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

Raymond Peterkin

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

A Reconfigurable Hardware Architecture for VPN MPLS Based Services

TITRE DE LA THÈSE / TITLE OF THESIS

D. Ionescu

DIRECTEUR (DIRECTRICE) DE LA THÈSE / THESIS SUPERVISOR

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

EXAMINATEURS (EXAMINATRICES) DE LA THÈSE / THESIS EXAMINERS

T. Kwasniewski

V. Groza

Gary W. Slater

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

A Reconfigurable Hardware Architecture for VPN MPLS based Services

by

Raymond Peterkin

Thesis submitted to the
Faculty of Graduate and Postdoctoral Studies
In partial fulfillment of the requirements
For the MASc degree in Electrical Engineering

Ottawa-Carleton Institute for Electrical and Computer Engineering
School of Information Technology and Engineering
University of Ottawa
April 2006

Your file Votre référence
ISBN: 978-0-494-18456-1
Our file Notre référence
ISBN: 978-0-494-18456-1

I hereby declare that I am the sole author of this thesis.

I authorize the University of Ottawa to lend this thesis to other institutions or individuals for the purpose of scholarly research.

Raymond Peterkin

I further authorize the University of Ottawa to reproduce this thesis by photocopying or other means, in total or in part, at the request of other institutions or individuals for the purpose of scholarly research.

Raymond Peterkin

Abstract

Internet applications are becoming increasingly resource intensive and perform poorly in the presence of significant congestion. Increased bandwidth cannot provide long-term congestion relief so Internet traffic must be prioritized and efficiently routed. Multiprotocol Label Switching (MPLS) [12] provides the means to process traffic quickly and reserve resources for applications with specific requirements. However, MPLS must provide the same resilience mechanisms as ATM [18] over SONET [46] to become an acceptable alternative for assigning and switching label switched paths (LSPs). This thesis proposes a reconfigurable architecture and a prototype of a hardware processor for MPLS to improve its overall performance. Establishing LSPs and label management are the central tasks of the processor. It is used to describe LSPs and perform packet switching. A significant subset of RSVP-TE is implemented in the processor to provide the necessary mechanisms of a signaling protocol. Functionality is also available for Traffic Engineering (TE) allowing a user to configure the allocation of resources available for MPLS. The processor is designed to interact with software so it can become part of an embedded system. Results and analysis for the processor are provided describing its resource usage and performance. Resource intensive tasks are identified through analysis and determinations are made about the worst case performance improvement compared to software implementations which depend on the number of LSPs considered and network size.

Acknowledgements

I would like to thank my supervisor, Dr. Dan Ionescu, for his constant guidance, encouragement and support throughout my research. It would have not been possible to finish this work without his support.

I wish to thank my colleague and friend Mohamed Abou-Gabal for his assistance, encouragement and support through my research.

Finally I would like to thank my parents (Raymond and Joan) and sister (Leanne) for their support, patience, kindness and love.

Table of Contents

List of Figures

List of Tables

Glossary

Term	Definition
API	Application Program Interface.
ARP	Address Resolution Protocol.
AS	Autonomous System.
ASIC	Application Specific Integrated Circuit.
ATM	Asynchronous Transfer Mode.
BGP	Border Gateway Protocol.
CAC	Connection Admission Control.
CoS	Class of Service.
CR-LDP	Constrained-Routing Label Distribution Protocol.
DRAM	Dynamic Random Access Memory.
FEC	Forward Equivalence Class.
FIFO	First In First Out.
FPGA	Field Programmable Gate Array.
FSM	Finite State Machine.
GMPLS	Generalized Multiprotocol Label Switching.
IIH	IS-IS Hello.
IPv6	Internet Protocol version 6.
IS-IS	Intermediate System to Intermediate System Protocol.
KISS	Keep-It-Simple Signaling.
LC	Logic Cell.
LER	Label Edge Router.
LIFO	Last-In-First-Out.
LSA	Link State Advertisement.
LSA-DB	Link State Advertisement Database.
LSB	Least Significant Bit.
LSDB	Link State Database.
LSP	Label Switched Path.
LSR	Label Switching Router.
LUT	Lookup Table.
MAC	Medium Access Control.
MPLS	Multiprotocol Label Switching.
MSB	Most Significant Bit.
OSPF	Open Shortest Path First.
PLD	Programmable Logic Device.
PSN	Pseudo-Node.
RAM	Random Access Memory.
RSVP-TE	Resource Reservation Protocol-Traffic Engineering.
SDRAM	Synchronized Dynamic Random Access Memory.
SONET	Synchronous Optical Network
SPP	Shortest Path Processor.
TE	Traffic Engineering.
TE DB	Traffic Engineering Database.

TLV	Time Length Value.
TTL	Time to Live.
QoS	Quality of Service.
VHDL	Very High Speed Integrated Circuit (VHSIC) Hardware Description Language.
VoIP	Voice over IP.
VPN	Virtual Private Network.

Chapter 1

1 Introduction

1.1 Motivation

Resource intensive Internet applications like voice over Internet Protocol (VoIP) and real-time streaming video perform poorly when the core network of the Internet is congested. Increasing bandwidth provides temporary relief. However, bandwidth alone is not sufficient to provide an environment where internet applications can be executed efficiently due to delays. Long term relief can only be achieved through efficient prioritization of network resources and traffic. Traffic engineering (TE) and Quality of Service (QoS) can be used to address the viability of Internet applications [3]. MPLS can be used for TE and QoS and is gaining wide spread acceptance as a mechanism for those applications.

Software based implementations are common and used in all vendor's equipment however there is a significant cost with respect to performance. For example processing packets can take in the order of milliseconds and that kind of delay can lead to substantial performance degradation and lost data in high speed networks. To minimize those phenomena, hardware implementations have been researched in this thesis for MPLS virtual private network (VPN) [24] protocols with the goal of yielding significant performance improvements. A list of objectives has been set to achieve that goal and they are listed below:

1. Examine existing MPLS hardware research and determine unexplored or deemphasized areas for improvement.
2. Address areas of improvement through the design and implementation of a processor using FPGAs for the family of MPLS VPN protocols to explore specific concepts.
3. Determine extent of total improvement by analyzing the overall performance of the processor.

4. Isolate further areas of improvement by analyzing the processor's resource distribution among components performing specific tasks.

1.2 A Brief Review

VPNs are private communications networks in public networks. Using a public infrastructure for private communications has many potential benefits to an organization including global coverage, low operational costs, high security and simplified network topologies [38]. Internet-based VPNs have evolved as the most secure and cost effective means of achieving these goals and consequently their popularity has grown significantly among academic, commercial and government organizations.

Asynchronous Transfer Mode (ATM) is a dedicated-connection switching technology that organizes digital data into fixed cell lengths and transmits them over a physical medium using digital signal technology. ATM is designed to be easily implemented by hardware and therefore faster processing and switching speeds are possible. Synchronous Optical Network (SONET) is a fiber optic transmission system for data with guaranteed physical layer restoration time of 50 ms when a failure occurs. The use of ATM over SONET would satisfy VPN performance requirements. However, the Internet is predominantly composed of IP networks that do not provide the dedicated-connection technology available in ATM.

MPLS is a data transport mechanism in IP networks that brings the TE capabilities of ATM to packet-based networks. IP packets are tagged with labels that specify an LSP and priority. TE is facilitated through explicit routing, the ability to compute an LSP at the source, the ability to distribute a network topology and its attribute throughout a network and the ability to reserve resources and modify link attributes. MPLS supports QoS capabilities to guarantee a minimum set of performance characteristics as data is transported from a source to its destination. MPLS provides the necessary functionality in ATM to IP networks and makes the implementation of VPNs possible through the

Internet. However, the proliferation of software implementations for MPLS leaves a significant gap between the performance of ATM/SONET VPNs and MPLS VPNs. Hardware solutions with reconfigurable technology are desired to close that gap and achieve ATM/SONET like performance with MPLS and IP networks.

Hardware research taken place with the goal of improving MPLS performance. Thus far there has been hardware development on individual MPLS mechanisms but no solutions that combine multiple components or include TE or QoS. Therefore this thesis presents a hardware solution to address those concerns.

1.3 Contributions

The main contribution of this thesis is the design and implementation of a reconfigurable hardware/software architecture along with corresponding algorithms and procedures for the control plane of MPLS based networks. As a result a prototype for an MPLS processor was implemented accounting for QoS and TE as much as possible. Such a processor would be implemented with reconfigurable hardware as part of an embedded system. This work has been undertaken to different extents in open literature. To the best of the author's knowledge certain aspects of MPLS have not been directly considered with respect to hardware design or implementation. The following contributions have been specifically made by this thesis:

1. Hardware implementation of a signaling protocol to establish label switched paths while accounting for user specified traffic engineering parameters, including any algorithms or procedures for maintaining and updating traffic engineering information.
2. The population and management of a label database so MPLS packets can be forwarded according to a set of acceptable values, including any algorithms or procedures specifically developed to perform these tasks.
3. An interface designed to interact with a bus such that interaction with software is immediately possible.

1.4 Organization of the Thesis

This thesis is organized into several chapters. Chapter 2 is an extensive literature review providing overviews of MPLS, RSVP-TE and hardware implementations of MPLS. MPLS is defined with explanations of relevant concepts, mechanisms and communications protocols. RSVP-TE concepts relevant to the thesis are then summarized and hardware implementations of MPLS are discussed in the significant detail. After discussing key hardware concepts commercial and academic implementations of MPLS are described to demonstrate potential areas of research in this domain. Chapter 3 provides a high level description of the MPLS processor with descriptions of system features, limitations and functions. Chapter 4 illustrates the processor's implementation in significant detail with detailed descriptions of every component. The processor's performance is analyzed and discussed in Chapter 5 and Chapter 6 summarizes the key points of this thesis.

Chapter 2

2 Literature Review

This chapter serves as a literature review for MPLS, RSVP-TE and relevant hardware implementations. Pertinent background information is provided and relevant work is cited when appropriate. A conclusion is provided at the end of the chapter discussing areas of improvement addressed in subsequent chapters of this thesis.

2.1 MPLS Framework

2.1.1 Definition

Multiprotocol Label Switching (MPLS) is a protocol framework used to prioritize Internet traffic and improve bandwidth utilization [47]. Those functions are accomplished by inserting a label between layer 2 data and layer 3 data in a packet and forwarding the packet based on the label contents alone as in ATM networks. Performance and efficiency are increased by this approach because less time is required to process a label than to process routing information like source and destination IP addresses. Errors are also detected more quickly they can only occur with a single label than with potentially several data types in different protocols. Figure 2-1 describes a packet that has been modified for MPLS. Protocol data above layer 3 is unused in MPLS while layer 2 data appears transparent. Since the Internet is a collection of numerous networks using various communications technologies MPLS can be used without any changes to those existing mechanisms.

2.1.2 Router Types

MPLS operations are performed on two types of routers. Label Edge Routers (LERs) operate at the "edge" of an MPLS network and typically contain interfaces to dissimilar

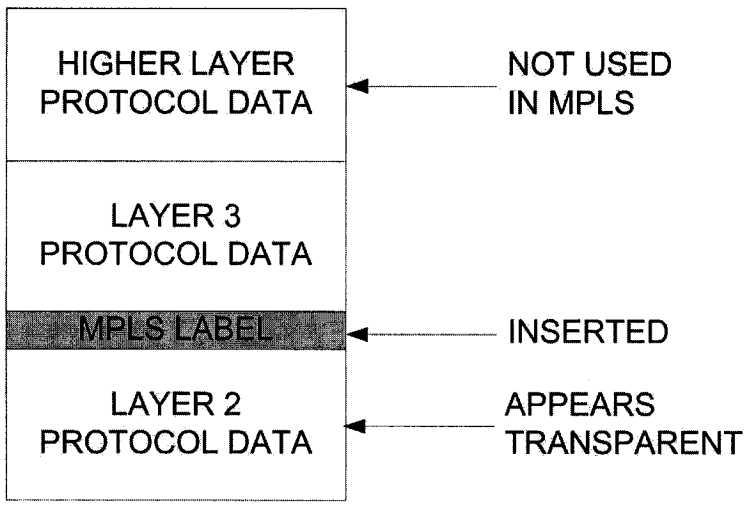


Figure 2-1. Network packet modified for MPLS

networks. LERs route traffic and are used as an interface between layer 2 networks like ATM [18], Frame Relay [23] or Ethernet and an MPLS core network.

When LERs receive a packet from a layer 2 network, a label is then attached to that packet and sent into the MPLS core network. Subsequent interpretation of that packet is done on the basis of that label. Depending on how other routers are configured, a packet will follow a specific path known as a label switched path (LSP) going from one LER to another. When an LER receives a packet from the MPLS network, the label is removed and the packet is sent to the appropriate layer 2 network. LERs sending packets into the MPLS network are described as ingress while LERs sending packets to layer 2 networks are called egress. Both ingress and egress LERs participate in the establishment of LSPs prior to exchanging packets.

Label Switch Routers (LSRs) form the core of the MPLS network. They participate in forwarding packets to other MPLS routers and establishing LSPs. They receive packets from an LER or an LSR, analyze the label and forward the packet to an LSR or LER depending on the label contents.

Figure 2-2 illustrates a typical MPLS network. Each LER is connected to one or more layer 2 networks and the MPLS network. LSRs are either connected to other LSRs or

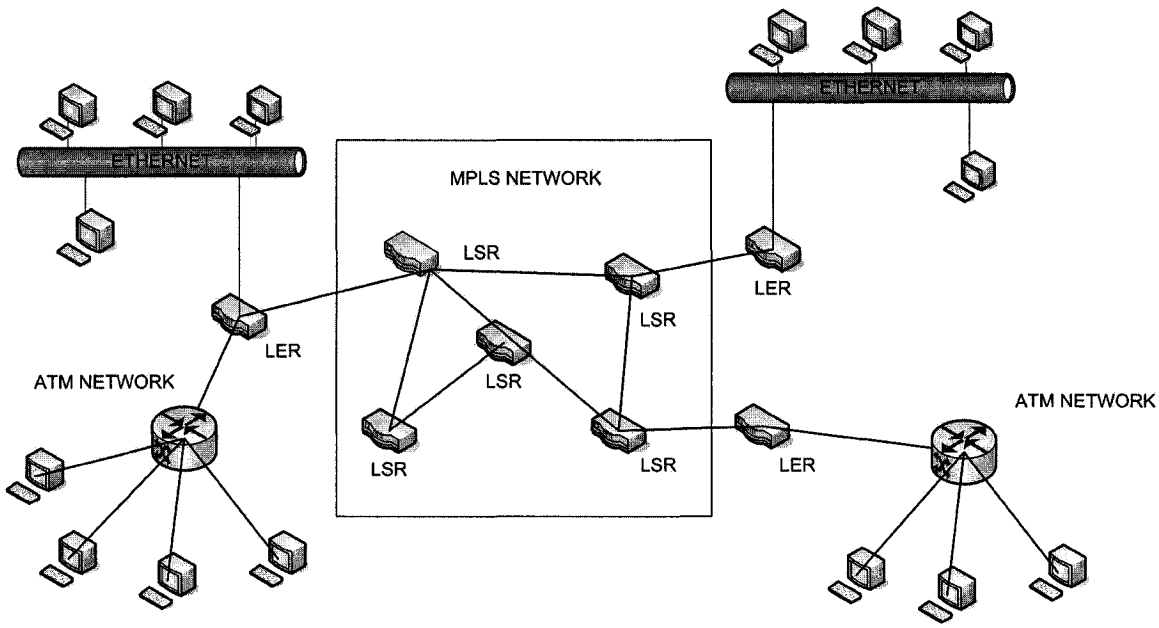


Figure 2-2. MPLS Network

LERs. Packets originate from a layer 2 network, are passed to an LER, then to one or many LSRs before reaching an egress LER that forwards the data to a final layer 2 network.

Figure 2-3 shows a set of packet exchanges from an ingress LER to an egress LER with intermediate LSR communication. When the ingress LER receives layer 2 data, it is analyzed and a label is added to the packet. The new packet is then forwarded to the appropriate LSR. Subsequent LSRs analyze the label, remove it and attach a new label so the next MPLS router can correctly interpret the label information. When the packet reaches the egress LER, the label is removed and the packet is forwarded to the appropriate layer 2 network.

2.1.3 Label Description

A label is a short, fixed length packet identifier. The label is considered unstructured, its significance is limited to a particular link and it decouples packet forwarding from IP headers. Figure 2-4 describes the format of a generic label. Each label is thirty-two bits

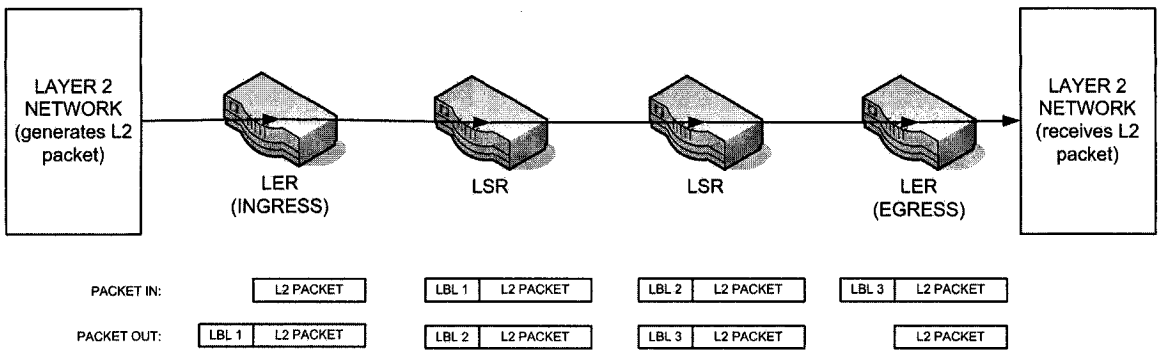


Figure 2-3. MPLS Packet Exchange

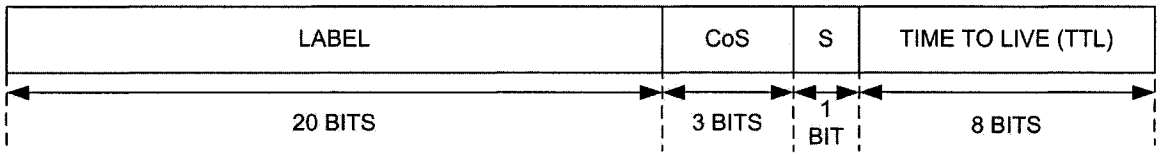


Figure 2-4. Generic MPLS Label

long with twenty bits dedicated to the actual label value. Three bits are used to describe the Class of Service (CoS) for creating different groups among packets using the same label. The S bit specifies which label is at the top of the label stack. The Time to Live (TTL) [35] field indicates how much longer the packet can remain in the network. The TTL value is decremented for every router that a packet passes through a network. Once the value reaches zero the packet is discarded.

2.1.4 Forwarding Concepts

Forwarding refers to the actual movements of packets from a router's input to a router's output. Typically, a forwarding table and packet information are needed to properly forward packets in a router. Procedures to make a forwarding decision under these circumstances are defining the information in a packet to find an entry in the table and specifying how an entry is found. Packet information used to make a forwarding decision is normally the destination address and table information usually contains a subnet number, subnet mask, interface number and next hop address.

In MPLS networks, all possible packets are separated into a set of disjoint groups. Each group is treated the same with respect to information normally found in a routing table (incoming interface, next hop, etc.) even if they differ in terms of network layer information. These groups are called Forward Equivalence Classes (FECs). All packets in an FEC are treated the same. They are forwarded along the same LSP and mapped to the same label. Multiple FECs may be mapped to the same FEC and QoS can be used with the CoS bits of any label. With FECs being mapped to LSPs mechanisms like routing tables are not required in MPLS.

FEC label binding is done once at the ingress router. It is based on the destination address (the information previously read from the packet). Further mappings are done based LSPs or policies set by the administrator.

2.1.5 Protocol Architecture

Each router must run a set of protocols in order for the label switching described above to be possible. These protocols are used to gain knowledge of the network topology, establish LSPs and perform other functions necessary for label switching to be possible in an MPLS network. Figure 2-5 illustrates the protocols that comprise MPLS.

Routing protocols (like OSPF, IS-IS and BGP described in [25], [11] and [49] respectively) are required to obtain a topology of the entire network. That network topology is stored in the routing table and used by the signaling protocols like Label Distribution Protocol (LDP), Constrained Routing LDP (CR-LDP), Resource Reservation Protocol (RSVP) and Resource Reservation Protocol Traffic Engineering (RSVP-TE) described in [27], [22], [37] and [9] respectively. Signaling protocols are used to create LSPs and labels based on the network topology and forward that information throughout the network. With the LSPs established, the data plane can successfully process incoming labeled packets and forward them to the correct destinations.

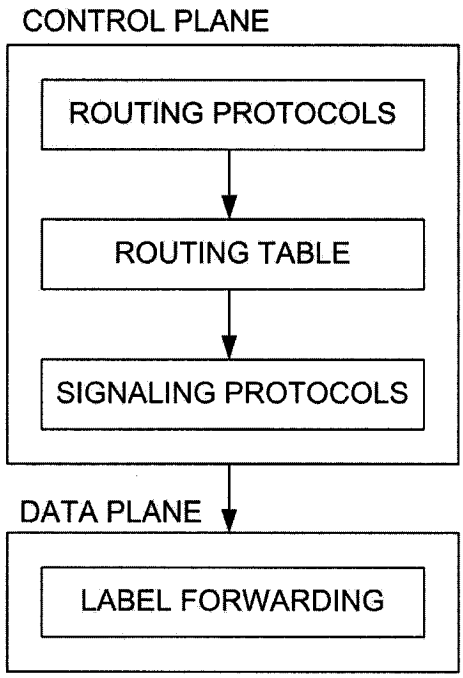


Figure 2-5. MPLS Protocol Stack

2.1.6 LSP Concepts

Data that flows along an LSP is said to flow either upstream or downstream. Downstream is the general direction from the ingress of the LSP its egress. So from any LSR, routers closer to the egress LER are considered downstream while any router closer to the ingress LER is upstream. LSPs are unidirectional and hence data can only flow downstream. Data flows upstream and downstream to perform various tasks including the reservation of resources and the verification of network data.

2.1.7 Label Concepts

There are three central issues related to the distribution of labels in MPLS. Those issues are how to control the distribution of labels, when to distribute labels and how long to keep labels. The protocol chosen to establish LSPs will dictate how each of these issues is addressed and the overall performance of that protocol.

Controlling the distribution of labels in an MPLS network can be done in an ordered or independent manner. Ordered label distribution means that routers wait to receive label bindings from upstream or downstream routers before sending its own labels. Independent label control means that routers distribute labels to their neighbors at any time. Ordered label control allows for simpler implementations but require more time because there may be a lengthy wait to receive labels necessary to perform tasks. Conversely, independent label control operates more quickly but functionality must be available to process labels that may be received at any time. The RSVP-TE mechanisms use ordered label control while LDP uses independent label control. If a router distributes their labels in an unsolicited manner, labels are advertised to all neighbors. If labels are distributed on-demand then they will not be sent until a request is made.

Label retention is done in a liberal or conservative manner. A liberal label retention policy means that once a binding is made it is always kept. Therefore, routers can maintain bindings received from LSRs other than the valid next hop. If the next hop changes those bindings can be used immediately and rapid adaptation to routing changes are possible. The main disadvantage to this approach is that more label mappings must be kept. So if a label binding becomes invalid for any reason it is not discarded. Conservative label retention discards bindings as soon as they are not considered useful. So fewer labels are kept but less adaptation to changing network conditions is possible.

Two types of label bindings are possible for incoming and outgoing labels, local binding and remote binding. With local binding an individual router selects and assigns labels. Routers using remote binding routers receive externally assigned labels. Remote binding reduces the work performed on each router but limits the number of labels available throughout the network. Local binding is more common in MPLS networks.

Routers can create or destroy a binding between a label and an FEC by data packets or by control information processed by a router. The former is considered data driven while the latter is control driven and both have an impact on performance, scalability and

robustness depending on the traffic in the network. Control driven binding is more commonly utilized given its simplicity and robustness.

2.1.8 Advanced MPLS Concepts

VPNs are a method of using public network for private communications among a set of remote sites. VPNs based on a layer 2 technology and managed at that layer are defined as layer 2 VPNs. VPNs based on tunneling above layer 3 are Layer 3 VPNs, (L2TP [48], IPSec [42], BGP/MPLS [14]). LSPs provide the mechanisms through which private communications are possible making MPLS a means through which to establish and maintain VPNs.

Quality of Service (QoS) [43] is the ability to guarantee end-to-end transmission characteristics for a flow of information. Those characteristics include bandwidth, delay, jitter and packet loss. Two models for IP QoS are Integrated Services (IntServ) [44] and Differentiated Services (DiffServ) [39]. IntServ is an RSVP-based architecture for QoS and DiffServ does not require signaling and deals with aggregates of flows. MPLS can be used to implement both IntServ and DiffServ through its signaling protocol or CoS label bits.

When network or link failures occur LSPs must be torn down and the process of reestablishing a new LSP must take place. That process may take a few seconds, an unacceptably long period of time for many Internet applications. MPLS can be used to perform fast reroute (FRR) [36] or protection to guard against potential failures. Protection can be for an entire path, link or node. In all cases backup LSPs are established in case of a failure. That way data can be rerouted to a backup LSP in less time than it would take to establish a new one.

2.2 RSVP-TE Protocol

This section provides an overview of RSVP-TE as a signaling protocol for MPLS. RSVP-TE is defined along with the common operations, reservation styles and packet formats with great emphasis on details for LSPs, errors and TE.

