_sram_interface => sopc_vosq0_ext_mem_data,
    mem_oe_n_from_the_ext_sram_interface => sopc_vosq0_oe_n_to_sram,
    mem_we_n_from_the_ext_sram_interface => sopc_vosq0_we_n_to_sram,
```

```
-- the_sdram_mt48lc2m32b2
zs_addr_from_the_sdram_mt48lc4m32b2 => sdram_addr,
zs_ba_from_the_sdram_mt48lc4m32b2 => sdram_ba,
zs_cas_n_from_the_sdram_mt48lc4m32b2 => sdram_casn,
zs_cke_from_the_sdram_mt48lc4m32b2 => sdram_cke,
zs_cs_n_from_the_sdram_mt48lc4m32b2 => sdram_csn,
zs_dq_to_and_from_the_sdram_mt48lc4m32b2 => sdram_dq,
zs_dqm_from_the_sdram_mt48lc4m32b2 => sdram_dqm,
zs_ras_n_from_the_sdram_mt48lc4m32b2 => sdram_rasn,
zs_we_n_from_the_sdram_mt48lc4m32b2 => sdram_wen,

-- the_vosq0_addr_generator
ext_get_store_ctl_from_the_vosq0_addr_generator => vosq_idle_addr_gen_get_store_ctl,
ext_ptr_en_from_the_vosq0_addr_generator => vosq_idle_addr_gen_ptr_en,
ptr_addr_data_to_the_vosq0_addr_generator => vosq_idle_addr_gen_out_addr_ptr,

-- the_vosq0_fifo_logic
addr_load_data_from_the_vosq0_fifo_logic =>  sopc_vosq0_addr_load_data,
addr_load_from_the_vosq0_fifo_logic => sopc_vosq0_addr_load,
delay_flag_to_the_vosq0_fifo_logic => sopc_vosq0_delay_flag,
delay_timer_en_from_the_vosq0_fifo_logic => sopc_vosq0_delay_timer_en,
end_of_packet_to_the_vosq0_fifo_logic => sopc_vosq0_pkt_blk_end,
in_data_to_the_vosq0_fifo_logic => sopc_vosq0_in_data,
master_id_to_the_vosq0_fifo_logic => sopc_vosq0_master_id,
master_mem_init_done_from_the_vosq0_fifo_logic => sopc_vosq0_master_mem_init_done_input,
mem_init_compl_to_the_vosq0_fifo_logic => sopc_vosq0_master_mem_init_done,
next_addr_to_the_vosq0_fifo_logic => sopc_vosq0_next_addr,
out_data_from_the_vosq0_fifo_logic => sopc_vosq0_out_data,
packet_cnt_en_from_the_vosq0_fifo_logic => sopc_vosq0_pkt_cnt_en,
packet_cnt_rst_from_the_vosq0_fifo_logic => sopc_vosq0_pkt_cnt_rst,
post_fifo_full_to_the_vosq0_fifo_logic => sopc_vosq0_postfifo_wrfull,
post_fifo_wrreq_from_the_vosq0_fifo_logic => sopc_vosq0_postfifo_wrreq,
pre_fifo_aempty_to_the_vosq0_fifo_logic => sopc_vosq0_prefifo_aempty,
pre_fifo_empty_to_the_vosq0_fifo_logic => sopc_vosq0_prefifo_rdempty,
pre_fifo_rdreq_from_the_vosq0_fifo_logic => sopc_vosq0_prefifo_rdreq,
q_lgth_cnt_en_from_the_vosq0_fifo_logic => sopc_vosq0_q_lgth_cnt_en,
sram_idle_area_addr_data_to_the_vosq0_fifo_logic => sram_idle_filler_out_addr,
sram_idle_area_addr_to_the_vosq0_fifo_logic => sram_idle_filler_mem_fill_addr,
sram_idle_area_done_to_the_vosq0_fifo_logic => sram_idle_filler_mem_seq_compl_flag,
sram_idle_area_init_en_from_the_vosq0_fifo_logic => sram_idle_filler_mem_cnt_en,
sram_idle_area_init_rstn_from_the_vosq0_fifo_logic => sram_idle_filler_rst_n,
sram_queue_area_addr_data_to_the_vosq0_fifo_logic => sram_queue_filler_out_addr,
sram_queue_area_addr_to_the_vosq0_fifo_logic => sram_queue_filler_mem_fill_addr,
sram_queue_area_done_to_the_vosq0_fifo_logic => sram_queue_filler_mem_seq_compl_flag,
sram_queue_area_init_en_from_the_vosq0_fifo_logic => sram_queue_filler_mem_cnt_en,
sram_queue_area_init_rstn_from_the_vosq0_fifo_logic => sram_queue_filler_rst_n,
q_lgth_ctl_from_the_vosq0_fifo_logic => sopc_vosq0_q_lgth_ctl,
sram_rd_rqt_delay_flag_to_the_vosq0_fifo_logic => sopc_vosq0_sram_rd_rqt_delay_flag,
sram_rd_rqt_timer_en_from_the_vosq0_fifo_logic => sopc_vosq0_sram_rd_rqt_delay_timer_en,
sram_rd_wr_delay_flag_to_the_vosq0_fifo_logic => sopc_vosq0_sram_rd_wr_delay_flag,
sram_rd_wr_timer_en_from_the_vosq0_fifo_logic => sopc_vosq0_sram_rd_wr_delay_timer_en,
```