2.2.1 Definition

Resource reservation protocol, traffic engineering (RSVP-TE) is a protocol for reserving resources in the Internet using label distribution. It defines how applications place reservations and how they can relinquish those resources once their need concludes. Figure 2-6 illustrates how RSVP-TE fits with other protocols in IP networks. Data link protocols are transparent to RSVP-TE and other MPLS protocols. RSVP-TE data is encapsulated in an IP packet as part of its payload.

Figure 2-7 describes how LSPs are established with RSVP-TE. The desired LSP has already been determined by a routing protocol (e.g. OSPF) or through some other means. A PATH message is sent from the desired ingress LER to every LSR until it reaches the desired egress LER. Once a PATH message reaches the end of the desired LSP, RESV messages are sent from the egress LER along the same path back to the ingress LER. When an RESV message is received at the ingress LER, and no errors have occurred while pass PATH or RESV messages, the LSP is considered established and can be used to send application data from the LSP source to the destination.

Once an LSP has been established, PATH and RESV messages must be continuously sent to maintain reservations of the desired resources. So a source or destination may choose to terminate an LSP by simply not sending PATH and RESV messages. The characteristic of LSPs allowing them to exist by continuous message transmission is called 'soft state'. By being a soft state protocol, RSVP-TE lets routers deal gracefully with changes in routes and automatically reestablishing resource reservation if necessary. There is also independence of any routing protocol through soft state functionality.

RSVP-TE LSPs are simplex (unidirectional) and can be unicast (from one sender to one receiver) or multicast (from one sender to multiple receivers). Changes can be made in multicast group membership and receivers may select once source from among multiple sources transmitting in multicast group.

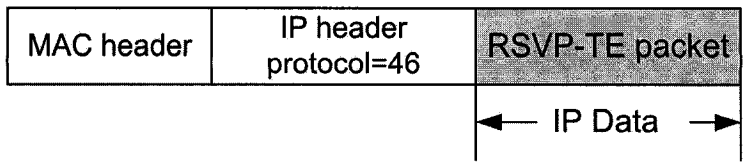


Figure 2-6. RSVP-TE Protocol Stack

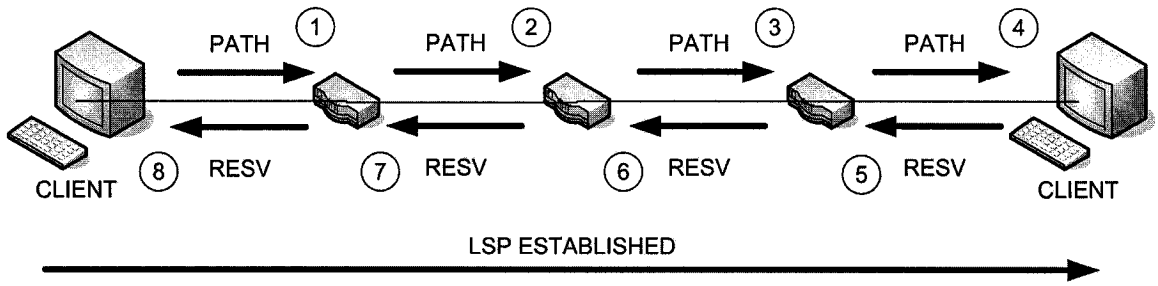


Figure 2-7. RSVP-TE Operation

2.2.2 Reservation Styles

With RSVP-TE it is possible to reserve resources in three different styles. Each style is more appropriate to a different set of circumstances depending on the applications reserving resources. The first, and simplest, form of reservation is Fixed Filter style. With this style resources are reserved for only one flow so there is one sender and one receiver. That option can be applied to all scenarios involving resource reservation but may not be the most appropriate.

If resources are being reserved for an audio conference among several computers, separate paths to account for all computers are not necessary because all people cannot speak at one time. So resources can be shared among numerous flows. The second and third reservation styles are 'Shared Explicit' and 'Wildcard Filter' respectively. The difference between the two styles is in how tightly the application must specify the flows that share the resource. With a shared explicit reservation, the application must explicitly identify every sender. With the wildcard filter style, every sender does not have to be identified. Any flow matching the reservation's specification may use the resource, regardless of the sender.

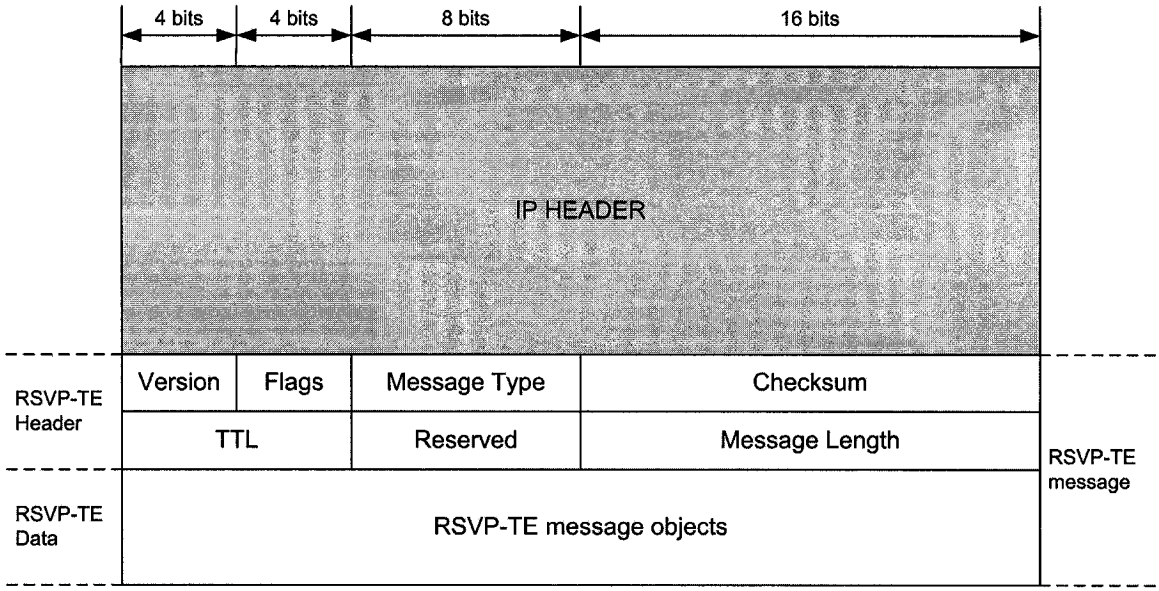


Figure 2-8. RSVP-TE Packet

2.2.3 Packet Formats

RSVP-TE packets have the format shown in Figure 2-8. RSVP-TE messages are encapsulated in IP packets and have a header and data components. The header has a four bit version number and four bit flags field. The current version number used for RSVP-TE is 1. The eight bit message type can describe several types possible in RSVP-TE. Only six message types are emphasized here and in the implementation described in subsequent chapters.

Path and Reservation (RESV) messages are used to establish an LSP and reserve resources respectively. Their use is described in the previous section. Path error (PATHERR) and reservation (RESVERR) messages are used when an error occurs during the propagation of PATHERR and RESVERR messages respectively. Once an error has been detected PATHERR messages are sent upstream until they reach the ingress. RESVERR messages are sent downstream until they reach the egress. Paths can be torn down with path teardown (PATHTEAR) or reservation teardown (RESVTEAR) messages. These messages are sent upstream and downstream like their error message counterparts when it has been determined that path or reservations are to be terminated.

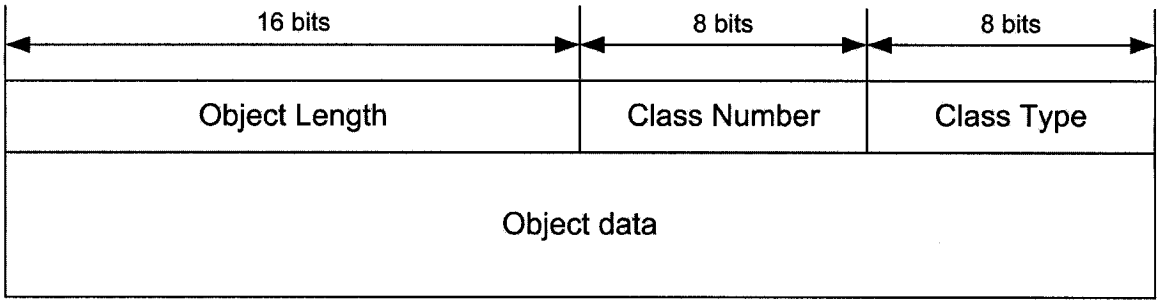


Figure 2-9. RSVP-TE Object Format

Table 2-1. RSVP-TE Object Information

Object Name	Class Number	Class Type	Applicable Messages	Description
ERROR_SPEC	6	1	PATHERR, RESVERR	Contains error information
FILTER_SPEC	10	7	RESV	Describes path ingress information
FLOWSPEC	9	2	RESV	Describes flow resources
LABEL	16	1	RESV	Lists label
LABEL_REQUEST	19	1	PATH	Defines request for label
RSVP HOP	3	1	All except PATHERR	Defines previous hop in path
SENDER_TEMPLATE	11	7	PATH	Describes path ingress information
SENDER_TSPEC	12	2	PATH	Describes flow resources
SESSION	1	7	ALL	Describes path egress information
STYLE	8	1	RESV, RESVERR	Lists reservation style
TIME VALUES	5	1	PATH, RESV	Lists time refresh interval

RSVP-TE data is composed of different objects. The general composition of an object is shown in Figure 2-9. The object length is a sixteen bit number with the number of total bytes for the object, including the header. The eight bit class number and class type identify the object. Thirty eight different object types are defined but only the mandatory objects of the packets defined above are of interest. Table 2-1 summarizes object information including the name, class type and number, description and associated messages.

2.2.4 Object Formats

This section describes the formats for all the objects summarized in Table 2-1. Details or references regarding the formats are provided as they are needed.

General Objects

This section describes objects that require no modification from the original RSVP protocol to accommodate TE. The RSVP HOP object in Figure 2-10 identifies the last system in the sequence of PATH or RESV messages. The logical interface handle is used to help identify the flow. Reservation requests from downstream include this value.

The STYLE object in Figure 2-11 describes the reservation style, indicated by the option vector. The first 19 bits of the option vector are unused. The next two bits identify the type of sharing control (01 for distinct reservations and 10 for shared reservations) and the last three bits indicate wildcard (001) or explicit (010) reservations.

The TIME VALUES object in Figure 2-12 tells how often the sender of the reservation request (LSP egress) will refresh its reservation. If upstream routers do not receive a refresh message in the period of time indicated below then the reservation is considered relinquished.

Flow Related Objects

FLOWSPEC and SENDER_TSPEC objects described in Figure 2-13 and Figure 2-14 carry the main part of the reservation request and are fully described in [26]. All resources required by the flow are identified in these objects. The token bucket rate, token bucket size, peak data rate, minimum policed unit and maximum packet size are meaningful to the part of the router responsible for QoS. All other values are reserved and subsequently are not used.

Object Length: 12	Class Number: 3	Class Type: 1
IPv4 previous hop address		
Logical handle interface		

Figure 2-10. RSVP HOP Object Format

Object Length: 8	Class Number: 8	Class Type: 1
Flags	Option vector	

Figure 2-11. STYLE Object Format

Object Length: 8	Class Number: 5	Class Type: 1
Refresh period (milliseconds)		

Figure 2-12. TIME VALUES Object Format

Object Length: 36		Class Number: 9	Class Type: 2
0	Reserved	7	
5	0 Reserved	6	
127	0	5	
Token Bucket Rate [r] (32-bit IEEE floating point number)			
Token Bucket Size [b] (32-bit IEEE floating point number)			
Peak Data Rate [p] (32-bit IEEE floating point number)			
Minimum Policed Unit [m]			
Maximum Packet Size [M]			

Figure 2-13. FLOWSPEC Object Format

The significance of having both objects, in spite of their similarity (they only differ with reserved values) is that the FLOWSPEC object is used in RESV packets and the SENDER_TSPEC object is used in PATH packets.

<table>
<tr><td colspan="3">Object Length: 36</td><td>Class Number: 12</td><td>Class Type: 2</td></tr>
<tr><td>4</td><td colspan="3">Reserved</td><td>7</td></tr>
<tr><td colspan="2">1</td><td>0</td><td>Reserved</td><td>6</td></tr>
<tr><td colspan="2">127</td><td colspan="2">0</td><td>5</td></tr>
<tr><td colspan="5">Token Bucket Rate [r] (32-bit IEEE floating point number)</td></tr>
<tr><td colspan="5">Token Bucket Size [b] (32-bit IEEE floating point number)</td></tr>
<tr><td colspan="5">Peak Data Rate [p] (32-bit IEEE floating point number)</td></tr>
<tr><td colspan="5">Minimum Policed Unit [m]</td></tr>
<tr><td colspan="5">Maximum Packet Size [M]</td></tr>
</table>

Figure 2-14. SENDER_TSPEC Object Format

<table>
<tr><td colspan="2">Object Length: 12</td><td>Class Number: 6</td><td>Class Type: 1</td></tr>
<tr><td colspan="4">IPv4 address of system that detected error</td></tr>
<tr><td>Flags</td><td>Error code</td><td colspan="2">Error value</td></tr>
</table>

Figure 2-15. ERROR_SPEC Object Format

Error Objects

Path messages and reservation requests may encounter a variety of errors. Separate error messages are prepared for each error and are transported in the ERROR_SPEC object, described in Figure 2-15. The error code and value provide details of the problem encountered. Specific errors are indicated through the error code while additional information is given through the error number.

Table 2-2 summarizes a common set of error codes and their corresponding error conditions. While the list is not comprehensive it does cover most of the common errors that can occur.

Table 2-2. RSVP-TE Error Codes and Values

Error Code	Description
1	Admission control failure.
2	Policy control failure.
3	No path information for this reservation.
4	No sender information for this reservation.
5	Conflicting reservation styles.
6	Unknown reservation style.
13	Unknown object class.
14	Unknown object type.
23	RSVP-TE system error.

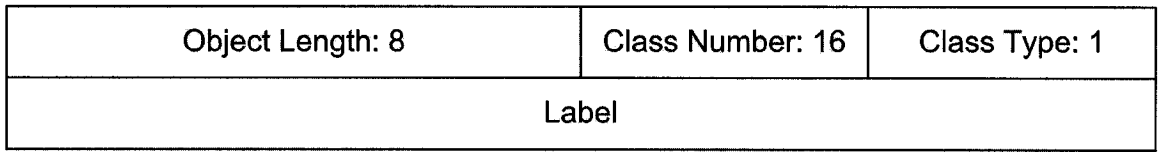


Figure 2-16. LABEL Object Format

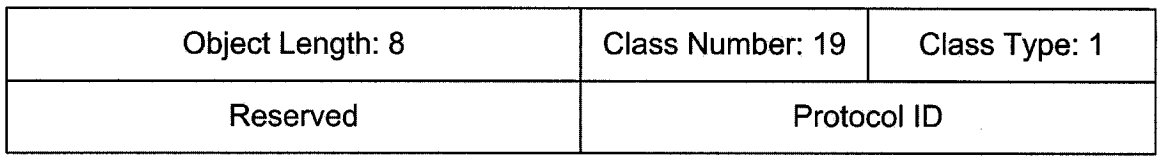


Figure 2-17. LABEL_REQUEST Object Format

TE Objects

MPLS facilitates traffic engineering through the use of explicit paths and label switching. The RSVP-TE objects to support traffic engineering are described in this section. Figure 2-16 and Figure 2-17 are the objects used for labels themselves. The former contains the actual label while the latter contains a label request.

The reserved value of the LABEL_REQUEST object must be set to zero on transmission and must be ignored when received. The protocol identifier of the LABEL_REQUEST identifies the layer 3 protocol using in the LSP.

The remaining figures describe the objects used to pass information identifying the LSP. The SENDER_TEMPLATE object described in Figure 2-18 is used in PATH packets to describe the LSPs ingress with the source IP address and the LSP ID. All data available for PATH packets in Figure 2-18 is used in the FILTER_SPEC object shown in Figure

Object Length: 12	Class Number: 11	Class Type: 7
IPv4 address of LSP ingress		
Reserved	LSP ID	

Figure 2-18. SENDER_TEMPLATE Object Format

Object Length: 12	Class Number: 10	Class Type: 7
IPv4 address of LSP ingress		
Reserved	LSP ID	

Figure 2-19. FILTER_SPEC Object Format

Object Length: 16	Class Number: 1	Class Type: 7
IPv4 address of LSP egress		
Reserved	Tunnel ID	
Extend tunnel ID (zero or sender IP address)		

Figure 2-20. SESSION Object Format

2-19 for RESV packets. Figure 2-20 describes the SESSION object and it is used in all packet types to describe the LSPs egress with the destination address, tunnel ID and extended tunnel ID.

2.3 Hardware Implementations of MPLS Protocols

This section summarizes hardware concepts and different hardware implementations for MPLS. A brief summary of known solutions are provided from commercial products and research. Further areas of research in this domain are presented wherever possible.

2.3.1 Hardware Concepts

Software implementations are relatively convenient from a cost/performance perspective but performance is not optimal because many instructions are required. Hardware

implementations are sought to achieve the best performance possible. Single-purpose processors are implemented when a single task is performed in hardware. To accommodate several algorithms an application-specific processor is implemented, where the desired instruction is read from memory and subsequently executed. Hardware is normally implemented in a control unit and a data path that interact with each other. The data path contains the means for manipulating and storing temporary data. The control unit retrieves instructions and manipulates the data path accordingly. Hardware description languages like Verilog or VHDL are used to implement hardware solutions. Processors are normally part of an embedded system as described in [15], where both software and hardware interact to perform several tasks. Common or time consuming tasks are typically implemented in hardware while everything else is done in software.

Hardware is normally implemented on application specific integrated circuits (ASICs) or programmable logic devices (PLDs). An ASIC is an integrated circuit customized for a particular use. They are relatively small and offer the best performance but are not suited to rapid prototyping. PLDs are larger than ASICs but can be used to implement a variety of hardware designs. Field programmable gate arrays (FPGAs) [40] are a complex type of PLD that has grown rapidly in popularity over the past decade. They represent a reconfigurable hardware platform that can be used to implement any solution at the hardware level provided there are sufficient resources. Several models of FPGAs are used in industry and research, like those from Altera Inc. and Xilinx Inc.

2.3.2 A Review of MPLS Based Networking Equipment

Numerous telecommunications equipment vendors offer MPLS emphasizing different protocols and features. Products on the market include the Alcatel 7300 ASAM [1] (configured through the Alcatel 5620 Service Access Manager (SAM) described in [2]), Avici QSR and SSR routers [7], Cisco 7600 [8], Ericsson AXD 301 [17], Foundry NetIron [16], Hammerhead HSX 6000 [19], Juniper M40 [32], Mangrove Piranha [33], and Nortel Multiservice Provider Edge (MPE) Portfolio [34] among several others. Differences are significant enough for comparisons among vendors to be difficult.

Greater emphasis is typically place on layer 2 or layer 3, different signaling or routing protocols are used and implemented to different extents and different amounts of configurability are possible in each case. Among the commercial products the following trends are true in most cases:

- Traffic engineering and Quality of Service (QoS) are available in most cases.
- RSVP is the most commonly used signaling protocol. Where traffic engineering has been made available RSVP-TE is normally used.
- Where layer 2 MPLS is used, Ethernet is most common over IPv4 although IPv6 is widely available as well.

2.3.3 MPLS Hardware Research

This section summarizes research that has taken place with respect to hardware development and MPLS protocols. The first section describes hardware research for routing protocols, the second section describes hardware development for signaling protocols and finally the last section describes hardware methodologies for the MPLS data plane. The images shown in the following subsections are taken from the work discussed at the time unless stated otherwise.

Routing Protocol Hardware Research

The sections below describe hardware research for OSPF and IS-IS as they are the most commonly researched intra domain routing protocols.

IS-IS

The work in [31] describes a hardware implementation of IS-IS. IS-IS is an interior gateway protocol and link state protocol that uses Dijkstra's algorithm to find the shortest path between two routers. Figure 2-21 shows the network elements involved in an IS-IS domain. A host is referred to as an End System (ES), a router is called an Intermediate System (IS), and a designated router is referred to as a Designated Intermediate System

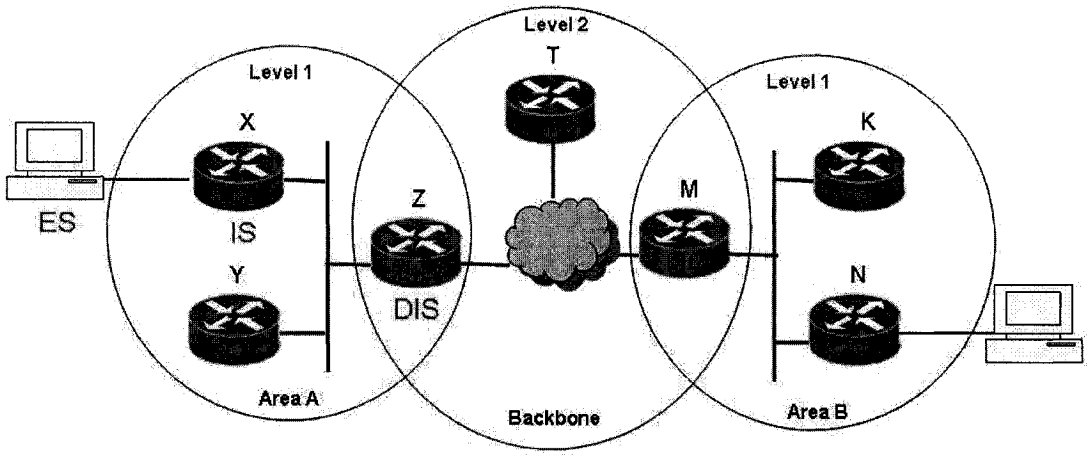


Figure 2-21. IS-IS Domain

(DIS). The data link interface address (MAC address) is known as a Sub-network Point of Attachment (SNPA). The network layer address (IP Address) is known as a Network Service Access Point (NSAP). IS-IS has two levels of area routing; level 1 (L1) and level 2 (L2). In L1, ISs route within their own areas. If the destination is not within the area then data will be routed to the closest IS. L2 acts as a backbone where ISs route towards other areas. Certain ISs can be configured for both L1 and L2. Routing information packets used among ISs are known as Link State PDUs (LSPs). Information collected by LSPs is stored in a Link State Database (LSDB). If an IS belongs two levels (L1L2) then that IS will have two LSDBs; one for each level.

Figure 2-22 illustrates a high level operation of IS-IS. The hello state illustrates the activities performed in establishing a connection between two ISs. From Figure 2-21, it is assumed that Y is an IS coming up. Z acts as a DIS (i.e. performs the flooding of LSPs) and as a Pseudo-node (PSN) (i.e. the virtual node that emulates a broadcast link). When Y comes up it will be idle and it must establish its adjacencies. Y will send out IS-IS Hello (IIH) packets with a null SNPA. Z receives IIH packets and checks its LSDB to determine if an adjacency already exists. The IIH packet is ignored if the adjacency exists; otherwise Z creates a new adjacency and sends an IIH packet to Y. Y receives the IIH packet from Z and returns an acknowledgement. Z returns an additional IIH packet to completely establish the connection.

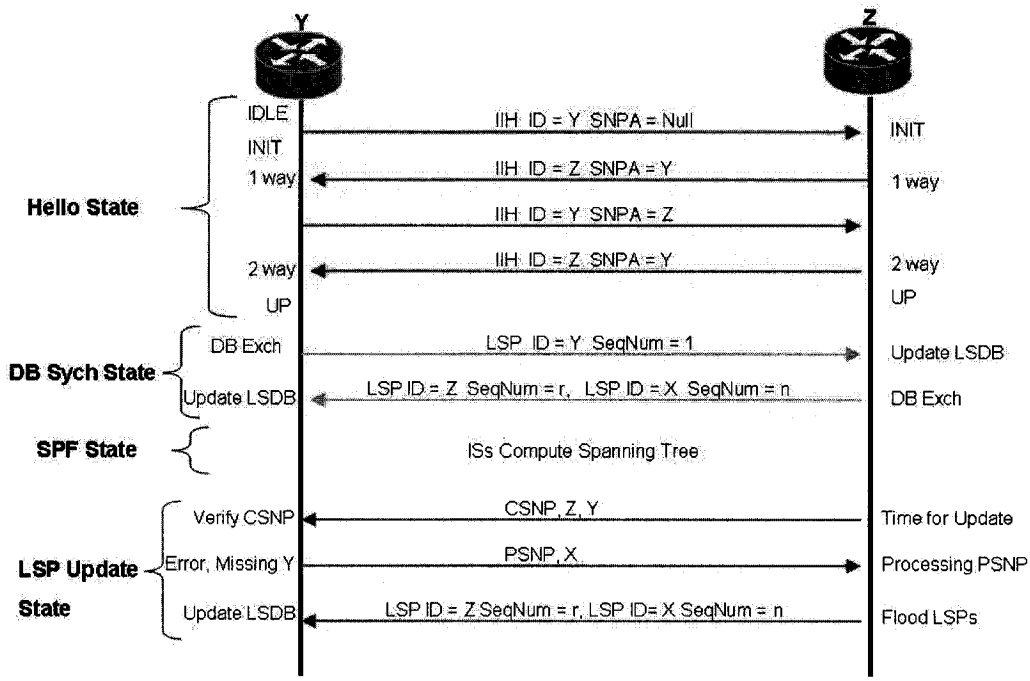


Figure 2-22. IS-IS Operations

The next state is the Database (DB) Synchronization state where exchanges of LSPs occur. The sequence number integrated with LSPs is used to differentiate older LSPs from newer LSPs. From Figure 2-21, after Y has established its adjacency, it will send out its LSP with sequence number 1. Upon receiving the LSP Z sends other LSPs to Y. Y uses these LSPs to create its L1 LSDB. Once databases are synchronized, all ISs compute the shortest path within their area in a state known as Shortest Path First (SPF).

The architecture is divided into a control unit and data path as illustrated in Figure 2-23. The control unit is composed of 4 processors, the Main Processor (MP) is responsible for invoking and updating the other 3 processors based on external commands and feedback from all other processors. The ingress packet processor (IPP) is responsible for processing and buffering packets received on the ingress path. It also updates the MP with status signals on the type of packets received. The Shortest Path Processor (SPP) is responsible for computing the SPF. The Egress Packet Processor (EPP) handles all tasks relating to sending packets as the result of LSP advertisements, responding or acknowledging packets.

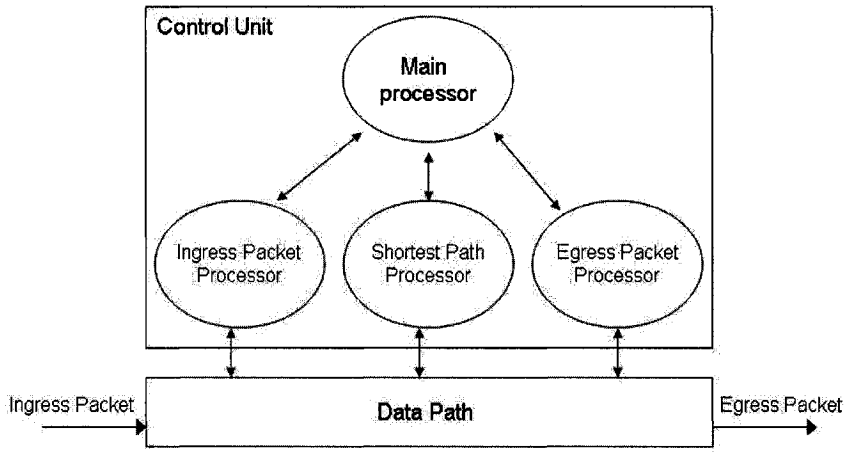


Figure 2-23. IS-IS Hardware System View

The control unit is presented in the state machine shown in Figure 2-24. The Hello state is where the adjacency establishment is achieved. If an adjacency was already established then the control unit ignores the hello packet and goes back to the IDLE state. If an adjacency was not established then the control unit remains in the hello state and performs all necessary computations. In the DB Exchange state the control unit receives and advertises LSPs to make sure that its LSDBs are synchronized with the latest network topology. Upon completing the DB Exchange state, the control unit processes buffered LSPs and stores them into the appropriate LSDB, in a state known as LSDB Update. Once the LSDB update state is complete, the SPF state is executed and the shortest path is computed. The control unit moves to an IDLE state where it waits for timers to be triggered. In the case of a DIS deployment, the triggered timers force the control unit to enter the C/P-SNP Processing state where CSNP packets are sent out to the network. The control unit stays in the C/P-SNP Processing state for a certain amount of time to ensure that no PSNPs are received. In a regular IS deployment, the control unit moves to the C/P-SNP Processing state if a CSNP is received on the ingress path. In some cases, if a CSNP was missing some LSP summaries, then LSP advertisements will be flooded on the network and as a result the control unit will go to the LSDB Update state and SPF states.

As shown in Figure 2-25, the data path is composed of ingress components, egress components, LSDBs, system inputs, the SPP data path, registers, multiplexers and

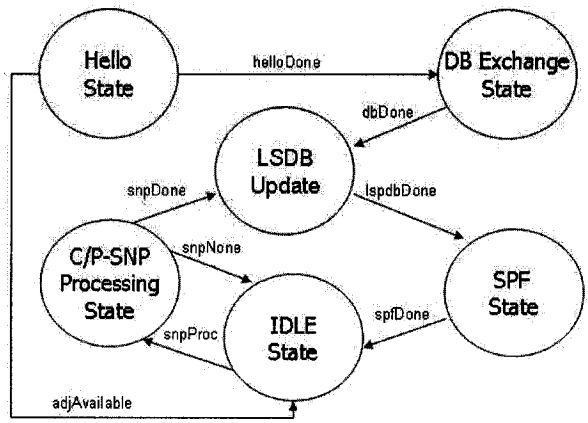


Figure 2-24. IS-IS Control Unit

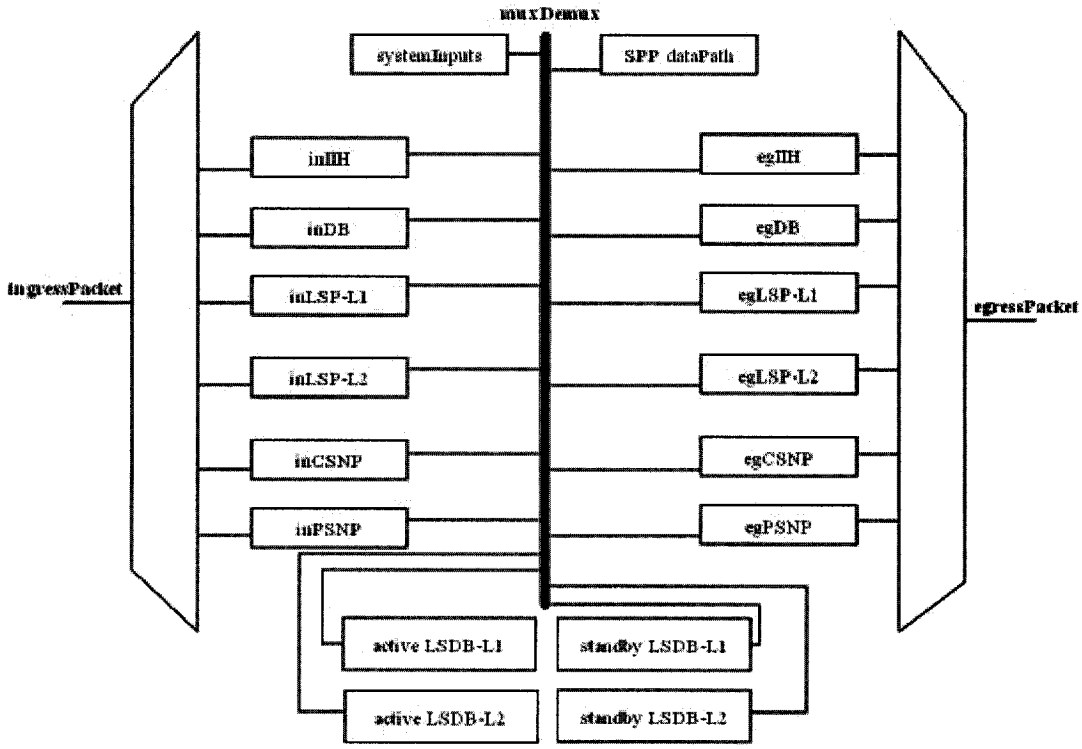


Figure 2-25. IS-IS Data Path

demultiplexers. The ingress and egress components are buffers that store packets. The ingress components are denoted by "in" in front of the component name and the egress components are denoted by "eg". Buffers have been implemented for IIH packets, database packets, level 1 and level 2 packets, CSNP packets and PSNP packets.

There are two major LSDBs; the active LSDB-L1 and the active LSDB-L2. The active LSDB-L1 stores the processed LSPs for L1 and the active LSDB-L2 stores the processed LSPs for L2. The standby LSDBs are initialized to zero. It is used during the deletion

(link failures or IS crashes) or the addition of LSPs. In the scenario of adding a new LSP, the IS-IS system adds the new LSP to the standby LSDB, copies the active LSDB to the standby LSDB, then it activates the standby LSDB and deactivates the active LSDB (the standby LSDB becomes active and the active LSDB becomes standby). In the scenario of deleting a LSP, the modification (replacing LSPs with zeros) is done on the active LSDB, then the IS-IS system copies the active LSDB (skipping previously inserted zeros) to the standby LSDB, then it activates the standby LSDB and deactivates the active LSDB. The switching between active and standby LSDBs occurs whenever necessary. The system inputs component are registers where the IS-IS inputs are stored. The muxDemux component represents multiplexers and demultiplexers that were hidden from Figure 2-25 to simplify the data path diagram.

OSPF

The work in [29] and [30] describes a hardware implementation of OSPF. OSPF is an interior gateway protocol and a link state protocol used to find the shortest path for an IP autonomous system (AS). Figure 2-26 illustrates a network where OSPF is used by all the routers labeled R1 through R10. The network consists of three areas (0.0.0.0, 1.1.1.1, 2.2.2.2). All areas are connected through a backbone area, whose ID must be 0.0.0.0. Area 1.1.1.1 represents a multi-access network hence a designated router (DR) must be elected (during auto discovery) to cut down on routing advertisement traffic and workload on routers. In some multi-access networks, it is preferable to have a backup designated router (BDR). An Area Border Router (ABR) is used to connect multiple areas. In Figure 2-26, R7 and R8 are ABRs used to connect areas 1.1.1.1 and 2.2.2.2 to the backbone respectively. An AS Boundary Router (ASBR) is used to exchange information with routers from other ASs. A link state advertisement (LSA) is used to describe the state of a router or network. The LSAs heard and collected from neighboring routers form a topological database. There are many types of LSAs in OSPF, however only four types were considered. The first type is the router link state advertisement which is broadcast to all nodes in an area. It contains the neighbor router's links. For example, in area 1.1.1.1, this type of LSA will be exercised by routers, R1, R2, R3 and

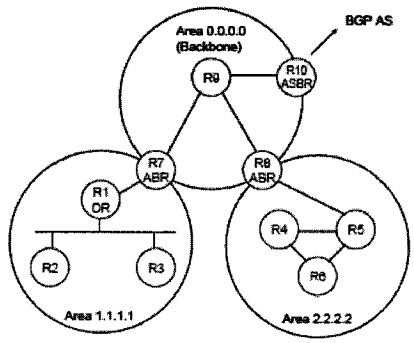


Figure 2-26. OSPF Network

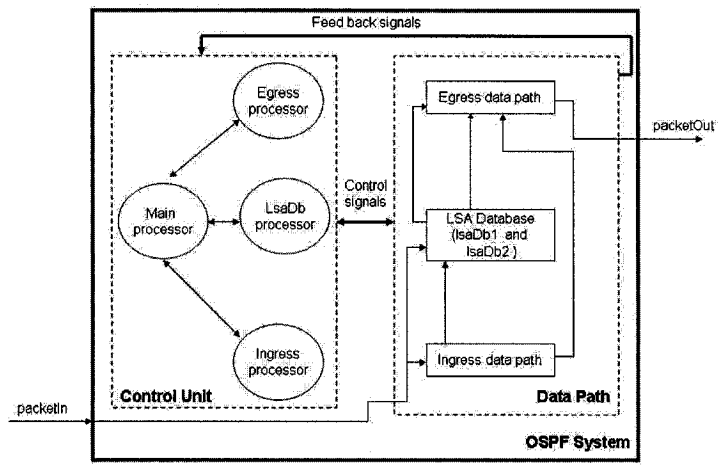


Figure 2-27. High Level Description of OSPF System

R7. The second type of LSA is the network link state advertisement which is flooded within an area by the DR. It includes information regarding all routers. The only router that generates a type two LSA is R1. The third type of LSA is the area summary link state advertisement. This LSA is flooded into an area by an ABR and describes the network accessibility from an outside area. Routers R7 and R8 advertise type three LSAs in Figure 2-26. The final type of LSA presented is the ASBR summary link state advertisement. This type of LSA is flooded into an area by the ABR and includes the cost from itself to an ASBR. Routers R7 and R8 advertise the final type of LSA in Figure 2-26.

Figure 2-27 describes the interaction between high level modules in performing OSPF tasks. The Ingress processor stores data going into the system while the Egress processor

performs packet assembly and sends packets out of the system. The tasks of the OSPF protocol are performed in the main processor module while the shortest path is computed in the shortest path processor (SPP) module.

The data path consists of four major units and they are the ingress data path, egress data path, SPP data path and LSA-Databases (active and standby). The active database contains the most recently processed LSAs while the standby database is only used during LSA updates, where additional LSAs require storing or deletion of old LSAs needs to occur. For example, in the scenario of adding a new LSA, the MP adds a new LSA to the standby LSA-DB, copies the active LSA-DB to the standby LSA-DB, and then it activates the standby LSA-DB and deactivates the active LSA-DB.

Signaling Protocol Hardware Research

This section summarizes the hardware research in the protocols used to establish LSPs. General frameworks are presented in addition to protocol specific research and the extent to which those protocols were explored.

Signaling Framework

The solution in [20] describes a general framework in which signaling protocols can be implemented in hardware. This section describes that framework and then elaborates on which protocols are in development.

Figure 2-28 illustrates the connection procedure assumed to be present for the signaling protocol. The network topology is known and a series of setup messages are passed from one end device to another through a number of switches. Messages indicating setup success are then passed from the target end device to the source end device. After all messages have been sent and successfully received a connection (or LSP) has been established that is used to transmit MPLS packets. At each switch the following steps are performed in determining where data will be sent.

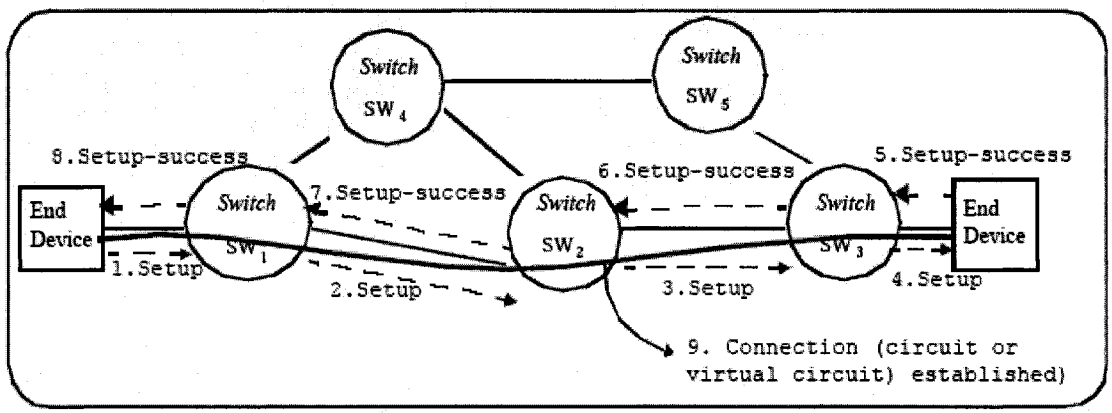


Figure 2-28. Signaling Framework Connection Setup

1. Check for availability of required resources (bandwidth, buffer space, etc.) and reserve them.
2. Assign labels for the connection.
3. Program switch fabric to map incoming labels to outgoing labels.
4. Set control parameters for scheduling and other run-time algorithms.

Once the connection has been setup, data arriving to the switch is forwarded according to the contents of the switch-mapping table (the result of step 3 listed above). Once data exchange is complete the connection is released similar to the way it was created in Figure 2-28. Release messages are sent from end to end, those messages are confirmed and any resources that were used in the connection are released.

The architecture shown in Figure 2-29 was used for the development of this framework. All data plane functionality would be programmed through the switch-mapping table. Control plane functionality is implemented either through a hardware accelerator or software on a microprocessor or FPGA. In addition to performing protocol specific functionality, the hardware logic interacts with all network interfaces directly. So any logic specific to those devices needs to be present as well. Since network interfaces can differ from device to device, lower level details of their manipulation are not presented in this document. Software processes required for initialization, maintenance, error handling, etc. are not discussed.

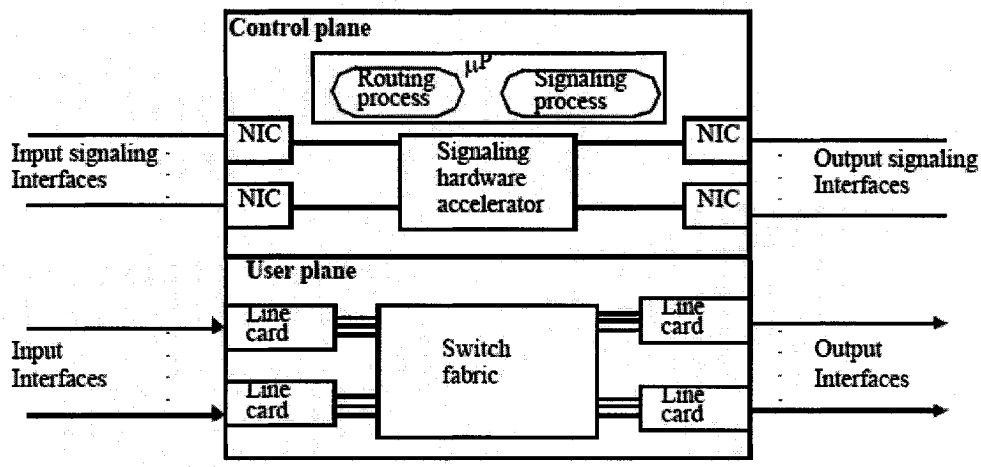


Figure 2-29. Signaling Framework Architecture

Since this framework is general to multiple signaling protocols, four messages related to connection management are defined. Those messages are Setup, Setup-Success, Release, and Release-Confirm (described in detail in Figure 2-30). More information is required to establish a message then to indicate its success or to release it. Consequently, the Setup message is the largest, mainly because it had three IP addresses (destination, source, previous node). All messages have information regarding message length, a previous connection reference, a current connection reference, message type, reserved data and a checksum. The setup-success message has bandwidth information while the release message has a field indicating the cause.

Those messages are processed using the state machine described in Figure 2-31. Initially the connection is in the *Closed* state. When a switch accepts a connection request, several steps are performed to create the connection. Those steps are allocating a connection reference to identify the connection, reserving the necessary resources including the labels, programming the switch fabric, sending the Setup message to the next switch and marking the connection state as *Setup-Sent*. When the switch receives a *Setup-Success* message for a connection, which means all switches along the path have successfully established the connection, then the state of the connection is changed to *Established*.

Release-Sent means the switch has received the *Release* message, freed the allocated resources and sent the outgoing *Release* message to the next node. When the switch

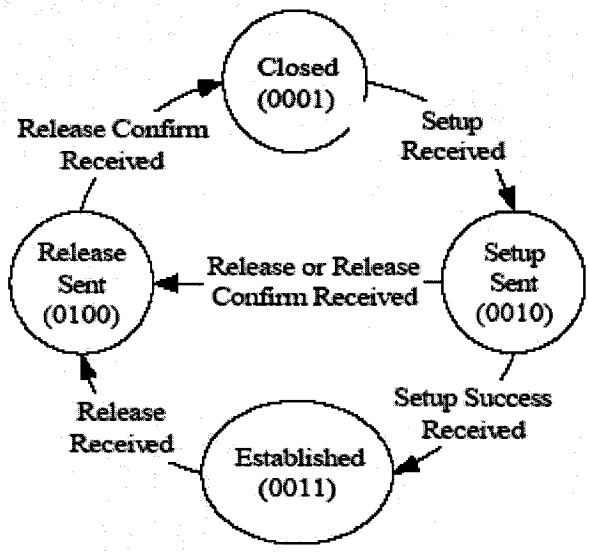


Figure 2-30. Signaling Framework State Machine

	Bit 31		16	15	Bit 0
Setup Message	Message Length	TTL		Msg.Typ (0001)	Connection Reference (prev.)
	Destination IP Address				
	Source IP Address				
	Previous Node's IP Address				
	Bandwidth	Reserved		Interface Number	Timeslot Number
	Pad Bits			Checksum	
Setup-Success Message	Message Length	Bandwidth		Msg.Typ (0010)	Connection Reference (prev.)
	Connection Reference(own)		Reserved	Checksum	
Release/ Release-Confirm Message	Message Length	Cause		Msg.Typ (0011/0100)	Connection Reference (prev.)
	Connection Reference(own)		Reserved	Checksum	

Figure 2-31. Signaling Framework Messages

receives the *Release-Confirm* message, the connection is successfully terminated and the state of the connection returns to *Closed.*

Figure 2-32 describes the tables that are used in the signaling protocol framework. Those tables are the Routing table, Connection Admission Control (CAC) table, Connectivity table, State table, and Switch-Mapping table. The *Routing* table is used to determine the next-hop switch. The index is the destination address; the fields include the address of the next switch and the corresponding output interface. The *CAC* table maintains the available bandwidth on the interfaces leading to neighboring switches. The *Connectivity* table is used to map the interface numbers used at neighboring switches to local interface

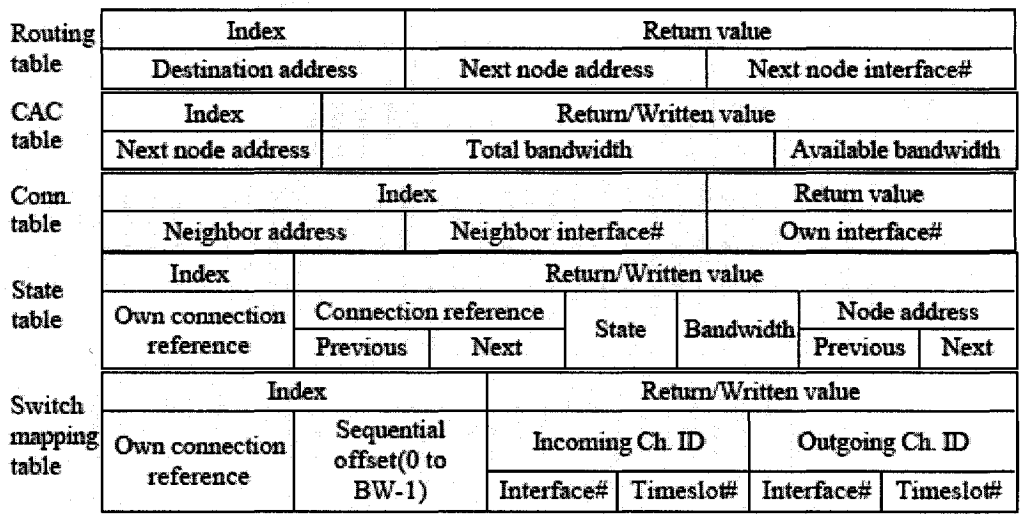


Figure 2-32. Data Tables for Signaling Protocol

numbers. This information will be used to program the switch fabric. The *State* table maintains the state information associated with each connection. The connection reference is the index into the table. The fields include the connection references and addresses of the previous and next switches, the bandwidth allocated for the connection, and most importantly, the state information.

These tables are intended to represent all information that a signaling protocol could need, but not necessarily all information a signaling protocol does need. An example would be to consider the basic implementations of LDP and RSVP. The concepts of traffic engineering and connection admission do not exist for those protocols. Consequently, the CAC table would not be necessary since it is assumed that all bandwidth is available.

Figure 2-33 an FPGA implementation of the signaling protocol is given. CPE0 and PE1 represent two separate FPGAs used to implement the protocol. FIFO0 and FIFO1 are First In First Out memory components used to store incoming and outgoing packets respectively. PE1 is used primarily as a controller for FIFO1 and has no other functionality related to the protocol. CPE0 has all other functionality including memory controllers for FIFO0 and the memory used to store the tables described in previous sections. It also contains the signal engine with the state machine to manipulate and control connections. Processing control is centralized in the signal engine to manipulate

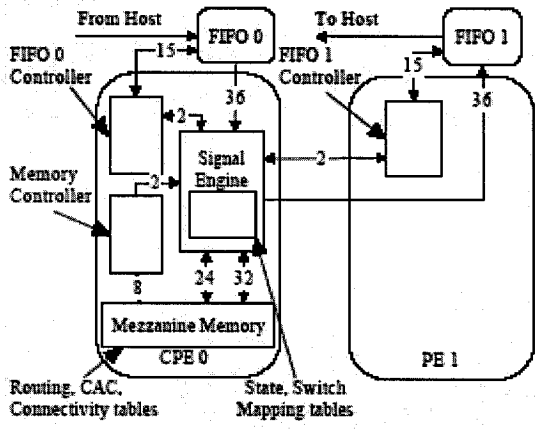


Figure 2-33. FPGA Implementation of Signaling Protocol

all other devices used to implement the protocol. The entire control component of the implementation is comprised of the signal engine, FIFO controllers and the memory controller.

The signal engine is intended to manage the connections in the manner shown in Figure 2-34. However, the implementation of the signal engine itself is a more complex matter with a state machine that is more elaborate and detailed. Figure 2-34 describes the state machine that is used in the signal engine. Messages are read from FIFO0 with messages delimited by the Message Length field. Once the checksum has been verified the state table is consulted to check the state of the connection. Messages are then processed accordingly to either setup, release or update the connection.

Processing a *Setup* message involves checking the Time to Live (*TTL)* field, reading the *Routing* table to determine the next switch and corresponding output interface, updating the *CAC* table, reading the *Connectivity* table to determine the input interface, allocating a connection reference to identify the connection, allocating timeslots and programming the *Switch-mapping* table. The *Setup-Success* message requires no special processing. The processing of the *Release* message involves updating the *CAC* table and releasing the timeslots reserved for the connection. When processing the *Release-Confirm* message, the allocated connection reference is freed and thus, the connection is terminated. After processing any message, the *State* table is updated. The new message is generated and buffered in FIFO1 temporarily, and then transmitted to the next switch on the path.