```
        sram_wr_rqt_delay_flag_to_the_vosq0_fifo_logic => sopc_vosq0_sram_wr_rqt_delay_flag,
        sram_wr_rqt_timer_en_from_the_vosq0_fifo_logic => sopc_vosq0_sram_wr_rqt_delay_timer_en
      );

  -- outputs to sopc block
  sdram_addr_gen_loaded_addr <= sopc_vosq0_addr_load_data;
  vosq0_pkt_cnt_en <= sopc_vosq0_pkt_cnt_en;
  sdram_addr_gen_en <= sopc_vosq0_pkt_cnt_en;
  vosq0_pre_fifo_rdreq <= sopc_vosq0_prefifo_rdreq;
  sdram_addr_gen_preload <= sopc_vosq0_addr_load;
  write_delay_timer_en_n <= sopc_vosq0_delay_timer_en;

  sram_rd_rqt_delay_timer_en <= sopc_vosq0_sram_rd_rqt_delay_timer_en;
  sram_wr_rqt_delay_timer_en <= sopc_vosq0_sram_wr_rqt_delay_timer_en;
  sram_rd_wr_delay_timer_en <= sopc_vosq0_sram_rd_wr_delay_timer_en;

  vosq0_pkt_cnt_rstn <= sopc_vosq0_pkt_cnt_rst;
  vosq0_queue_lgth_cnt_en <= sopc_vosq0_q_lgth_cnt_en;
  vosq0_queue_lgth_ctl <= sopc_vosq0_q_lgth_ctl;
  vosq0_post_fifo_data <= sopc_vosq0_out_data;
  vosq0_post_fifo_wrreq <= sopc_vosq0_postfifo_wrreq;


  -- inputs to sopc block
  sopc_vosq0_pkt_blk_end <= vosq0_pkt_cnt_end_flag;
  sopc_vosq0_in_data <= vosq0_pre_fifo_q;
  sopc_vosq0_next_addr <= sdram_addr_gen_out_addr;
  sopc_vosq0_prefifo_aempty <= '1';
  sopc_vosq0_master_mem_init_done <= '1';
  sopc_vosq0_prefifo_rdempty <= vosq0_pre_fifo_rdempty;
  sopc_vosq0_delay_flag <= write_delay_flag;
  sopc_vosq0_postfifo_wrfull <= vosq0_post_fifo_wrfull;

  sopc_vosq0_sram_rd_rqt_delay_flag <= sram_rd_rqt_delay_flag;
  sopc_vosq0_sram_wr_rqt_delay_flag <= sram_wr_rqt_delay_flag;
  sopc_vosq0_sram_rd_wr_delay_flag <= sram_rd_wr_delay_flag;

END ARCHITECTURE main_behvrl;
```

END OF DOCUMENT