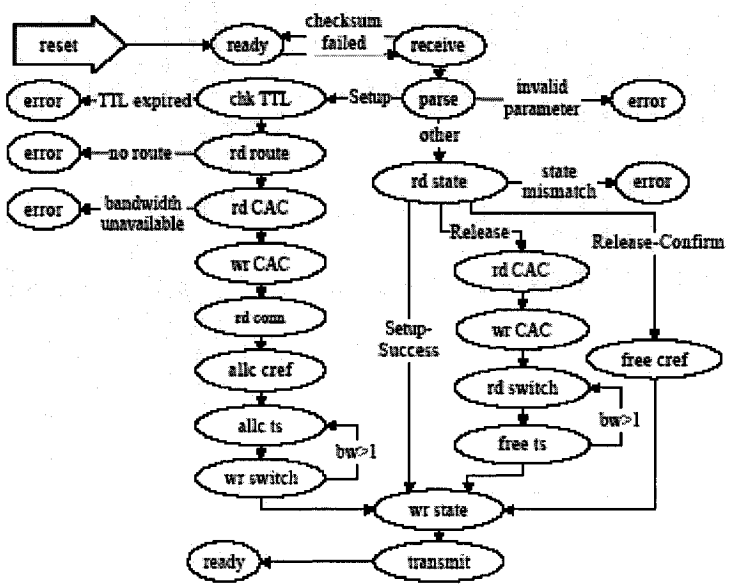


Figure 2-34. Signaling Engine State Machine

Figure 2-35 illustrates the implementation of a timeslot manager for the interfaces used to transmit packets. In the example, a time slot is desired for interface 3, so a lookup in a matrix occurs at that number. The vertical reference in the matrix is the interface number. The entry in the table is a 16 bit number where each bit represents a different time slot. A '1' indicates that the slot is in use while '0' indicates that the slot is available. So, for entry 3, the highest available bit is number 14 and the priority decoder generates number 14 with that entry. The table entry is also sent to what's called a scratch register where the corresponding bit can be changed to reflect the time slot being used. The new value is then written back to the table to update the timeslots being used by the interface. In this fashion the priority decoder will generate the timeslot for the given interface while the reference updates itself. Connections are managed in a similar fashion where a connection reference would be used instead of an interface and the earliest available reference would be generated from the table.

The approach described above can be applied to signaling protocols in general. However, as they are applied to specific protocols, certain changes must be made depending on the protocol in use and the extent to which the protocol is being implemented. GMPLS

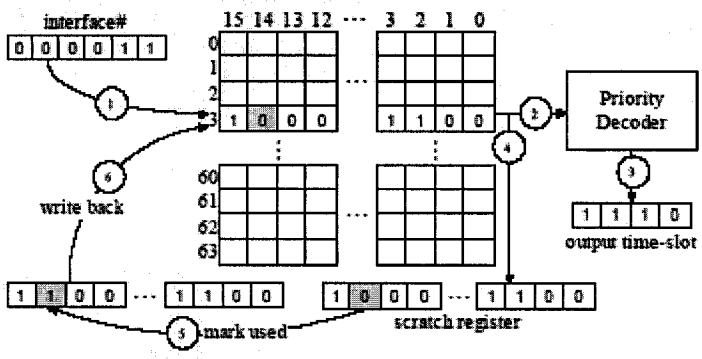


Figure 2-35. Timeslot management

extensions have been designed for RSVP-TE and CR-LDP using the framework described. Those extensions are described in [21] and [45] respectively. GMPLS extends MPLS to provide the control plane for devices that switch in any of these domains: packet, time, wavelength, and fiber. This common control plane promises to simplify network operation and management by automating end-to-end provisioning of connections, managing network resources and providing the level of QoS that is expected in the new, sophisticated applications. The following section describes relevant hardware developments for these implementations that have not been mentioned.

Connection establishment for RSVP-TE is identical to the way it was described in previous sections. However, RSVP has defined seven signaling messages. Those messages are *Path*, *Resv*, *PathErr*, *ResvErr*, *PathTear*, *ResvTear*, and *ResvConf*. RSVP-TE added the *Hello* message for node failure detection. When RSVP-TE was extended for GMPLS, the *Notify* message was introduced to support fast failure notification. Among these messages, *Path* and *Resv* messages are used to set up a connection while *PathTear/ResvTear* messages are used to tear down a connection. Only those four messages were selected for hardware acceleration because *Path* and *Resv* messages are needed for connection setup, the procedure targeted for speedup, and *PathTear/ResvTear* because resources need to be released at a pace corresponding to the fast setups. The remaining messages are relegated to software in this implementation. Only optional objects MESSAGE_ID, MESSAGE_ID_ACK, and SUGGESTED_LABEL were supported in hardware as they were deemed the most common and important.

Simulated results for the proposed framework illustrate that basic operations occur within approximately 100 cycles. Receiving and transmitting a setup message consumes 12 clock cycles each, processing of the setup message consumes 53 clock cycles. Consequently receiving, processing and transmitting a setup message requires 77 cycles. Processing the setup-success, release and release-confirm message requires about 70 cycles since the messages are shorter. With a 25 MHz clock a complete setup and teardown of a connection requires 6.6 microseconds. This performance demonstrates a potential 100x-1000x speed up over a software implementation. Calling handling for RSVP-TE messages has a capacity of 400,000 connections per second through pipelining. Specific results for CR-LDP have not been obtained at this point.

Signaling Paradigm

The solution in [10] proposes a design for a signaling paradigm called "Keep-It-Simple" Signaling (KISS). It is optimized for hardware signaling of unicast connections. So signaling protocols can be developed according to the paradigm and it would allow for fast, simple, and scalable hardware implementation. Conversely that hardware implementation would allow for optimized performance. A sample signaling suite was developed for MPLS based IP networks.

Figure 2-36 describes the message exchanged assumed to exist for a typical KISS connection. At the beginning of the first phase (1), the initiator (NIM 'A') sends a setup message towards the destination. This message may carry an IP source-route and a flow identification label (FIL). The message also carries a flowspec of the new flow. When 'B' receives the message, it verifies with its CAC unit that it has enough resources to accept the new connection. From the *setup* packet, it extracts 'A' address. 'B' is ready to accept the connection, so it sends back a *setup-ack* message to 'A'. This message carries the FIL from the *setup* message. The *setup-ack* message is acknowledged with the *setup-ack2* message. The connection was accepted so 'B' sends the *setup* message to the next hop ('C'). This message carry 'B' FIL2. The CAC unit can reduce the *flowspec* if it doesn't have enough resources. In phase (2), the destination accepted the new connection.

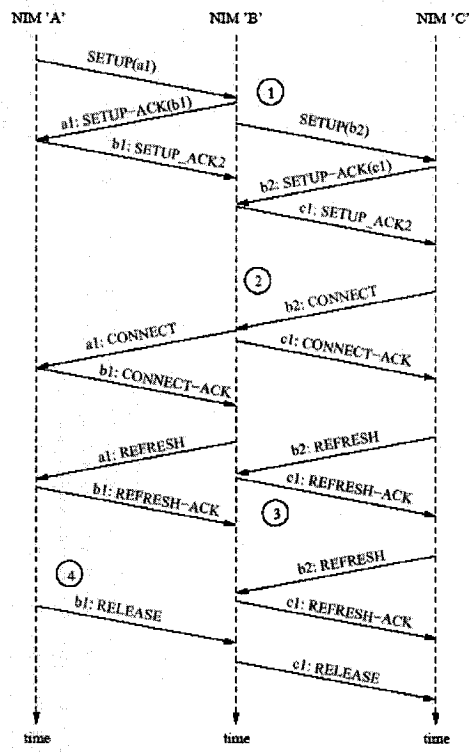


Figure 2-36. KISS message flow

It sends a *connect* message, which carries the adjusted *flowspec*. The message travels upstream toward the initiator (A), all the NIM along the way modify the amount of reserved resources according to this *flowspec*. The *connect* message is acknowledged with the *connect-ack* message. Note the FIL swapping has the message travels upstream. When the message reaches the initiator the reservation is active and it can start to send data. During the connection lifetime (3), the soft-states refreshed in relatively long intervals (30 seconds). The refresh messages are initiated by the NIM, in a hop-by-hop manner. When one party wishes to terminate the connection (4), it sends a *release* message. When a NIM receives this message, is releases all reserved resources and sends the message to the next hop.

Figure 2-37 describes a conceptual model of the KISS implementation. Packets are forwarded without software intervention using the speedpaths located at the top and the bottom of the diagram. A speedpath contains a packet classifier, packet scheduler forwarding table and Address Resolution Protocol (ARP) table. An ARP table is used to translate an IP address into a MAC address. Packet classification separate packets and

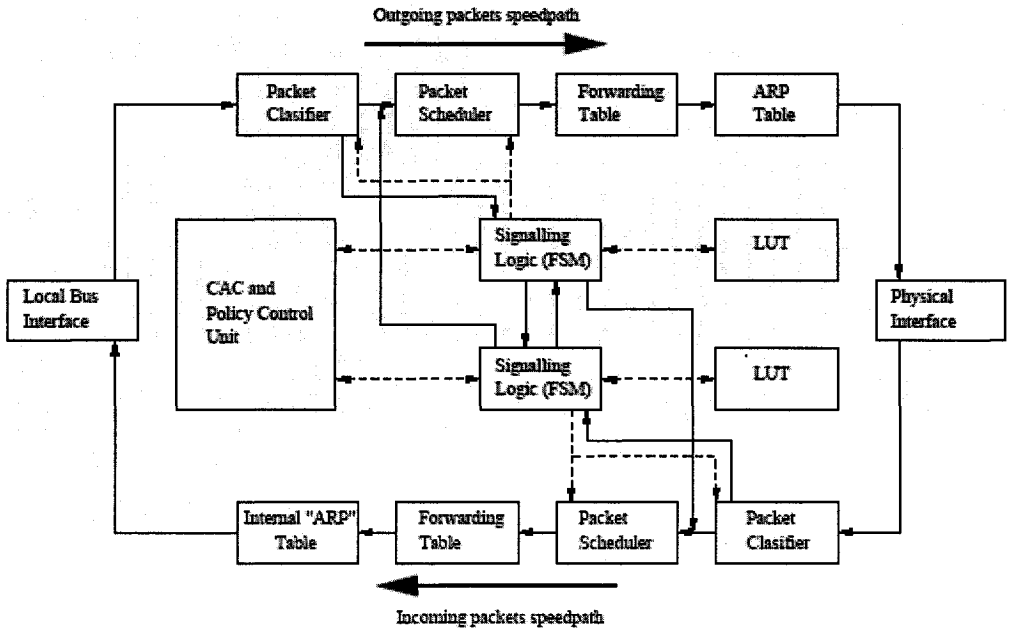


Figure 2-37. Conceptual model of KISS

routes them to queues that are separately processed. The packet scheduler is responsible for determining when packets should be transmitted. The forwarding and ARP tables are used to retrieve the correct information before the packet is sent. Signaling logic coordinates the movement of packets through both speed paths. There is one signaling logic module per speed path but there could have been one finite state machine (FSM) for both speedpaths. Lookup tables (LUT) are used to store flow state information. Those are accessed along with the Connection Admission Control (CAC) and policy control unit in the operation of the protocol.

The current unicast implementation of KISS uses approximately 180 bits for each flow state. 512 bits per flow would be sufficient for every CAC unit. LUTs are implemented in DRAM. Simulations illustrated that 559K SDRAM cycles per second to accommodate 200 connection tear-downs and 200 setups per second. An FPGA implementation at 133MHz SDRAM can support approximately 45,000 set up and tear down requests per second. In about 40 seconds (the average connection duration) approximately 1.8 million connections would be accommodated.

Data Plane Hardware Research

The work in [28] describes a hardware approach to perform MPLS data plane functionality in significant detail. Figure 2-38 provides a high level description of the MPLS hardware implementation. A custom hardware block interacts with an embedded CPU and a MAC chip to perform functions related to MPLS including label removing, binding and switching.

Figure 2-39 shows the second layer block diagram within the MPLS functional block, which contains 6 functional modules: Transmit Buffers for 8 outgoing ports, Receive Buffers for 8 incoming ports, Label Removing, Label Binding and Switching, Lookup Table, and State Machines/Service Schedulers.

The MPLS functional block has two dedicated unidirectional thirty-two bit wide data buses for transmitting and receiving data respectively. It also provides good architecture flexibility when in the future there is a need to reconfigure the MPLS functional block to support sixty-four bit parallel data transfer. The thirty-two bit input and output ports can be redefined as bidirectional ports however the bidirectional port feature is not presently implemented.

When outputting a packet, the MPLS functional block can provide the packet length, indicators of where the packet begins and ends and signals indicting if current data on the data bus are valid or not. Similarly, the MPLS block has to be provided with the same information when there is a packet arriving from the ingress side.

There is no LSP merging under discussion and thus there is no need for intermediate buffers at present. In the future, when intermediate buffers are added, the number of packets held within each buffer can be computed by setting constraints in packet delay time while controlling the probability of buffer overflow under a required level.

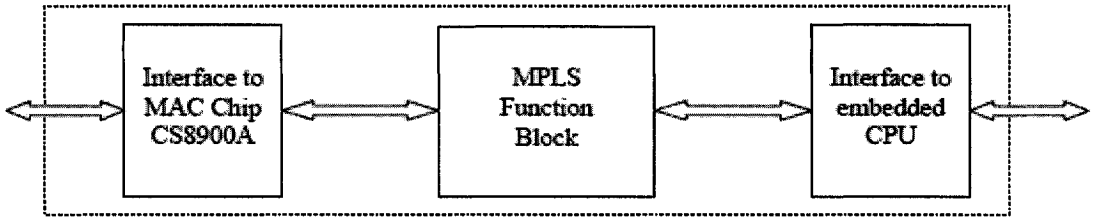


Figure 2-38. High Level MPLS Block Diagram

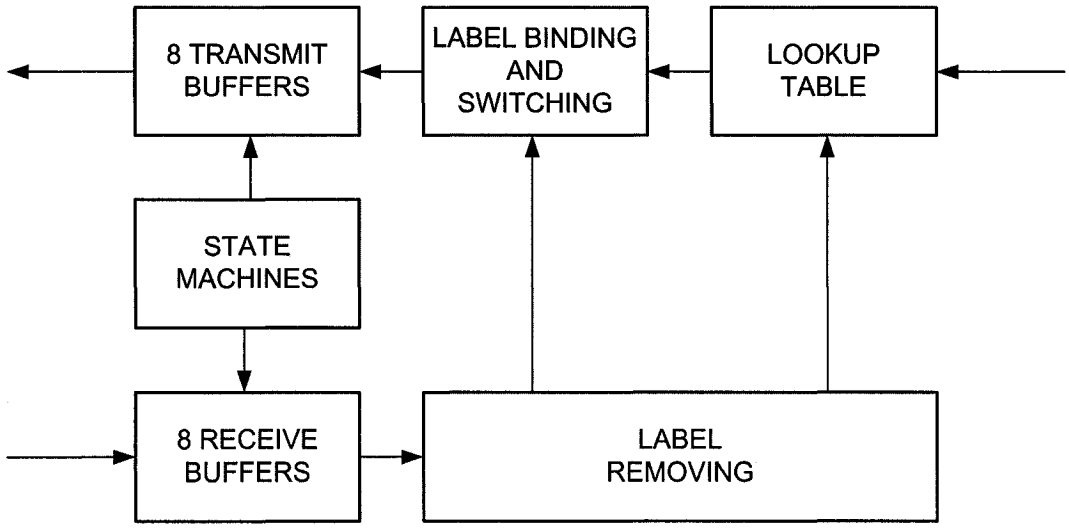


Figure 2-39. MPLS Function Block Description

Eight transmit and eight receive buffers are integrated to make the design more cost effective and suffer less data transfer delay. Each set of buffer space, for both receive and transmit, can be taken as the extension of that of a corresponding physical port in the media access controller CS900A. All transmit and receive buffers are realized in FIFO whose length is currently set to be 1600 bytes, which can hold several short IEEE 802.3 packets.

The interface for the local microprocessor, named as the ninth port, does not have its own buffer. This microprocessor is embedded and it can itself buffer the packet generated there. IP packets from local layer 3 are sent through the ninth port to the lookup table module directly. The lookup table module contains a corresponding MPLS label is assigned to the packet according to its IP header or other additional service requirements specified for a certain FEC the packet belongs to. The specifications are settled between customers and Internet service providers in advance.

Packets entering the node from the network side are first buffered at one of the 8 receive buffers waiting for their turns for further processing. At the Label Removing module, the label of the packet is stripped off and then this label is fed into the Lookup Table module as an index to find a new appropriate outgoing label for the packet. After the new outgoing label is ready and the outgoing port is determined, the Label Binding module binds this label to the packet coming from either the network side or the local microprocessor, and sends the packet to the corresponding transmit buffer, where the packet waits for its turn to get transmitted onto the Ethernet.

The state machine module is designed to control the service order and duration time for each port. Together with other signals, it regulates the working procedure of the whole system and keeps the other 5 second-layer modules cooperating together with a proper time schedule. Detailed tasks it completes include manipulating the procedure of checking 8 receive buffers and the microprocessor interface to see if there is any data ready for processing, and then having each port served to finish its label switching in an pre-determined order within its weighted service interval.

The lookup table module only talks with the label removing module to get necessary information and is completely separate from other modules. This organization provides a clear distinction between functional modules.

2.4 Conclusion

This chapter provided an overview of MPLS and RSVP-TE to describe relevant background information for the research presented in this thesis. Industry solutions to MPLS were summarized as well as open literature research with respect to hardware implementations. The following conclusions can be drawn based on existing research for MPLS and hardware implementations:

- Significant hardware research exists for intra domain routing. Therefore it may not be desirable to directly pursue further research in this area for MPLS but it would be advantageous to include functionality for routing protocols if possible for any hardware implementation of MPLS.
- General frameworks for the hardware implementation of a signaling protocol account for most of the functionality that must be provided, but implementations usually consider a subset of that functionality.
- Timeslot management for data transmission over several interfaces is possible in a modular manner but it is more difficult to maintain similar information for an arbitrary number of LSPs transmitted over several links.
- No hardware architecture is readily available integrating a signaling protocol with label management. Only one set of functionality is usually presented.
- Traffic engineering is unavailable in hardware implementations in open literature.
- No mention was made on the mechanisms necessary for hardware to interact with software and how those mechanisms would affect overall performance.

The remainder of this thesis describes the design, implementation and performance of a processor to perform the tasks of a signaling protocol, label management and TE. The processor is specifically designed to interact with a bus so it may be integrated into a larger embedded system. As previously stated RSVP-TE is the most common signaling protocol in commercial deployment and it will be used as the signaling mechanism in the processor.

Chapter 3

3 High Level System Description

This chapter fully describes the design of the MPLS processor. The processor is designed to perform signaling tasks, label management and TE. Therefore the architecture described below illustrates the mechanisms used to perform those tasks directly and other functions facilitating their efficient execution.

The processor's requirements are first listed to illustrate the functions that are performed. Specifications are then listed to demonstrate the processor's limitations and the architecture is provided to define system components and how they interact.

3.1 Requirements

Main requirements for the processor are based on the research presented in Chapter 2 and restated at the beginning of this chapter. Signaling tasks, label management and TE must be part of the processor's implementation. Additional requirements are listed below:

- The processor must be implemented to interact with software as to facilitate the creation of an application program interface (API).
- The processor must be able to process and generate PATH, RESV, PATHERR, RESVERR, PATHTEAR and RESVTEAR packets as described in section 2.2.3.
- The processor must allow for the entry of all required LSP data, TE data and router data.
- The processor must allow LSP data, TE data, router data and packet data to be reset individually and separately.
- The processor must provide mechanisms for TE information to be assigned arbitrarily to LSPs.
- The processor must automatically generate and update label information as LSPs are established. Mechanisms must also available to specify the first label value

that can be assigned and retrieve label information for the purposes of packet switching.

- The processor must be implemented in a manner such that the number of supported LSPs and can be easily increased without significant modifications.

3.2 Specifications

The design and implementation of the processor was performed according to the requirements above and the specifications below:

- Thirty-two LSPs and bidirectional ports are supported in the processor's memory.
- Ten incoming and outgoing packets can be stored in the processor's memory.
- Referring to Table 2-2, errors associated with codes 13, 14 and 23 are detected and all others are not supported.
- Only Fixed-filter LSPs are supported.
- Checksum calculations, QoS tasks, timing related tasks and IEEE floating point numbers (shown in Figure 2-13 and Figure 2-14) are not supported and presumed to be implemented in software.
- When entering object data, the objects may be entered in any sequence.
- The TTL is automatically updated when generating new packets and need not be done at the software level.
- TE information will be limited to the bandwidth available on the links in the network.

3.3 Architecture

Figure 3-1 provides a high level description of the MPLS processor. Data is written to the processor through the write and writedata signals. The read and readdata signals are used to retrieve data from the processor. The processor is asynchronously reset with the

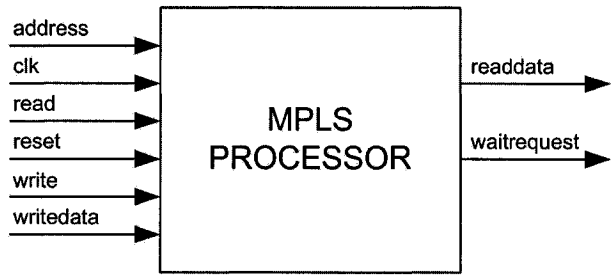


Figure 3-1. High Level Description of MPLS Processor

reset signal. Both readdata and writedata are thirty-two bits wide, the address signal is four bits and all other signals are one bit wide. The processor interacts directly with the Avalon bus architecture described in [4]. The waitrequest signal indicates when the processor is performing a task. Consult Appendix A for a test bench illustrating how these signals are used to perform various tasks with the MPLS processor.

Table 3-1 describes how the address is interpreted depending on whether the user is reading or writing data. Data for LSPs, links, packet data and configuration are written while packet data, status information and labels are read. Full descriptions of the register formats associated with the addresses are provided later in this chapter when the relevant modules are discussed.

Figure 3-2 describes the format of the control register. Four bits are used for a command and five bits are used as an LSP reference when necessary. The LSP reference must correspond to the sequence of LSPs that has been entered with the numerical reference beginning at zero. Therefore, if the number three is put as the LSP reference then it will refer to the fourth LSP that was successfully saved.

Table 3-2 shows the formats and definitions for the user commands available in the control register. The input and output stack can be separately cleared as well as LSP module and the TE DB. Users can also create PATHTEAR or RESVTEAR packets on command, process a saved packet, establish LSPs (generate PATH packets given the network topology) and save various data types.

Table 3-1. MPLS Processor Address Values

Value	Type of data	Read/Write	Description
0	LSP	Write	Destination address
1	LSP	Write	Source address
2	LSP	Write	Tunnel and LSP IDs
3	LSP	Write	Refresh rate
4	LSP	Write	Peak data rate
5	LSP	Write	QoS data (excluding peak data rate)
6	Link	Write	Next hop IP address
7	Link	Write	Bandwidth
8	Link	Write	Port IP address
9	Link/LSP	Write	Upstream, downstream, TE DB ports
10	Configuration	Write	Packet flags and initial TTL
11	Configuration	Write	Initial label
12	Packet	Write	Packet data
13	Configuration	Write	Control register
0	Packet	Read	Packet data
1	Status	Read	Status register
2	LSP	Read	Input label
3	LSP	Read	Output label
Others	N/A	N/A	N/A

Figure 3-2. MPLS Processor Control Register

Figure 3-3 describes the components of the processor and their interaction. The high level logic module coordinates all other modules and the Avalon bus to satisfy all the processor's requirements. Functions directly supported by the high level logic module are interaction with software, data entry and data removal. All connections to the high level logic module are not shown to clarify Figure 3-3. The following hardware components have also been developed for the processor: the packet parser, the packet generator, the error generator, the traffic engineering database, the label generator, the label database, the LSP module, and external data module.

Modules collectively used to satisfy the requirement of processing and generating the mandatory set of RSVP-TE packets are the packet parser, packet generator, error generator and label generator. The packet parser is used to process an entire packet,

Table 3-2. User Command Formats for Control Register

Value	Command
0	Reset processor
1	Clear input stack
2	Clear output stack
3	Clear LSP data
4	Clear TE DB data
5	Create PATHTEAR packet
6	Create RESVTEAR packet
7	Process packet
8	Establish LSPs
9	Save LSP data
10	Save TE DB data
11	Save packet data
Others	No command

separating all the individual components into addressable parts and checks for a predetermined set of errors. All packets are created with the packet generator given the type and various possible data sources. Those sources include every other component of the processor so the packet generator has logic to differentiate between the appropriate data source and value in forming a packet correctly. The error generator takes its inputs from the packet parser and sends an output to the packet generator to indicate if there is an error. Values sent to the packet generator are error codes documented in section 2.2.4. The label generator generates a previously unused number as a label. A subset of labels that are reserved according to the MPLS specifications. The first label that can be assigned is stored in the label generator.

All requirements relating to TE data are provided by the traffic engineering database (TE DB). The TE DB contains all information for the router's bidirectional links. That information includes IP addresses, internal port identifiers, reserved bandwidth and remaining available bandwidth. It is also possible for a user to specify the percentage of total bandwidth that can be reserved per link.

The label database facilitates the requirements for storing, retrieving and updating label data. References are maintained for each LSP and its corresponding pair of labels. Those labels are for packets headed upstream and downstream on the LSP so the correct

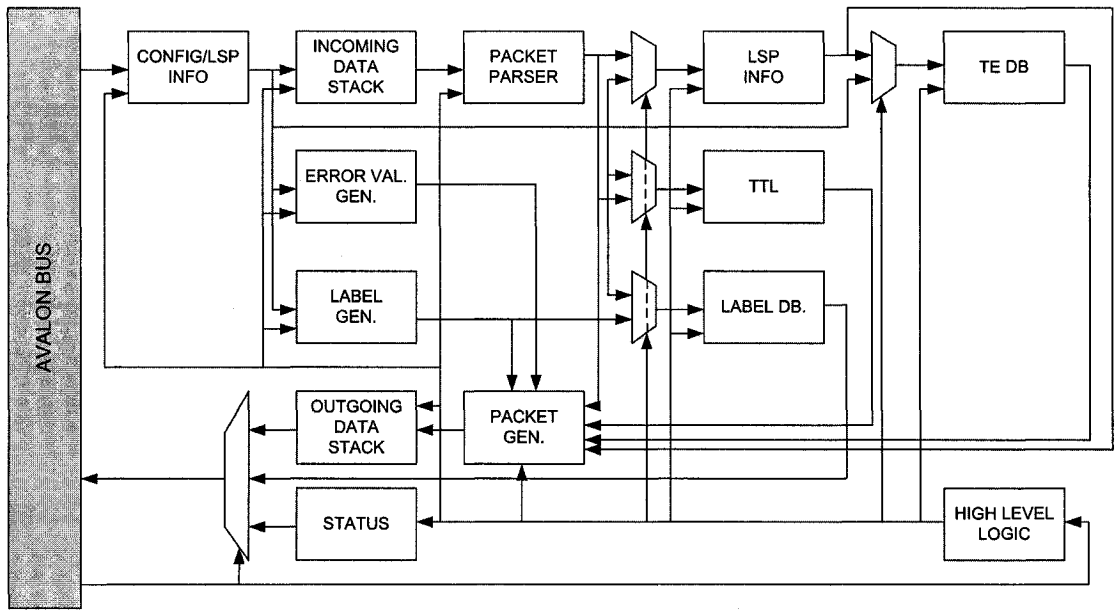


Figure 3-3. High Level MPLS Design

value can be read by the user or switched into the packet depending on the direction of traffic.

Before data is directed to its appropriate module it is stored in an external data module, denoted 'Config/LSP Info' in Figure 3-3. Connections from this module to all other modules are not shown to simplify Figure 3-3. Information to configure the processor (i.e. router type, etc.) are processed by the high level logic module but not redirected to a separate module.

Chapter 4

4 Detailed Design

This chapter fully illustrates the modules described in section 3.3 of the previous chapter. Given the requirements of the processor certain components were not readily available for use in the high level components. Those components are described first and the high level components described in the previous chapter are explained in explicit detail.

4.1 Custom Components

The modules described in later sections were implemented with components largely available through Altera Megafunctions [5]. Some components were separately designed and implemented because they were unavailable or did not exist. The subsections below describe those parts in further detail.

4.1.1 Demultiplexer

There are times when it is necessary for multiple signals to be available at one input. Instead of designing a component with multiple inputs, when only one can be used at any time, a separate component is connected at the input to determine which input will be used at any time. That module is called a multiplexer and Altera provides automated means of creating multiplexers of arbitrary size for the hardware developer.

Sometimes it is necessary to have a have a module performing the opposite function of a multiplexer. Such a module would take a single input and direct it to one of many potential outputs. That module is called a demultiplexer and cannot be created through automation. Consequently it was designed and implemented as shown in Figure 4-1.

Data is connected to a series of registers, each controlled through a decoder. The demultiplexing address is the input of the decoder. When the desired address is selected,

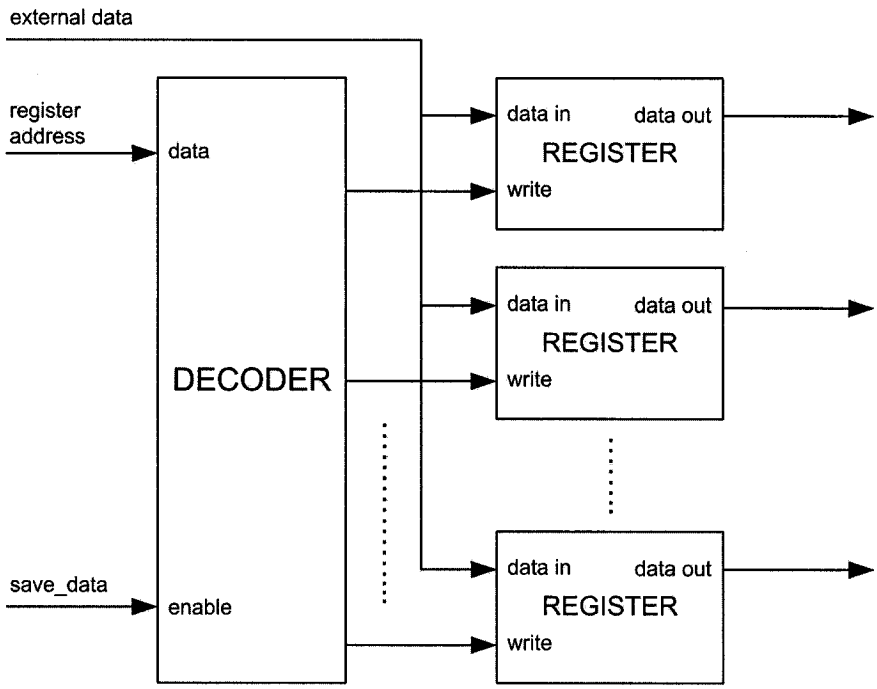


Figure 4-1. High Level Design of Demultiplexer

the data connected to all registers will only be saved in the desired register. Once the operation is complete only the corresponding register output changes.

4.1.2 Search Module

Depending on a module's functionality, it may be necessary to perform a search for desired data. A search module was designed so that function would be provided in a modular manner. The following simplifications and conditions were made so the search could be done in as efficient a manner as possible.

- The search algorithm is a linear search.
- Data being compared are in two RAM modules with the same size and interface.
- It is only necessary to iterate through one RAM module so every value can be compared against a set value in the other module if necessary. It is presumed that the read signal for the RAM module being iterated is constantly asserted, since it is not necessary to deassert it at any time during the search.
- All data being searched is stored in RAM in consecutive addresses starting at the first address.

- The data being compared is fed to a comparator. The output of the comparator is processed as opposed to the actual data.
- The search module manipulates a counter, whose interface is defined, that acts as a read address for a RAM.

Figure 4-2 describes the state machine used to implement the search module with the condition specified above. Once the search module is idle, a search can begin at which point the search index is cleared and a check is performed to see if the item has been found. From that point on, the search index is incremented every time the data is not found and a check is performed again. The search ends when the desired item has been found or all items have been searched. No other action is necessary once the search ends. The states for the search module are described in Table 4-1. States to clear and increment the search index require two cycles so the new index value can propagate and be used to compare indices. No other states are used to manipulate data elements and only require one cycle for completion.

All signals used for the search module are summarized in Table 4-2. The only outputs are index_ctrl, used for indirectly manipulating the memory address during the search. All other signals are inputs, informing the search module of when to begin, when to finish and if the search has been successful.

The search algorithm is identical in every instance where the search module is used but changes may occur with respect to the interface.

4.2 High Level Modules

The modules shown in Figure 3-3 are the high level modules of the processor and are used to perform every task. Explanations for those modules are provided through the subsections below with descriptions of their control units and data paths whenever possible.

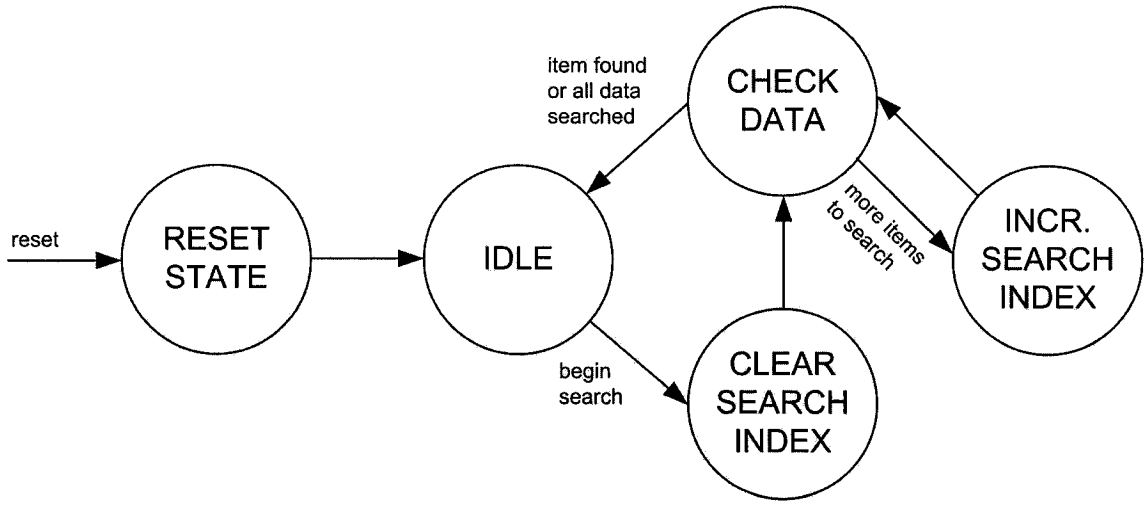


Figure 4-2. State Machine for Search Module

Table 4-1. States for Search Module

Name	Number of Cycles	Description
CHECK DATA	1	Search indices are compared to determine if a match has been found.
CLEAR SEARCH INDEX	2	Reset the search index so the entire memory space can be searched.
IDLE	1	The default state where no functions are being performed.
INCR. SEARCH INDEX	2	Increment the current index so the next element can be compared in the search.
RESET	1	Used to reset the state machine.

Table 4-2. Signals for Search Module

Name	Width (bits)	I/O	Description
begin_search	1	Input	Used to begin the search.
clk	1	Input	Clock signal.
enable	1	Input	Enables state machine functionality.
index_ctrl	2	Output	Control signals to manipulate the counter used for the index.
item_found	1	Input	Indicates that the indices match and a match has been made.
reset	1	Input	Resets the state machine.
search_done	1	Input	Indicates that all possible items have been searched.

4.2.1 Error Generator

Figure 4-3 illustrates a simplified state machine for the error generator. There is no so all the functionality is performed with a state machine. The error generator can be reset through the reset signal, after which an IDLE state is occupied until instructions are received to generate an error. If the error generator is enabled and is instructed to generate an error, inputs describing the packet contents and the type of error (if any) determine the following state for generating the error code and value. After those values have been produced the error generator occupies the IDLE state. The states are fully described in Table 4-3.

Table 4-4 summarizes the signals used in the error generator state machine. There are several inputs describing the packet and the type of error that may exist. Using that information, error values and codes are generated when instructed to do so with the generate_error_values signal. Outputs of the error generator are the error values and signals indicating if the error generator is idle or has found an error.

4.2.2 External Data Storage

Figure 4-4 illustrates a high level description of the module used to store external data. Several pieces of data are necessary however only thirty two bits are available may be used at once. So data must be entered thirty-two bits at a time. Data pertains to an LSP, network link or configuration information. So that LSP, link or configuration data must be entered and saved before proceeding further.

The data to store is fed to a series of registers but is only stored in one register. All register write values are accessed through a decoder that is used to select the desired register for data storage. An address for referencing the registers is the data value for the decoder. Once the data is available and the address has been set, the decoder is enabled so the correct register is chosen to store the data while all other registers are unaffected.

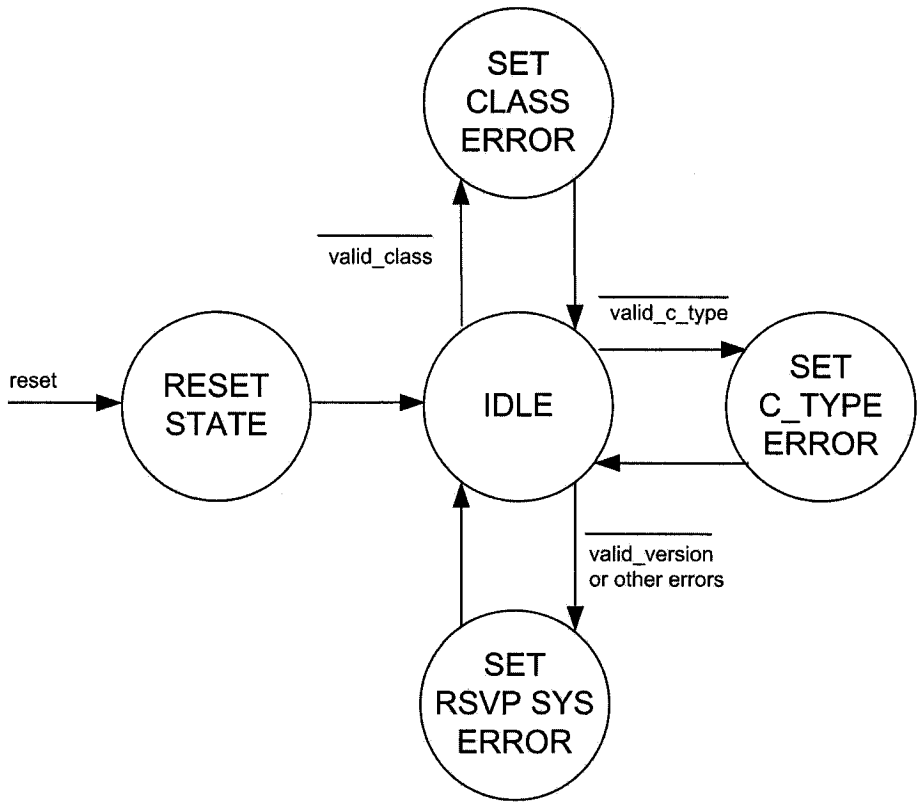


Figure 4-3. State Machine for Error Generator

Table 4-3. States in the Error Generator State Machine

Name	Number of Cycles	Description
IDLE	1	The default state where no functions are being performed.
RESET	1	Used to reset the state machine.
SET_C_TYPE ERROR	1	Set the error and c type values for a c type error.
SET_CLASS ERROR	1	Set the error and c type values for a class error.
SET_RSVP_SYS ERROR	1	Set the error and c type values for a system error.

The structure resembles random access memory (RAM) where data can be stored and retrieved from an addressable location. However with RAM only one data component can be accessed at one time. Several data values related to LSPs, links or other aspects of MPLS are required to perform certain operations. Retrieving that data in a serial fashion through RAM would have limited overall performance. Consequently all data was made available simultaneously through a series of registers.

Table 4-4. Signals used in the Error Generator

Name	Width (bits)	I/O	Description
clk	1	Input	Clock signal.
discard_packet	1	Input	Used to indicate that a packet is being discarded.
enable	1	Input	Enables state machine functionality.
error_code	8	Output	Error code for the packet.
error_found	1	Output	Indicates that an error has been found.
error_value	16	Output	Error value for the packet.
idle	1	Output	Indicates if the state machine is currently performing an operation.
generate_error_values	1	Input	Used to begin process of generating an error code and error value.
packet_contents	16	Input	Describes the contents of the packet. Each bit represents a potential object. A high value indicates that the object is present and a low value indicates otherwise.
packet_contents_valid	1	Input	Indicates if the packet has the correct sequence of objects.
reset	1	Input	Resets the state machine.
valid_c_type	1	Input	Indicates if the packet has any invalid c type numbers for any objects. A high value indicates that an error exists and a low values indicates otherwise.
valid_class	1	Input	Indicates if the packet has any invalid class numbers for any objects. A high value indicates that an error exists and a low values indicates otherwise.
valid_version	1	Input	Indicates if the packet has a valid version number. A high value indicates that an error exists and a low values indicates otherwise.

The input signal values for the external data module are described in Table 4-5. The register_address and writedata signals correspond to the address and writedata signals in Figure 3-1. Data is saved into a specific register through the save_data signal.

Output signals for the external data module are described in Table 4-6. These outputs relay data for every other component in the processor except the error value generator. Most data in the storage module is for LSPs and links so there are more connections going from this module to the LSP module and the traffic engineering database.

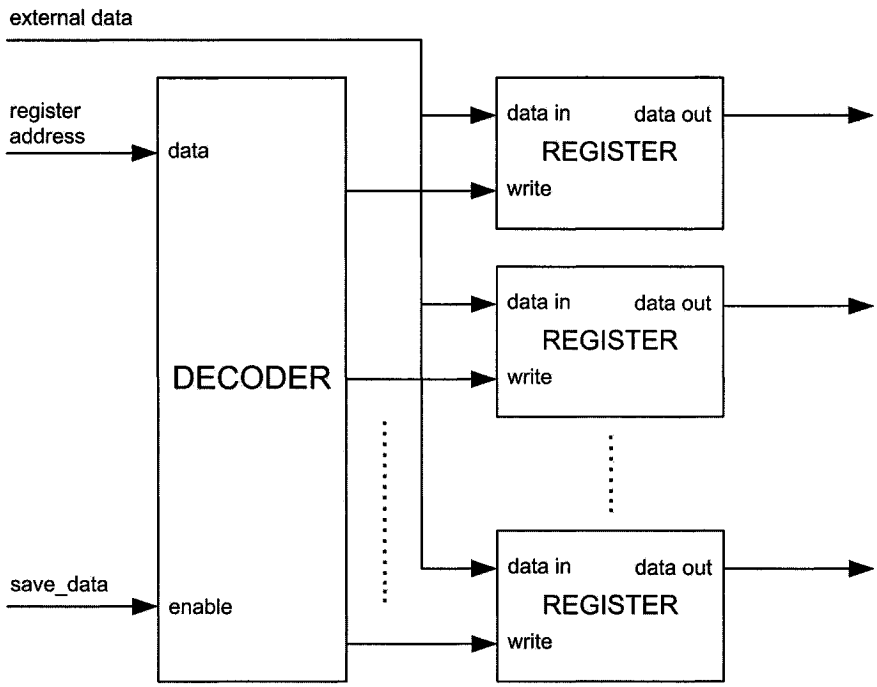


Figure 4-4. External Data Storage Architecture

Table 4-5. Input Signals used in External Data Storage

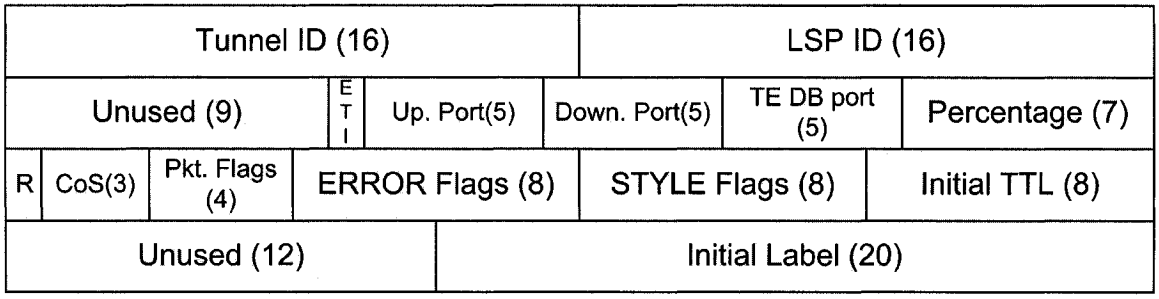
Name	Width (bits)	Description
clk	1	Clock signal.
register_address	5	Value describing the value to be saved.
save_data	1	Used to save data into its appropriate register.
writedata	32	The data that will be saved into a register.

Several values are less than thirty two bits long and can be saved simultaneously. These values are grouped into register groups to minimize the number of save operations and unused bits. These registers are described in Figure 4-5. Each row represents a separate register where the most significant bit (MSB) is to the far left and the least significant bit (LSB) is to the far right. Five registers are used to store values less than thirty two bits. They are organized so a single register has information only pertaining to an LSP, link or configuration as much as possible. Consequently there are a significant number of unused bits. The number beside each value indicates the total number of bits used. Values with no numbers use one bit. That number was not shown due to space constraints.

Table 4-6. Output Signals used in External Data Storage

Name	Width (bits)	Description
cos	3	Class of Service bits
bandwidth	32	Bandwidth value for a network link.
destination_address	32	Destination address for the LSP.
downstream_port_id	4	The ID of the port where data will be sent downstream for the current LSP.
extend_tunnel_id	1	Indicates if the extended tunnel ID will be zero or the LSP source address. A high value indicates that the source address will be used and a low value indicates that zero will be used.
flags_error	8	Flags value for the ERROR_SPEC object.
flags_packet_header	4	Flags value for the packet header. For the sake of simplicity it is presumed that all packets will contain the same flags value.
flags_style	8	Flags value for the STYLE object.
initial_label	20	This value indicates the first label value that will be used in label assignments.
initial_ttl	8	Indicates the initial TTL value of packets originating at this router.
lsp_id	16	The ID for the current LSP.
next_hop	32	The IP address used by the router receiving data on the current link.
peak_data_rate	32	The amount of bandwidth reserved on the current LSP.
percentage	7	The percentage of bandwidth on the current link that may be used to allocate resources to LSPs. The number used is the maximum of 100 and the number provided or zero.
port_addr	32	The IP address associated with the port of the current link.
rdaddress	5	A reference to generated labels that can be read by the user.
refresh	32	The time interval between which packets must be received so LSP resources can remain reserved.
router_type	1	The type of router being emulated. A high value indicates that it is an LSR and a low value indicates that it is an LER.
src_addr	32	The source address of the current LSP.
te_db_port_id	4	The internal ID of the port for the current link.
tunnel_id	16	The ID of the tunnel for the current LSP.
upstream_port_id	32	The ID of the port where data will be sent upstream.

MSB LSB



Legend:
ETI: Extend tunnel ID preference
R: Router type
U: Unused

Figure 4-5. Register Formats for Data less than 32 bits

The register in the top row contains the tunnel ID and LSP ID for a specific LSP. The second row shows the format for a register containing data for both an LSP and a link, the only such register to do so. From the most significant bit (MSB) onward, there are 12 unused bits followed by a bit to denote the choice of extended tunnel ID. A value of zero indicates that the extended tunnel ID should be 0.0.0.0 and a value of 1 indicates that the LSPs source address should be the extended tunnel ID. Upstream, downstream and TE DB port numbers are all represented with five bits each. Seven bits are used to represent the percentage of data on the TE DB port allocated for LSP reservation. Any value greater than 100 given for the percentage is discarded and substituted with 100.

The third and fourth register all contain data that only needs to be written once while the last register is used explicitly for configuration and management of the overall architecture. The router type is described with one bit where zero indicates an LER and one indicates an LSR. Data for CoS, an initial TTL value and all flags are in one register. The RSVP-TE specification has flags for the packet header, the ERROR_SPEC object and the STYLE object that can all be specified by the user if so desired. A separate register is used for the first label that can be used in label assignments.

4.2.3 High Level Logic

The high level logic of the processor is probably the most complex component among high level components. It coordinates all other modules to perform processor tasks. Due to the module's complexity the signal table will not be presented here but the state machine will be discussed.

Figure 4-6 illustrates the state machine used to coordinate the processor's modules. All actions are performed by reading or writing data. Types of data that are read are stack data, label data, IP addresses and a status register.

When a write signal is received the command is verified and the state corresponding to the command from Table 3-2 is used to execute that command. The entire processor may be reset and so may a specific module (the input stack, output stack, LSP module or TE DB). A TEAR packet (PATHTEAR or RESVTEAR) may be generated, different data may be saved, LSPs may be established and a packet may be processed. When the FPGA is asynchronously reset all modules are reinitialized in the RESET state. All tasks require different amounts of time to execute. However establishing LSPs and processing a packet require the most sophisticated logic and processing time. The states for those functions are now described in further detail.

Figure 4-7 describes the high level state machine summarizing the process of establishing an LSP. The process is for an LER to iterate through all the LSPs and find those whose ingress begins at the router. The traffic engineering database is checked to see if the link described with the LSP information does exist. If the LSP exists and there are sufficient resources exist then a PATH packet is generated. The process repeats until all LSPs have been checked and all possible packets have been generated. Once the process is complete the status register is updated and the high level module becomes idle. Most high level states are of variable lengths because other high level modules are used to perform actions and the user must read data, which may take an indeterminate period of time.

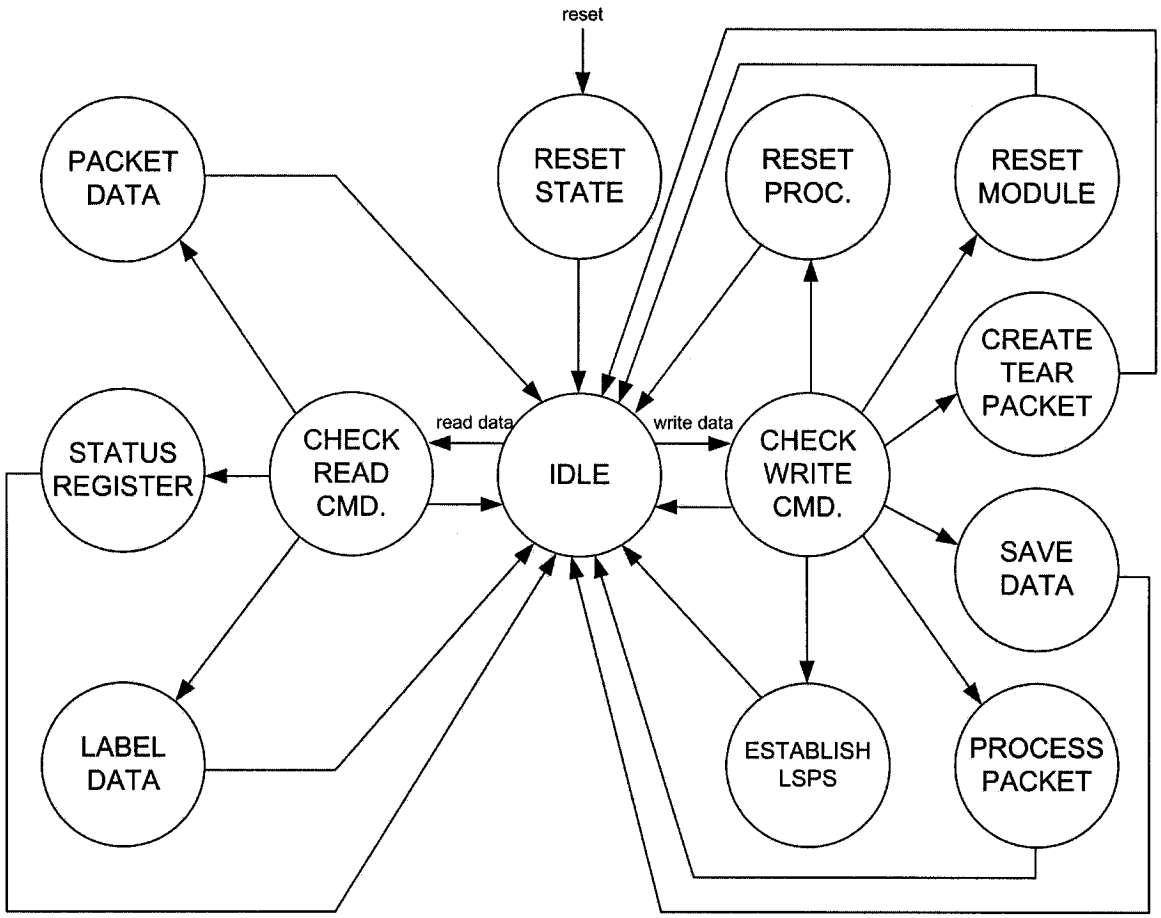


Figure 4-6. High Level Logic State Machine

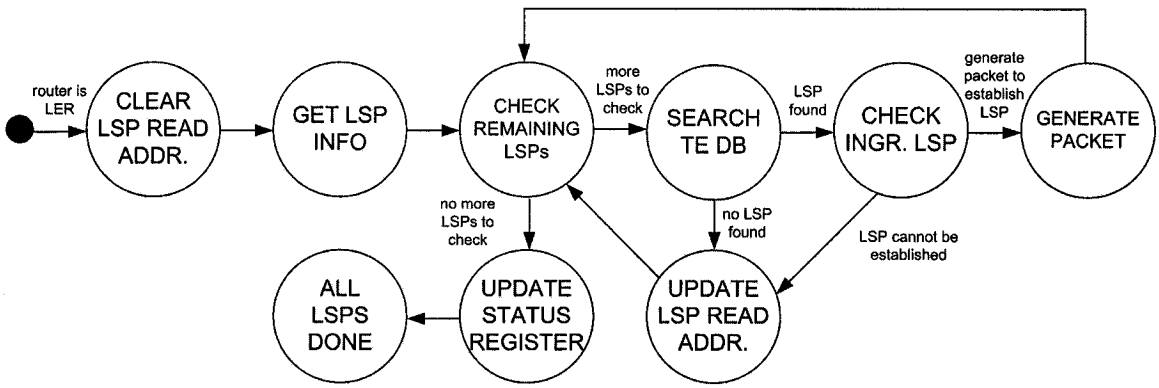


Figure 4-7. State Machine for Establishing LSPs

The procedure for processing a packet is illustrated in Figure 4-8. A packet is parsed with the parser described in section 0. If the version type is supported the TTL is checked and the packet type is retrieved if the TTL has not expired. If the packet type is supported all the LSP information in the packet is retrieved and the LSP module is

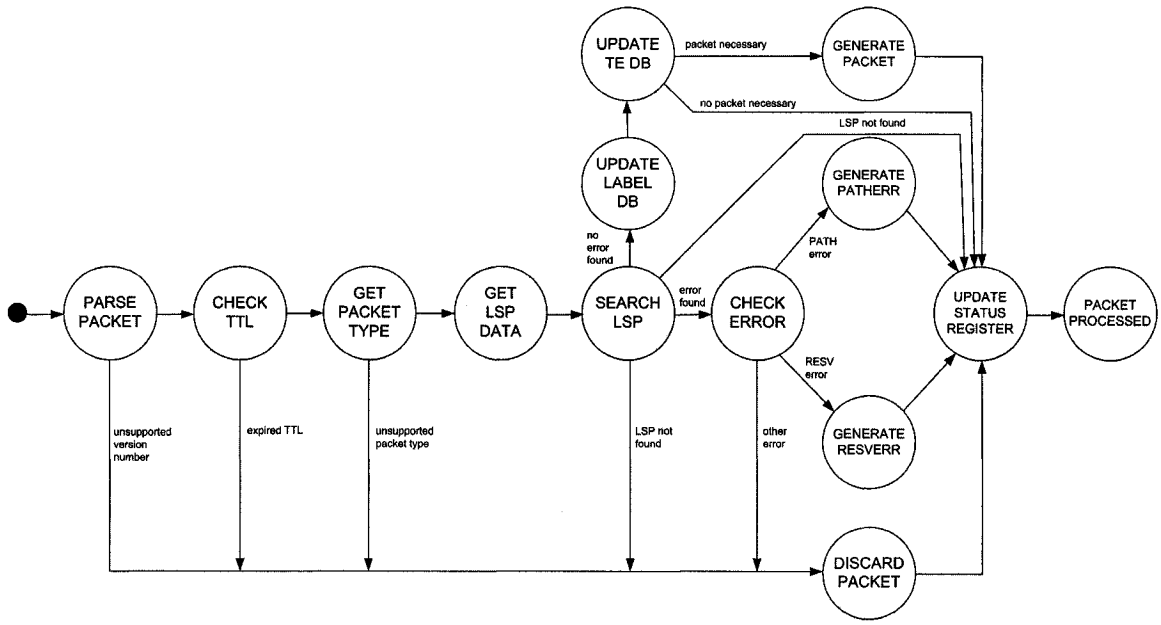


Figure 4-8. State Machine for Packet Processing

searched. Once the LSP has been found and no errors have been detected the label database and TE DB are updated if necessary. A packet is then generated if necessary and the status register is updated to conclude packet processing.

Packet errors are checked after the LSP search however the error value generator is used to generate the appropriate values immediately after the packet has been successfully parsed. No separate state is required because the error value generator uses data from the packet parser only. Once an error is found a PATHERR or RESVERR may be generated or the packet may be discarded with no response.

There are numerous scenarios where packet may be generated after parsing a packet, however they are not all shown in Figure 4-8 to simplify the diagram. Those scenarios depend on whether the router is an LSR or LER. LSRs mainly switch packets and mainly generate the same packet it receives and sends it upstream or downstream. LERs only generate PATH or RESV packets (other packets are received but not retransmitted). After the appropriate action has been performed the status register is updated and all operations stop.

4.2.4 Label Generator

The label generator is used to generate previously unused labels for LSPs. The labels can be used for incoming packets. It is presumed that there is a single, global label space for use on a per router basis, as opposed to a global space for the entire network. So a given label can have as many distinct meanings as there are routers in the MPLS network.

The procedure to generate labels can be arbitrary so long as they are unique. The simplest method of generating unique labels is to start with an arbitrary number and increment it to generate a new label. Using that method, a round-robin incrementing approach can be used whereby a number is incremented until a maximum is reached, at which point the minimum number is used and incrementing begins again. Labels cannot be negative numbers and according to [13] the numbers zero through fifteen inclusive cannot be used as labels. Since a label is twenty bits long the label space is from sixteen to $2^{20} - 1$.

Control Unit

Figure 4-9 describes the control unit for the label generator. Once a signal has been received to perform that task, checks are performed to see if any labels are available or if the maximum label number has been assigned. If neither of those scenarios exist a new label is generated by incrementing the previous value and the operation is complete. If the maximum label has been assigned then the lowest possible number is generated as the next valid value. When there are no more valid labels, no action is performed and a signal is asserted to indicate that no action was performed and that no more labels can be generated.

Table 4-7 describes the states for the label generator control unit. Comparing the label against the maximum value requires a maximum of two states because a separate state is required to indicate that there are no more labels available.

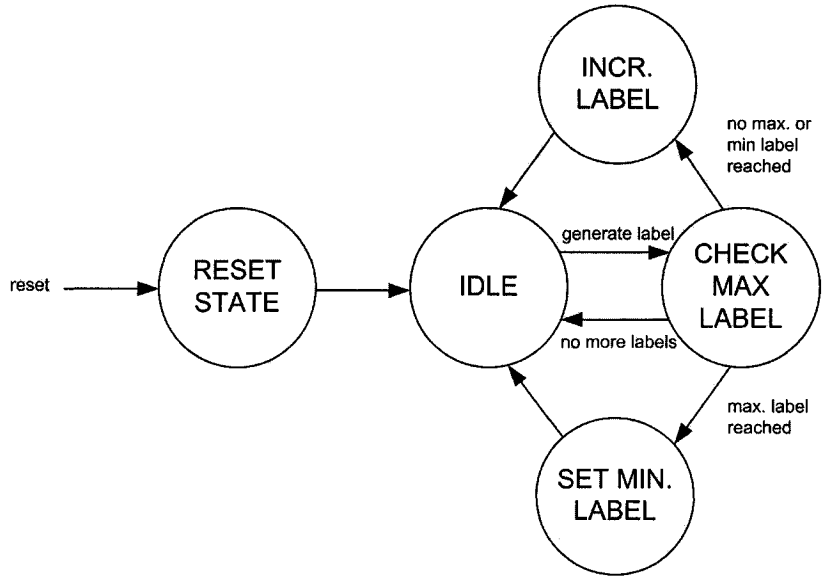


Figure 4-9. Control Unit for Label Generator

Table 4-7. States for Label Generator Control Unit

Name	Number of Cycles	Description
CHECK MAX LABEL	2	State where it is determined if the maximum label has been reached.
IDLE	1	The default state where no functions are being performed.
INCR. LABEL	1	State where a new label is generated by incrementing the previous label value by one.
RESET	1	Used to reset the state machine.
SET MIN LABEL	1	State where the current label is set as the minimum label.

All signals forming the interface of the label generator control unit are described in Table 4-8. Input signals are mainly feedback from the data path and the signal to begin signal generation. Outputs manipulate the counter for generating labels and indicate that no more labels can be generated when that is the case.

Data Path

The data path for the label generator is described in Figure 4-10. It is composed of a counter, for generating unique labels and comparators serving different purposes. The comparator and multiplexer feeding the data signal of the counter serve to set the first

Table 4-8. Signals for the Label Generator Control Unit

Name	Width (bits)	I/O	Description
clk	1	Input	Clock signal.
enable	1	Input	Enables state machine functionality.
first_label_value	1	Input	Indicates if the first label value has been generated.
generate_label	1	Input	Used to indicate that a new label should be generated.
idle	1	Output	Indicates if the label generator is active. A high value indicates that no operations are taking place and a low value indicates otherwise.
label_counter_ctrl	3	Output	Used to manipulate the counter for generating label values.
max_label_value	1	Input	Indicates if the maximum label value has been generated.
no_more_labels	1	Output	Used to indicate if there are any more labels can be generated. A high value indicates that no more labels are available and a low value indicates otherwise.
reset	1	Input	Resets the state machine.

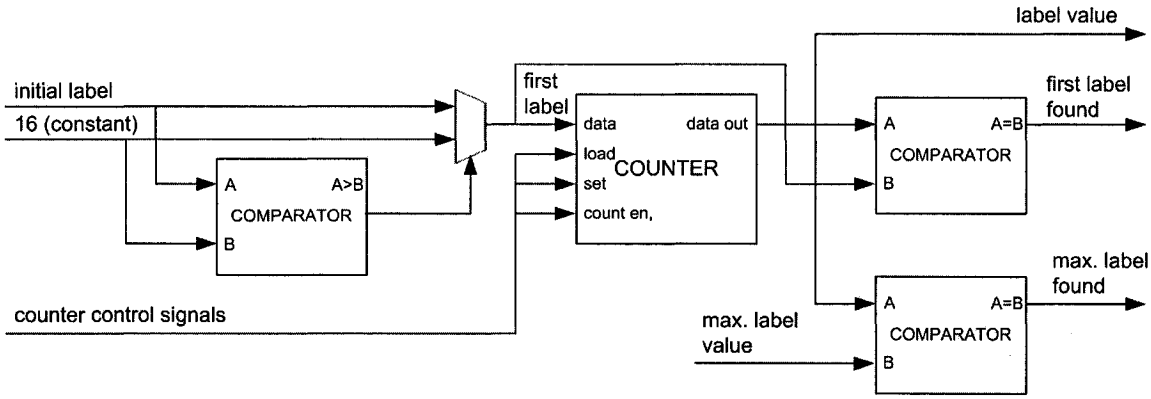


Figure 4-10. Data Path for Label Generator

label that can be used. An initial label is provided as an external value and is set by the user. Such an option is provided in case it is more convenient for a router to have labels in a certain range for administrative purposes. Should no desire exist, random number generators are typically available in software to provide any number.

The data path for the label generator is described in Figure 4-10. It is composed of a counter, for generating unique labels and comparators serving different purposes. The

comparator and multiplexer feeding the data signal of the counter serve to set the first label that can be used. An initial label is provided as an external value and is set by the user. Such an option is provided in case it is more convenient for a router to have labels in a certain range for administrative purposes. Should no desire exist, random number generators are typically available in software to provide any number.

When a label is generated it is compared with the first label and the maximum value. The results of those comparisons are made available to the control unit so it can be known if further labels can be generated.

4.2.5 Label Database

The label database is used to store and retrieve labels. The only two functions available with the label database are reading and writing data so one can think of the label database as an abstracted RAM (or storage mechanism) for labels. These labels are used in the MPLS data plane (shown in
Figure 2-5) to perform label switching. All that is required from the label database is a means of storing labels and identifying them with the corresponding LSP. Consequently the address used to store a label will be the same address used to store LSP data.

Each LSP is associated with two labels: the label for incoming data and outgoing data. An ingress LER does not have an incoming label and an egress LER has no outgoing label for LSPs. Those values are presumed to be zero.

Control Unit

The control unit for the label database is illustrated in Figure 4-11. The state machine only has functional states for reading and writing besides idle and reset states. The states are described in Table 4-9 and all of them require one cycle. Table 4-10 summarizes the signals for the label database control unit. All signals are one bit wide and the only outputs are for RAM manipulation and an activity indicator. Inputs are used to enable or

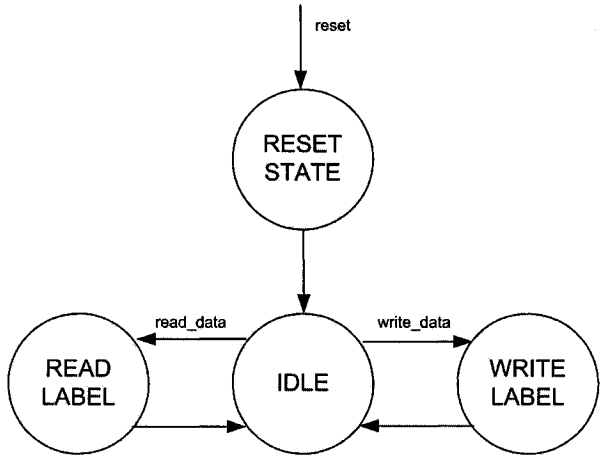


Figure 4-11. Control Unit for Label Database

Table 4-9. States for Label Database Control Unit

Name	Number of Cycles	Description
IDLE	1	The default state where no functions are being performed.
READ LABEL	1	All actions to read data from the database are done.
RESET	1	Used to reset the state machine.
WRITE LABEL	1	All actions to write data to the database are done.

Table 4-10. Signals for Label Database Control Unit

Name	Width (bits)	I/O	Description
clk	1	Input	Clock signal.
enable	1	Input	Enables state machine functionality.
idle	1	Output	Indicates if the label database is active. A high value indicates that no operations are taking place and a low value indicates otherwise.
read_data	1	Input	Used to indicate that data should be read from the label database.
read_data_mem	1	Output	Used to manipulate the RAM in the data path for reading data.
reset	1	Input	Resets the state machine.
write_data	1	Input	Used to indicate that data should be written to the label database.
write_data_mem	1	Output	Used to manipulate the RAM in the data path for writing data.

reset the label database in addition to reading or writing data. Output signals are used to read and write data only.

Data Path

Figure 4-13 illustrates the data path for the label database. It is only composed two RAM modules, for incoming and outgoing labels of a given LSP. The read and write addresses correspond to the addresses used to store LSP. LSPs are addressed in a sequential manner as users enter the information. Therefore, users can easily know which LSP to retrieve data for by remembering the sequence used to store LSP data. The RAM is controlled by the control unit while the labels and addresses are provided by external values.

4.2.6 LSP Module

This module is used to store all information about the LSPs that will be supported in the network. Several pieces of data must be maintained for every LSP including all data in the flow related objects, refresh rate and ports for upstream and downstream data flow. The LSP identifiers (source IP address, destination IP address, tunnel ID, LSP ID and extended tunnel ID) are stored as well.

The module is used to read, write and search for LSP data. LSP identifiers are searched although the infrastructure exists to search any value saved with the LSP. Data can be read following the results of a search or with an arbitrarily specified internal LSP address. LSP data is written incrementally and there is no need for an external address.

Control Unit

Figure 4-12 illustrates a high level control unit for the LSP storage module. When the control unit is idle, an operation to read, write, or search LSP data is performed. Data can be read based on internal addressing mechanisms for an external address. That option was made available so there was absolute flexibility in retrieving any stored at any time. No such flexibility is applicable to writing or search data.

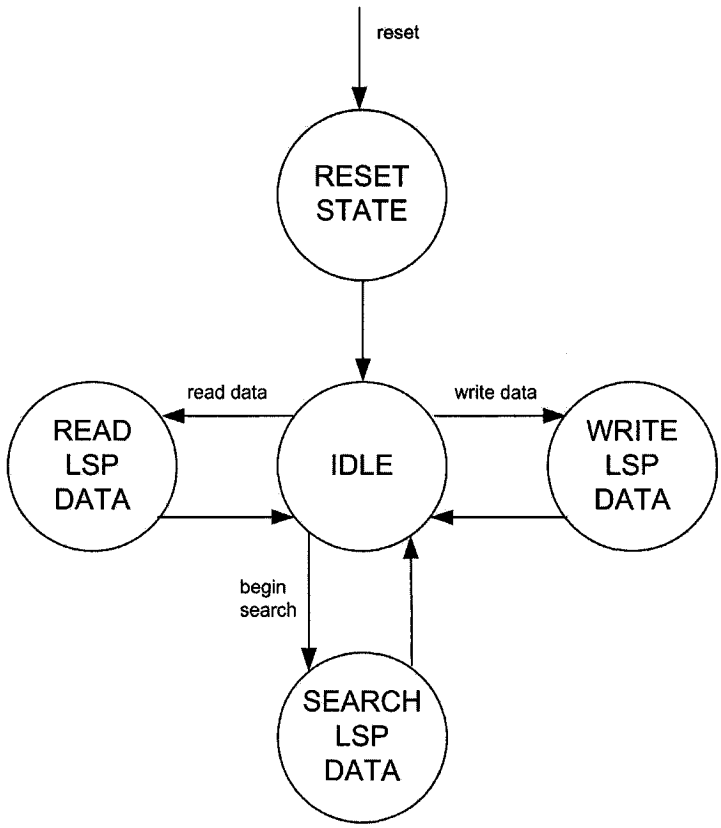


Figure 4-12. Control Unit for LSP Module

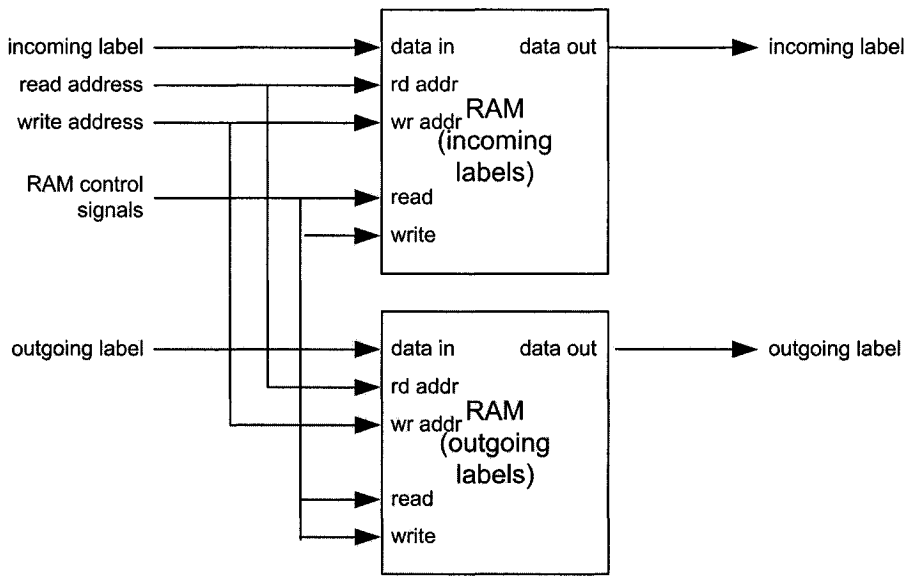


Figure 4-13. Data Path for Label Database

The high level states for the control unit are described in Table 4-11. The search LSP data state is the only one where a variable number of cycles are required to perform the

Table 4-11. States for LSP Module Control Unit

Name	Number of Cycles	Description
IDLE	1	The default state where no functions are being performed.
READ LSP DATA	2	All actions to read data from the LSP module are done. Data can be read based on an external address or an internal counter.
RESET	1	Used to reset the state machine.
SEARCH LSP DATA	Variable	Used to search the LSP module for a specific LSP based on the source address, destination address, tunnel ID, LSP ID, and extended tunnel ID.
WRITE LSP DATA	2	All actions to write data to the LSP module are done.

operation because there are always an unknown number of LSPs before the search begins. All other states require two cycles or less to complete. The control unit signals are summarized in Table 4-12.

Data Path

A high level illustration of the LSP storage data path is shown in Figure 4-14. There is one RAM module for every data component that must be stored per LSP. The five data components that uniquely identify an LSP have their values stored in registers before they are stored in RAM so they can be used in a search while not being read from RAM before hand. The extended tunnel ID can either be the source address or zero, so multiplexers are used before the appropriate register to choose between the desired choice (when specified from the user), or when to use the value from the packet parser. Counters are used to increment the read and write addresses but the search module may also manipulate the read address while performing a search. Comparators and other signals are used to generate search results but they are omitted from Figure 4-14 to simplify the diagram. The implementation of the search module and the signals used are explained in 4.1.2.

No state information is saved as was shown for the hardware signaling framework described in section 0 because the functionality provided in the processor is not state dependent. However should additional features be provided, the state of an LSP can

Table 4-12. Signals for LSP Module Control Unit

Name	Width (bits)	I/O	Description
begin_search	1	Output	Used to begin an LSP search through the search module.
clk	1	Input	Clock signal.
enable	1	Input	Enables state machine functionality.
enable_search	1	Output	Enable the search module.
idle	1	Output	Indicates if the LSP module is performing any operations.
item_found	1	Input	Indicates if an LSP was found during the search.
rdaddress_cntr_mux_ctrl	1	Output	Used to indicate which module is manipulating the read address counter (control unit or the search module).
rdaddress_mux_ctrl	1	Output	Used to indicate the source of the read address (an external value or an internal counter).
read_data_address	1	Input	Indicates that data should be read based on the external LSP address.
read_data_counter	1	Input	Used to read data using an internal counter as the LSP address.
read_data_mem	1	Output	Used to read data from the data path.
reset	1	Input	Resets the state machine.
reset_search	1	Output	Resets the search module.
search_data	1	Input	Used to indicate that an LSP search should begin.
search_done	1	Input	Feedback from the search module indicating that the search is complete.
wraddress_cntr_ctrl	2	Output	Used to manipulate the counter for the write address.
write_data	1	Input	Indicates that LSP data should be saved.
write_data_mem	1	Output	Used to write data to the data path

easily be saved with an additional RAM component in this data path and high level logic to update it accordingly.

4.2.7 Packet Parser

The packet parser is one of the more complex units in the MPLS processor. Its only two functions are to parse a packet and read parsed data. While the latter task is relatively straight forward the former is rather complex and requires most of the processing and architecture described in the following subsections.

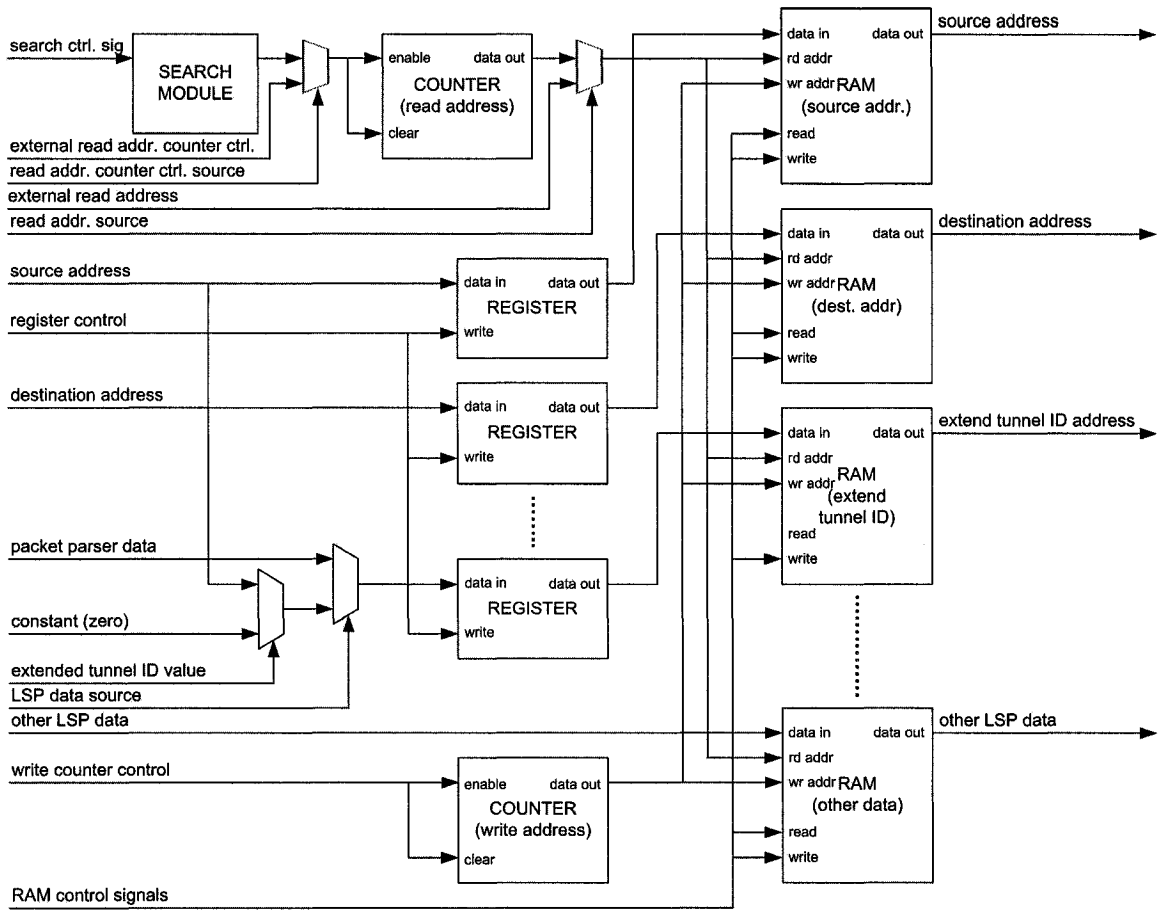


Figure 4-14. LSP Module Data Path

In addition to parsing the packet, this module ensures that all data is accessible in an addressable and consistent manner. A variety of conditions are checked as the packet is parsed. Those conditions include ensuring that all object lengths match the packet length, specifying missing objects and checking the version number in the packet header.

Control Unit

Figure 4-15 illustrates the control unit for the packet parser. In its idle state, the control unit can be prompted to read packet data or begin the process of parsing a packet. A packet is parsed from the beginning (the packet header) to the end (the collection of objects). Therefore the first task is to read all header data and save it to the data path. This task also includes a search to ensure that the packet type is supported by the architecture. Then the total amount of processed data is updated and the packet is

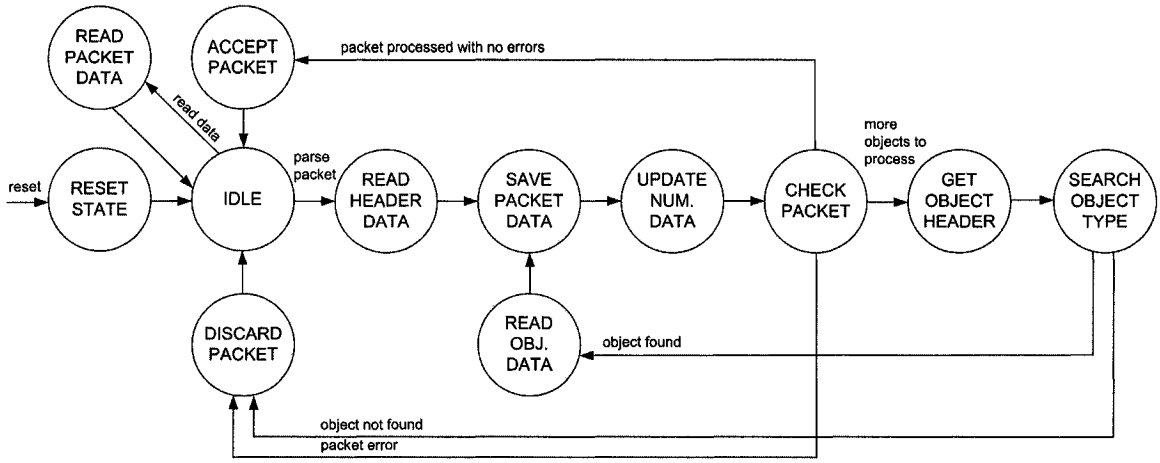


Figure 4-15. Control Unit for Packet Parser

checked. If there are any errors in the packet header, the entire packet is immediately discarded and the control unit becomes idle, otherwise the object header is read and saved. A search for the object type, based on the c-type and class number, is performed to ensure that is it supported by the processor. The packet is discarded if no match is found otherwise object data is read and saved. After data is processed the packet is checked and the process continues until the entire packet is parsed. If no errors or inconsistencies are found while parsing the packet it is accepted with all data stored in the data path.

Table 4-13 summarizes the states of the packet parser control unit. There are several states requiring multiple cycles to perform their tasks. Reading packet header data requires more cycles than any other task because two thirty-two bit words must be read with many components of various widths. All those components must be separated and compared with values supported by the implementation. Reading an object header (while parsing a packet) and reading packet data (while not parsing) are also among the most time consuming tasks because time must be allowed for data to propagate before the next task can be performed. It should also be noted that searching the object type requires an indeterminate number of cycles because it is not known if the desired object exists. The same scenario exists for reading header data because the packet type must be searched as well. All other states require one or two cycles.

Table 4-14 and Table 4-15 describe the signals used in the packet parser control unit. All inputs are one bit wide and are mainly feedback from the data path, indicating the validity

Table 4-13. States for Packet Parser Control Unit

Name	Number of Cycles	Description
ACCEPT PACKET	1	State where it is indicated that the packet has been accepted.
CHECK PACKET	1	State where it is determined if there are more objects to process and if there has been any parsing errors.
DISCARD PACKET	1	Indicate that the packet should be discarded.
GET OBJECT HEADER	4	State where the amount of object data is reset, the object header is read and saved.
IDLE	1	The default state where no functions are being performed.
READ HEADER DATA	Variable	Read all data for the header (two thirty two bit words) and separate all components.
RESET	1	Used to reset the state machine.
READ OBJECT DATA	2	State where object data is read from a stack external to the packet parser.
READ PACKET DATA	4	State where packet data is read from a specified data component in a specified object.
SAVE PACKET DATA	1	Packet data is saved to the location reserved for the data type and object type.
SEARCH OBJECT TYPE	Variable	State where the search module operates for the object with the specified c type and class number. The control unit waits until the search is complete.
UPDATE NUMBER OF DATA	1	State where the total amount of data processed for the object and the packet is updated.

of the packet at the time. All output signals are used to manipulate various data path components except for the idle signal and the stack control.

Data Path

Figure 4-16 illustrates data path components for packet parser storage. All packet data is kept in the storage unit that stores data based on its object and data type. All objects described in section 2.2.4 store data in thirty-two bit words. Each object is separately enumerated and within each object, each word is enumerated with consecutive numbers starting at zero. In that manner, all possible values in a packet can be distinctly referenced. The data type and object type can be specified externally by a user reading packet data or internally while parsing a packet. The internal data type is provided by a counter and incremented when necessary because object data is presumed to appear in the

Table 4-14. Input Signals for Packet Parser Control Unit

Name	Width (bits)	Description
clk	1	Clock signal.
enable	1	Enables state machine functionality.
finished_proc_message	1	Indicates that the amount of data processed from all objects matches the specified packet length.
finished_proc_object	1	Indicates that the object has been processed.
object_item_found	1	Indicates that an object has been found in the object search.
object_search_done	1	Used to indicate that the object search is complete.
read_data	1	Used to indicate that a data element from the packet should be read.
packet_contents_valid	1	Indicates that the contents of the packet are valid given its type.
packet_type_item_found	1	Used to specify that the packet type in the header is supported.
packet_type_search_done	1	Specifies that the search for the packet type is complete.
parse_packet	1	Used to indicate that the packet should be parsed.
reset	1	Resets the state machine.
stack_idle	1	Indicates that the stack containing the packet to be parsed is idle.
valid_version	1	Indicates that the version number in the packet header is valid.

same sequence. The object type is provided by a search module, because the c-type and class number must be supported by the processor.

Packet data can be stored in the storage module or a series of registers through a demultiplexer. Registers hold all the components of the packet header so there are simultaneously available.

The storage component in Figure 4-16 is described below in Figure 4-17. A RAM component is used to store all packet data. The object type is an absolute address used to access the memory reserved for the object while the data type acts as an offset. In the manner the object type and data type can be added to get the total address for the data being read or written.

Table 4-15. Output Signals for Packet Parser Control Unit

Name	Width (bits)	Description
begin_search_object_type	1	Used to begin a search for the object type.
begin_search_packet_type	1	Used to begin a search for the packet type.
data_type	2	The type of packet data being parsed.
discard_packet	1	Used to indicate that the packet should be discarded.
enable_object_search	1	Enables the object search module.
enable_packet_type_search	1	Enables the packet type search module.
msg_len_cntr_ctrl	2	Used to control the counter for keeping track of the amount of packet data processed.
obj_data_type_cntr_ctrl	2	Used to control the counter describing the object data being processed.
obj_len_cntr_ctrl	2	Used to control the counter for keeping track of the amount of object data processed.
object_info_src	1	Specifies the source of control (the user for reading data or the parser or other operations).
parser_idle	1	Indicates if the parser is performing any actions.
read_object_data	1	Used to read object data from the data path.
reset_object_search	1	Used to reset the search module for the objects.
reset_packet_type_search	1	Used to reset the search module for the packet type.
stack_control	3	Used to manipulate the stack containing the packet to parse.
write_header_ttl_data	1	Used to write the part of the packet header containing the TTL.
write_header_version_data	1	Used to write the part of the packet header containing the version header.
write_object_data	1	Used to save object data in the data path.
write_object_header	1	Used to save the object header in the data path.

Figure 4-18 illustrates the mechanisms used to determine if an object or the entire packet has been processed. A counter is incremented to indicate how much data has been processed. That counter value is multiplied by four because data is processed in thirty-two bit words and there are four bytes in each word. That value is then subtracted from length value in the header and compared against the value zero. If the difference between the length of the object or packet and the amount of data processed is zero then the signal goes high, indicating that processing has finished.

Figure 4-19 describes the mechanisms used to validate the contents of a packet. Each packet has a mandatory set of objects it must contain as described in Table 2-1. There are

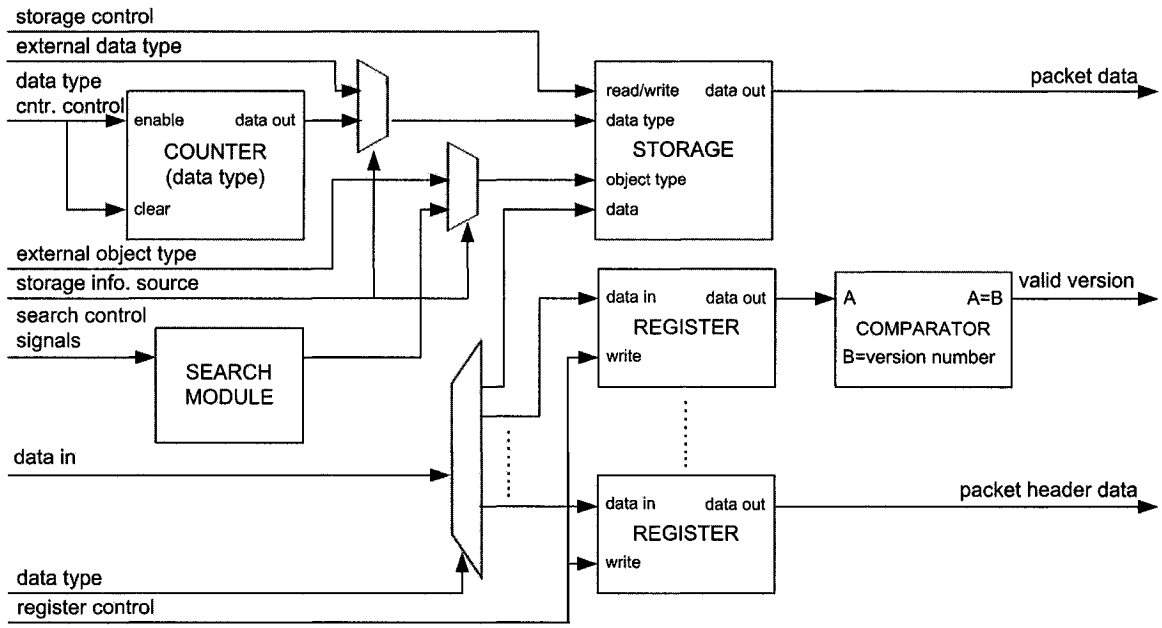


Figure 4-16. Packet Parser Data Path Components for Storing Data

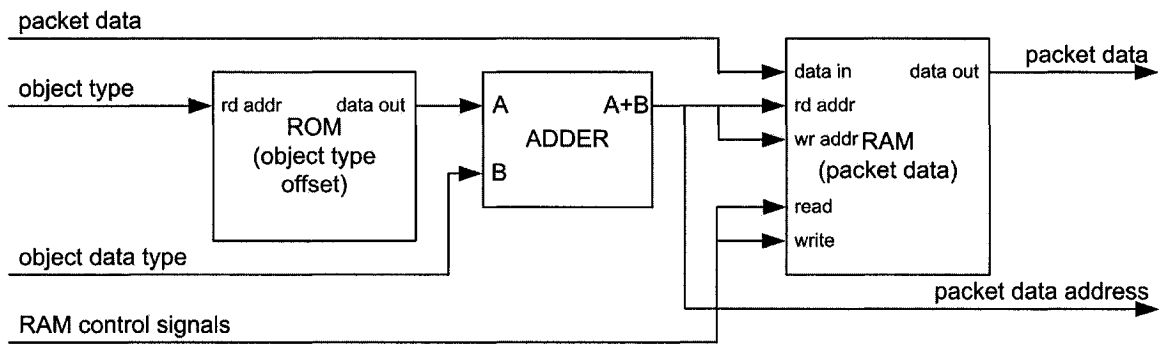


Figure 4-17. Storage Component for Packet Parser Data Path

eleven objects supported by this processor. Those objects are described in an eleven bit word where each bit represents an object. One ROM component is used to store the packet contents in a bitmap and another module contains the bit mask for every object supported by the architecture. In this manner, additional objects can be supported by adding values to both ROM components, and expanding other memory modules described earlier in this section. The packet type is searched to retrieve the correct bitmap that must be present after parsing the entire packet. As objects are read and parsed, the object value is loaded and an OR gate is used to update the previous value. After all objects have been processed, a comparator generates a value indicating if all the correct objects have been processed.

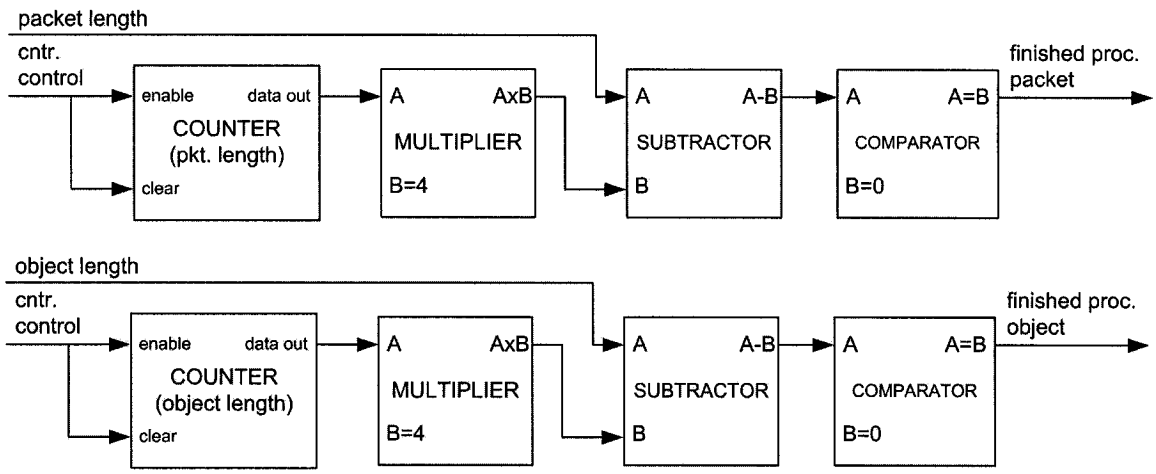


Figure 4-18. Packet Parser Data Path Components to Check Parsing Completion

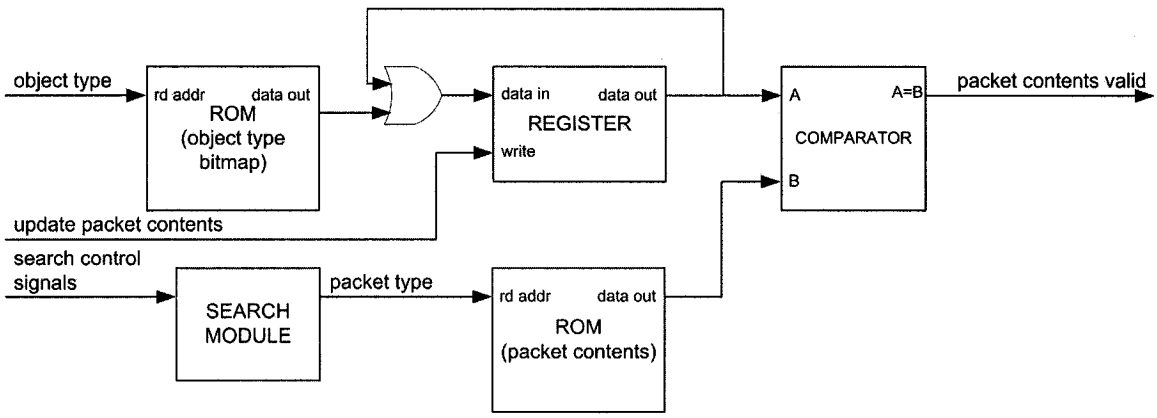


Figure 4-19. Packet Parser Data Path Components for Packet Verification

4.2.8 Packet Generator

The packet generator is used to generate a packet of any type supported by the processor. All data is written to the outgoing data stack shown in Figure 3-3. Data can be provided through the packet parser from a previously processed packet or through other modules.

Control Unit

The control unit has been separated into two components. One component is a packet generator, used to perform the high level tasks of generating a packet. The other component is an object generator, performing the same tasks at the object level. Both modules are very similar in their composition but the object generator is more complex

since there are more potential objects to generate than packets, and there are multiple sources of data for objects.

The packet generator is shown in Figure 4-20. When it is idle, any one of six packets can be generated and the appropriate high level state is occupied to generate its data. Once all the data has been written the packet header is written and packet generation is complete. The high level states coordinate the generation of object by manipulating the object generator to create objects in a specified sequence. While an object is being generated the packet generator waits for completion. Consequently the time spent in each data generating state depends on the number of objects generated. Writing the packet header requires two cycles. Packet parser signals are described in Table 4-16.

A simplified version of the object generator is illustrated in Figure 4-21. The number of possible objects that can be created, taking the different sources into account, makes it unfeasible to illustrate them all in the diagram. When the object generator is idle, it can be prompted to create an object by writing the data to the outgoing data stack. After all data has been written the object header is written to the stack and the object generator becomes idle. The number of cycles required to generate the objects depends on the source. Data is read from the packet parser but presumed available when it is not used. It requires three cycles to read and write data from the parser to the stack and two cycles without using the stack. The signals for the object generator are shown in Table 4-17 and Table 4-18.

Besides the enable and reset functions, input signals indicate when an object should be generated. The data for an object may come from the packet parser or from other modules in the processor. Consequently there is a signal to indicate the data source for every object. The source is either the packet parser, where data must be explicit read, or an internal module where it is presumed that data is already available. Output signals manipulate the object generator data path and packet parser.

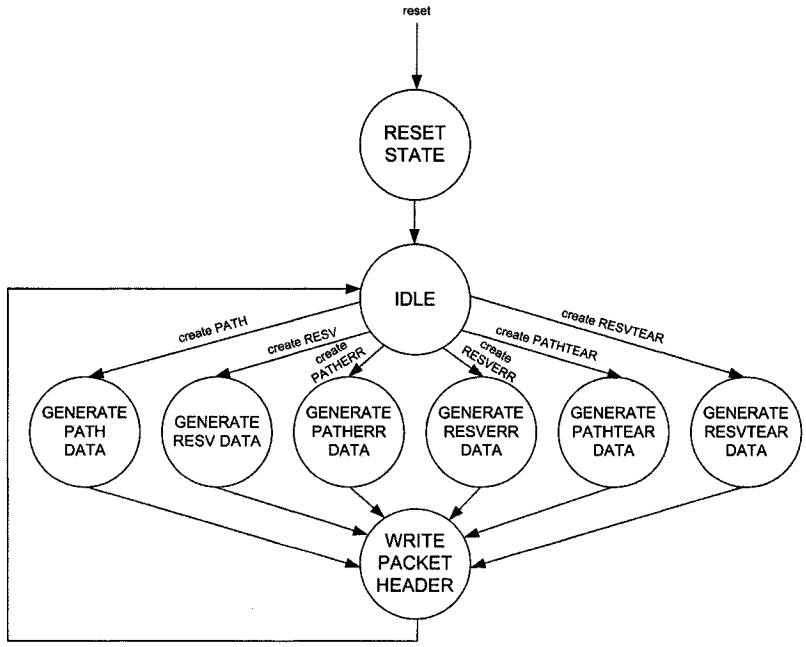


Figure 4-20. State Machine for the Packet Generator

Table 4-16. Signals for the Packet Generator

Name	Width (bits)	I/O	Description
clk	1	Input	Clock signal.
enable	1	Input	Enables state machine functionality.
external_stack_src	5	Output	Describes the type of data being written to the stack.
generate_object	1	Output	Used to begin generating an object through the object generator.
generate_packet	1	Input	Used to begin generating a packet.
idle	1	Output	Indicates if the state machine is idle.
module_in_control	1	Output	Describes which module is currently writing data to the output stack. A high value indicates that it is the object generator and a low value indicates that it is the packet generator.
object_generator_enable	1	Output	Enables the object generator.
object_generator_idle	1	Input	Indicates if the object generator is idle.
object_generator_reset	1	Output	Resets the object generator.
object_type	4	Output	Describes the object to be generated.
packet_type	4	Input	Describes the packet to be generated.
packetLenCtrl	2	Output	Controls the counter to increase the length of the packet when appropriate.
reset	1	Input	Resets the state machine.
stack_control	3	Output	Signals to manipulate the output stack.

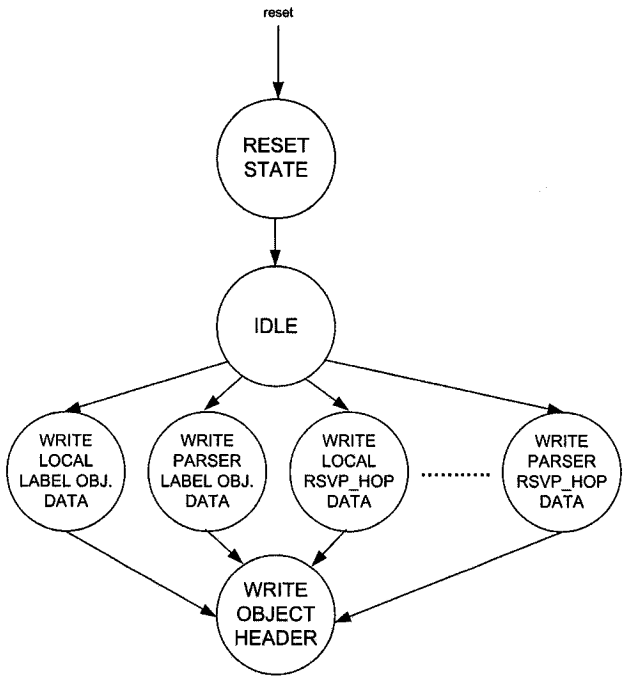


Figure 4-21. State Machine for the Object Generator

Data Path

The data path shown in Figure 4-22 shows a multiplexer to coordinate all possible packet data that can be written to the outgoing data stack. There are too many distinct thirty-two bit formats for them to be shown effectively in this diagram. They are all described in section 2.2.4.

ROM modules are used to store the message type, class number and c type required for writing different types of data to the stack. Those values are generated by supplying the packet type and object type. The object length is determined by incrementing a counter and multiplying its value by four. When the correct object length has been determined, it is added to a running total of all object lengths so it can be used for the packet length.

4.2.9 Stack

Two stacks are used to buffer data entering or leaving the architecture. A stack is a last-in-first-out (LIFO) data structure where the last piece of data saved is the first piece of data to be retrieved. Saving data to the stack is called a push operation and removing

Table 4-17. Input Signals for the Object Generator

Name	Width (bits)	Description
clk	1	Clock signal.
enable	1	Enables state machine functionality.
generate_object	1	Used to begin generating an object.
idle_output_stack	1	Indicates if the stack is idle.
idle_packet_parser	1	Indicates if the packet parser is idle.
object_type	4	Describes the object to be generated.
reset	1	Resets the state machine.
source_pkt_prsr_error_spec	1	Indicates if the source of the ERROR_SPEC data is the parser.
source_pkt_prsr_filter_spec	1	Indicates if the source of the FILTER_SPEC data is the parser.
source_pkt_prsr_flowspec	1	Indicates if the source of the FLOWSPEC data is the parser.
source_pkt_prsr_label	1	Indicates if the source of the LABEL data is the parser.
source_pkt_prsr_label_rq	1	Indicates if the source of the LABEL REQUEST data is the parser.
source_pkt_prsr_rsvp_hop	1	Indicates if the source of the RSVP_HOP data is the parser.
source_pkt_prsr_sndr_tmp	1	Indicates if the source of the SENDER_TEMPLATE data is the parser.
source_pkt_prsr_sndr_tspec	1	Indicates if the source of the SENDER_TSPEC data is the parser.
source_pkt_prsr_session	1	Indicates if the source of the SESSION data is the parser.
source_pkt_prsr_style	1	Indicates if the source of the STYLE data is the parser.
source_pkt_prsr_time_values	1	Indicates if the source of the TIME VALUES data is the parser.

data from a stack is a pop operation. Conventional RAM alone or a LIFO structure could have been used for buffering data but would have not been allowed for operations to be performed from the software level as quickly as possible. RAM alone would not have been possible because it is necessary to keep track of the amount of data currently in the stack. Subsequent sections will illustrate that the data component of the stack contains RAM as a primary storage mechanism.

Table 4-18. Output Signals for the Object Generator

Name	Width (bits)	Description
external_stack_src	5	Describes the type of data being written to the stack.
idle	1	Indicates if the state machine is idle.
object_data_type_parser	4	Describes the object data being to read from the parser.
object_type_parser	4	Describe the object type to read from the parser.
objLenCtrCtrl	2	Controls the counter to increase the length of the object when appropriate.
packetLenCtrCtrl	2	Controls the counter to increase the length of the packet when appropriate.
read_packet_parser_data	1	Indicates that data should be read from the packet parser.
stack_control	3	Signals to manipulate the output stack.

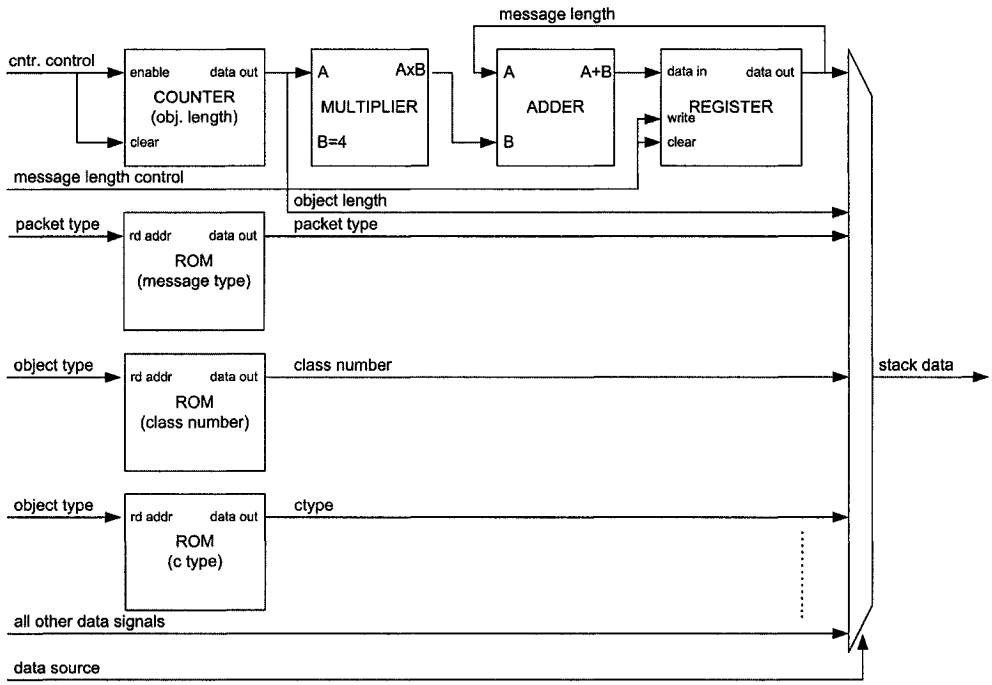


Figure 4-22. Packet Generator Data Path

An RSVP-TE packet is composed of a header and numerous objects. The header is at the beginning of the packet while the objects and their data are at the end. When composing a packet, the object data must be available first before the header can be generated because the packet message length must account for the number and composition of the packet objects. So the end of the packet is available before the beginning, but a full

packet is processed from beginning to end. Consequently, it is more efficient to enter data in reverse order, as it becomes available, but allow it to be processed easily from the beginning. A stack makes this scenario possible in an efficient manner because objects can be pushed immediately. But the first data that is popped is the packet header.

Control Unit

Figure 4-23 illustrates a high level description of the control unit for the stack. The states are summarized in Table 4-19. It can be reset asynchronously; thereby clearing all data in the stack and the control unit would then become idle. Only two high level functions can be performed on a stack (push and pop) so there are states to perform each of those functions when the stack is idle. However each function requires a different number of cycles. Reading stack data requires three cycles to decrement the number of data entries, read data and wait for data to propagate. Only two cycles are required to write data because it is not necessary to wait for data to be saved.

The stack is not implemented in a circular manner. Consequently no data can be pushed to the stack when it is full and no data can be popped when it is empty. Relatively few signals are required to manipulate the stack to perform push and pop operations. Those signals are summarized in Table 4-20. Inputs describe which operation must be performed and the current state of the stack (empty, full or neither). Outputs are used to coordinate the data path by manipulating a counter (for the number of elements) and memory.

Data Path

Figure 4-24 shows the data path for the stack. Packet data is stored in RAM and the counter is used for addressing purposes, thereby providing the total number of data elements in the stack at all times. Therefore, resetting the stack can be performed by resetting the counter alone. All data immediately following the reset will be successfully ignored. Comparators are used to indicate if the stack is empty or full. The counter value is compared to zero to see if the stack is empty, and the total number of RAM addresses

Table 4-19. States for Stack Control Unit

Name	Number of Cycles	Description
IDLE	1	The default state where no functions are being performed.
READ STACK DATA	3	All actions to read data from the stack are performed.
RESET	1	Used to reset the state machine.
WRITE STACK DATA	2	All actions to write data to the stack are performed.

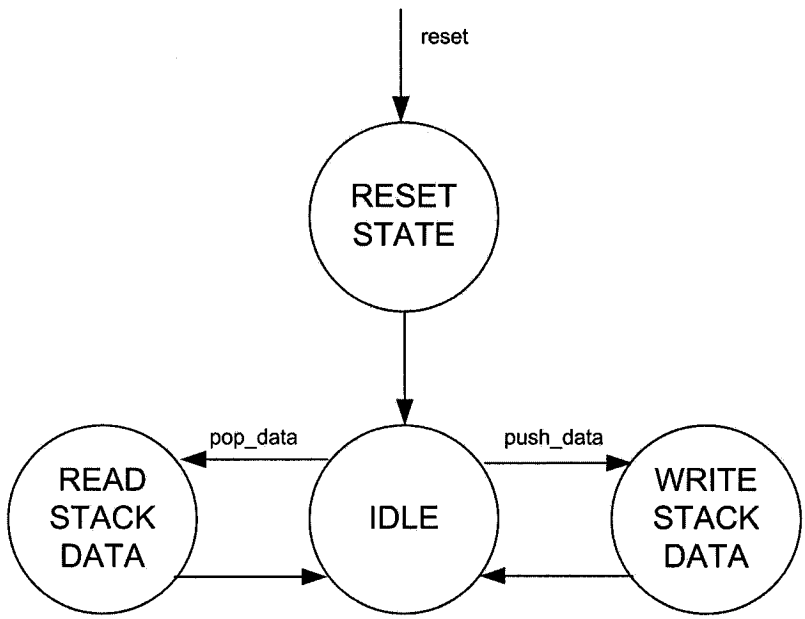


Figure 4-23. Stack Control Unit

to see if it is full. In this manner, the capacity of the stack can be easily increased by adding more RAM and updating one comparator. The control unit would not have to be modified in any way. All data pushed and popped from the stack is thirty two bits wide.

4.2.10 Traffic Engineering Database

The traffic engineering database (TE DB) is used to store all information pertaining to the links of the router. Information stored for each link is the following:

Table 4-20. Signals for Stack Control Unit

Name	Width (bits)	I/O	Description
clk	1	Input	Clock signal.
enable	1	Input	Enables state machine functionality.
pop_data	1	Input	Used to indicate that a pop operation should be performed with the stack. A high value indicates that a pop should occur and a low value indicates otherwise.
push_data	1	Input	Used to indicate that a push operation should be performed with the stack. A high value indicates that a push should occur and a low value indicates otherwise.
reset	1	Input	Resets the state machine.
stack_counter_ctrl	3	Output	Used to manipulate the counter in the stack data path.
stack_empty	1	Input	Indicates if there is no data in the stack. A high value indicates if the stack is empty and a low value indicates otherwise.
stack_full	1	Input	Indicates if the stack is full. A high value indicates if the stack is full and a low value indicates otherwise.
stack_ram_ctrl	2	Output	Used to manipulate the RAM in the stack data path.

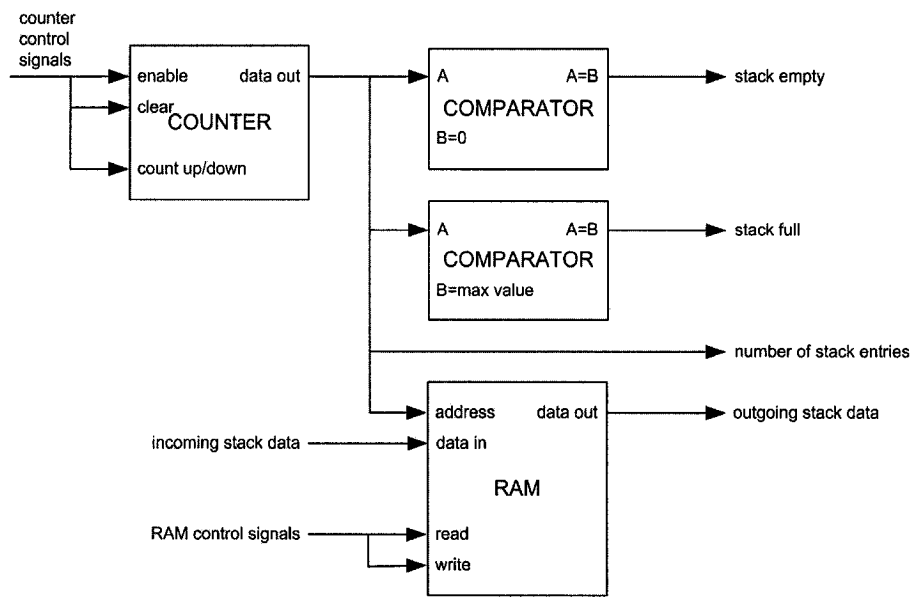


Figure 4-24. Stack Data Path

- Total bandwidth.
- Local ID number.
- Local IP address.
- IP address at the connecting router.
- Currently reserved bandwidth.
- Percentage of total bandwidth that can be used for LSPs.

Four main functions have been implemented for the TE DB. Those functions are to read data, write data, reserve bandwidth and restore bandwidth. The desired link must be specified when those actions are performed. Searches have been enabled for the local ID number and the IP address at the connecting router. When the function of reading data or reserving bandwidth is chosen, those criteria will be used to find the desired link for which to perform the desired operation.

Control Unit

The control unit for the TE DB is illustrated in Figure 4-25 with the states described in Table 4-21. When the TE DB is idle it can immediately write data or begin searching for the desired link. If no link is found the control unit immediately becomes idle, otherwise the desired operation is performed. Writing data, reserving bandwidth and restoring bandwidth require two cycles to perform their operations reading data require one cycle. Writing data involves updating the total number of links and the bandwidth must be checked before a reservation can occur.

Table 4-22 summarizes the signals used in the control unit for the TE DB. Inputs are the desired functions and feedback from the data path. Outputs manipulate the data path and search module. Input and output signal indicate a relatively simple interface to the control unit since most of the functionality and logic is implemented in the data path.

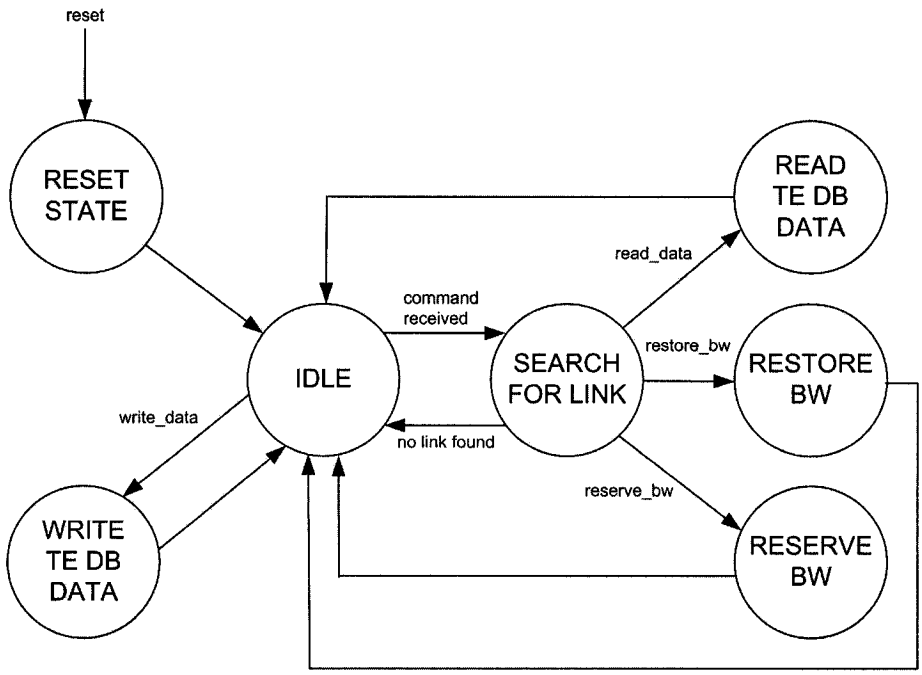


Figure 4-25. Control Unit for Traffic Engineering Database

Table 4-21. States for the Traffic Engineering Database Control Unit

Name	Number of Cycles	Description
IDLE	1	The default state where no functions are performed.
READ TE DB DATA	1	State where all data for a given link is read from the traffic engineering database.
RESERVE BW	2	Specific bandwidth is reserved for a link in this state. A check is performed to ensure that there is sufficient bandwidth for the operation.
RESET	1	Used to reset the state machine.
RESTORE BW	2	Bandwidth is restored (or released) that was previously allocated to an LSP.
SEARCH FOR LINK	1	Wait state while the search module performs the search.
WRITE TE DB DATA	2	State where all data for a given link is written to the traffic engineering database.

Data Path

Figure 4-26 provides a simplified version of the data path for the traffic engineering database. Output signals indicating the status of the search are omitted to simplify the diagram. There are RAM modules to store the six types of information described at the beginning of section 4.2.10. Counters are used for the read and write addressed of the

Table 4-22. Signals for the Traffic Engineering Database Control Unit

Name	Width (bits)	I/O	Description
begin_search	1	Output	Used to prompt the search module to find the desired link.
clk	1	Input	Clock signal.
enable	1	Input	Enables state machine functionality.
enable_search	1	Output	Enables the search module in the data path.
idle	1	Output	Indicates if the traffic engineering database is active.
insufficient_bandwidth	1	Input	Indicates if there is sufficient bandwidth to reserve the desired amount.
item_found	1	Input	Indicates if the desired link was found.
read_data	1	Input	Used to begin process of reading data from the traffic engineering database.
read_data_mem	1	Output	Used to read data from the data path.
reserve_bw_db	1	Output	Used to reserve bandwidth in the data path.
reserve_bw_user	1	Input	Used to begin process of reserving bandwidth in the traffic engineering database.
reset	1	Input	Resets the state machine.
reset_search	1	Output	Resets the search module in the data path.
restore_bw_user	1	Input	Used to begin the process of restoring bandwidth to the traffic engineering database.
search_done	1	Input	Indicates when the search module is idle.
wraddress_cntr_ctrl	2	Output	Used to manipulate the number of links stored in the traffic engineering database.
write_data	1	Input	Used to write data to the data path.
write_data_mem	1	Output	Used to write data link to the data path.

RAM modules however the read address counter is controlled by the search module. Besides RAM and counters, other components are used to ensure that operations are performed with valid values.

The data for the RAM module storing the available percentage of data is fed by the minimum of 100 and the specified number, in case the user provides an errant value. The minimum is provided by a comparator connected to a multiplexer, where the lesser value is used as an index for the correct number. To reserve bandwidth an adder is connected to the input of the appropriate RAM module. The desired bandwidth is one value in the adder and the current amount of reserved bandwidth is the other. When a reservation takes place both numbers are added to obtain the new amount of bandwidth that has been reserved for the given link.

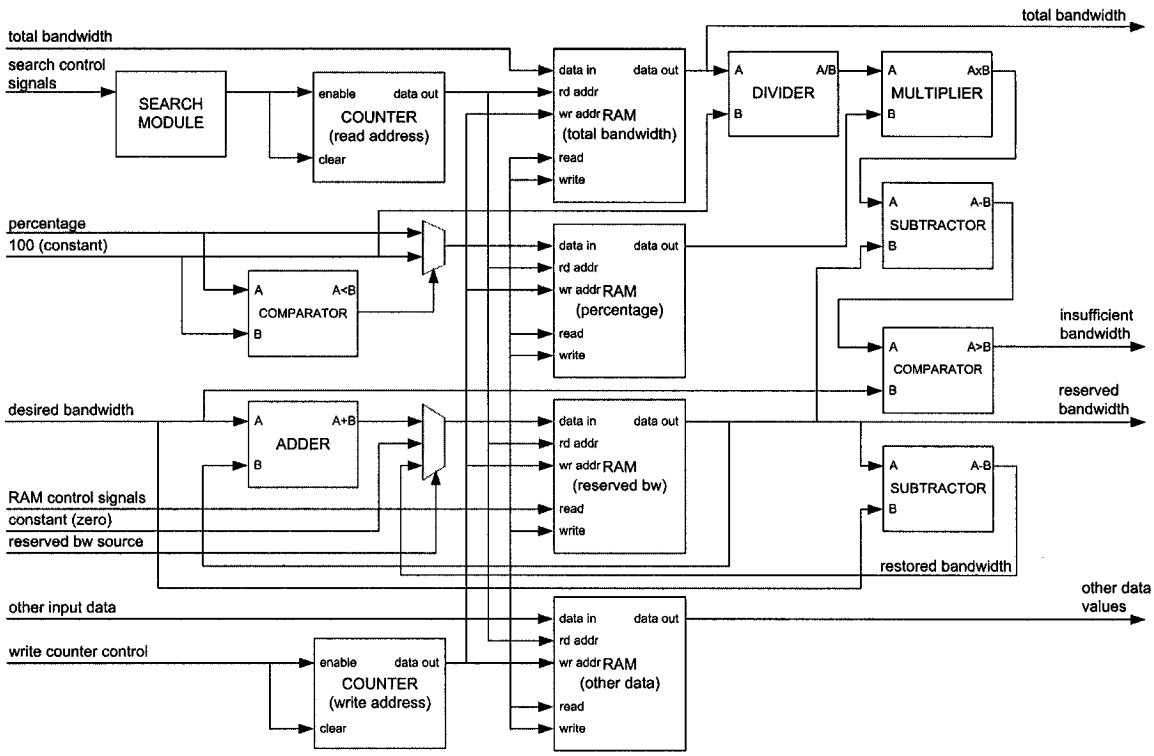


Figure 4-26. Data Path for Traffic Engineering Database

A divider, multiplier, subtractor and comparator are all used to determine if there is sufficient bandwidth for reservation. The total bandwidth is multiplied by the allocated percentage to determine what can be used to reserve resources on the link. The operation is (bandwidth/100) * percentage and is performed with the divider and multiplier shown. The reserved bandwidth is then subtracted from the result and compared to the desired bandwidth. If the desired bandwidth is greater than what is available then a signal is asserted to indicate that no reservation can take place. To restore bandwidth to the TE DB, the desired bandwidth is subtracted from the reserved bandwidth and rewritten to the reserved bandwidth RAM component.

Chapter 5

5 Results and Analysis

An analysis of the MPLS processor is provided in this section with respect to the resources that are used and the operations it performs. The first section describes the resources used by the architecture and general conclusions are drawn based on the functionality and resources used on a component by component basis. A testing scenario is then presented illustrating a VPN using LSPs among various sites for private communications. Several simulations are then provided illustrating how different components operate in sequence to perform several tasks. The processor's overall performance is discussed to conclude the chapter.

5.1 Resource Usage

This section describes the resource usage of the processor as implemented on an Altera Stratix EP1S40F780 device [6] with great emphasis on logic cells, memory and fan-out. A logic cell (also known as a logic element) is a generic term for a basic building block for Altera devices. On Stratix devices, a logic cell consists of a four-input LUTs, a programmable register, and a carry chain. A carry chain is a dedicated architectural feature providing a high performance carry forward function between logic cells. Fan-out refers to output signals that are fed by the output equations of a logic cell. The total number of resources used is illustrated in Table 5-1 and a more detailed breakdown of the resource usage is given in Table 5-2. The entry for 'miscellaneous items' describes the parts used to integrate the main components of architecture (a small set of multiplexers and registers). Every component uses logic cells however only six of a possible eleven components (including the miscellaneous items) use memory bits to store data.

Table 5-1. Total Resource Usage for MPLS Architecture

Resource	Usage
Total logic cells	3971
Total memory bits	35904
Maximum fan-out	2046
Total fan-out	22607
Average fan-out	4.78

Table 5-2. Resource Usage by Component for MPLS Architecture

Component	Logic Cells	Memory Bits
Error Value Generator	8	0
External Data	418	0
Label Database	6	1280
Label Generator	80	0
LSP Information	335	10496
Main Module	519	0
Packet Generator	925	320
Packet Parser	492	3040
Stacks (input and output)	48	16384
Traffic Engineering Database	966	4384
Miscellaneous items	174	0

A complete analysis of how resources are used is presented in Figure 5-1 and Figure 5-2. To clarify the figures, no component using less than one percent of the resources are illustrated. Consequently there is no representation of the resource usage for the error value generator or the label database.

Figure 5-1 shows how the logic cells are distributed among the various components in the processor. We see a disproportionate number of logic cells used by the main module and the packet generator (nearly half the total number of logic cells combined). Small but significant numbers of logic cells are used with the stacks, traffic engineering database, external data module and LSP module with negligible amounts used for the remaining components. The logic cell distribution strongly suggests that high level logic and packet generation will each require about one quarter of all logic based resources used in a comprehensive hardware implementation of MPLS with the remaining resources nearly equally divided among processing input and output data, parsing packets, managing links and their resources and managing LSP information.

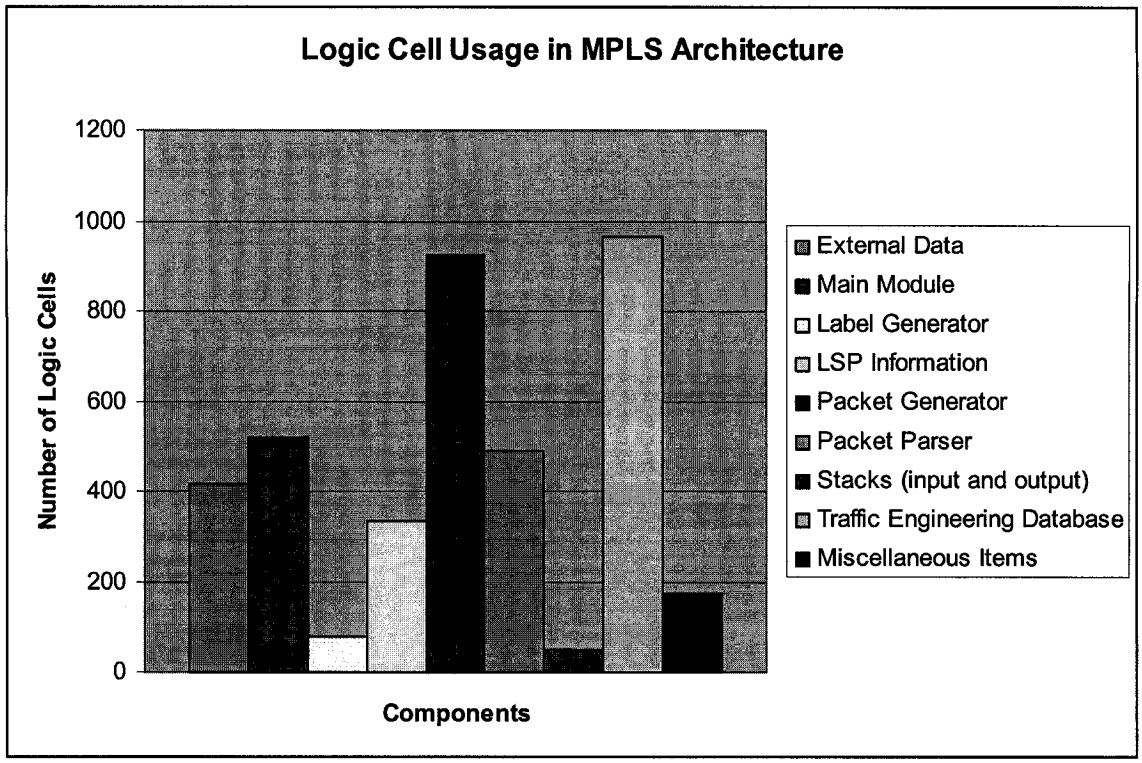


Figure 5-1. Logic Cell Usage by Component in MPLS Architecture

The memory distribution is shown in Figure 5-2. As previously noted in this section not every component in the architecture uses memory and that is shown by the relatively small number of components in the figure. Furthermore, we see larger portions of memory being used among these components than we have seen with logic cells and registers. Nearly half of the memory used in the architecture resides in the stacks used for incoming and outgoing packets. A large and significant amount of memory is used in LSP module with other notable amounts of memory used in the traffic engineering database, packet parser, label database and packet generator. Based on the memory distribution it is reasonable to conclude that most memory in a hardware based MPLS implementation will be allocated to packet storage with decreasing but significant amounts going towards information for LSPs, links, packets and labels.

Figure 5-2 also demonstrates the components providing the most flexibility to the processor because their functionality is most easily enhanced with additional memory. The greatest flexibility is available in the number of packets that can be received and generated, which can be increased by adding more memory to the input and output stacks.

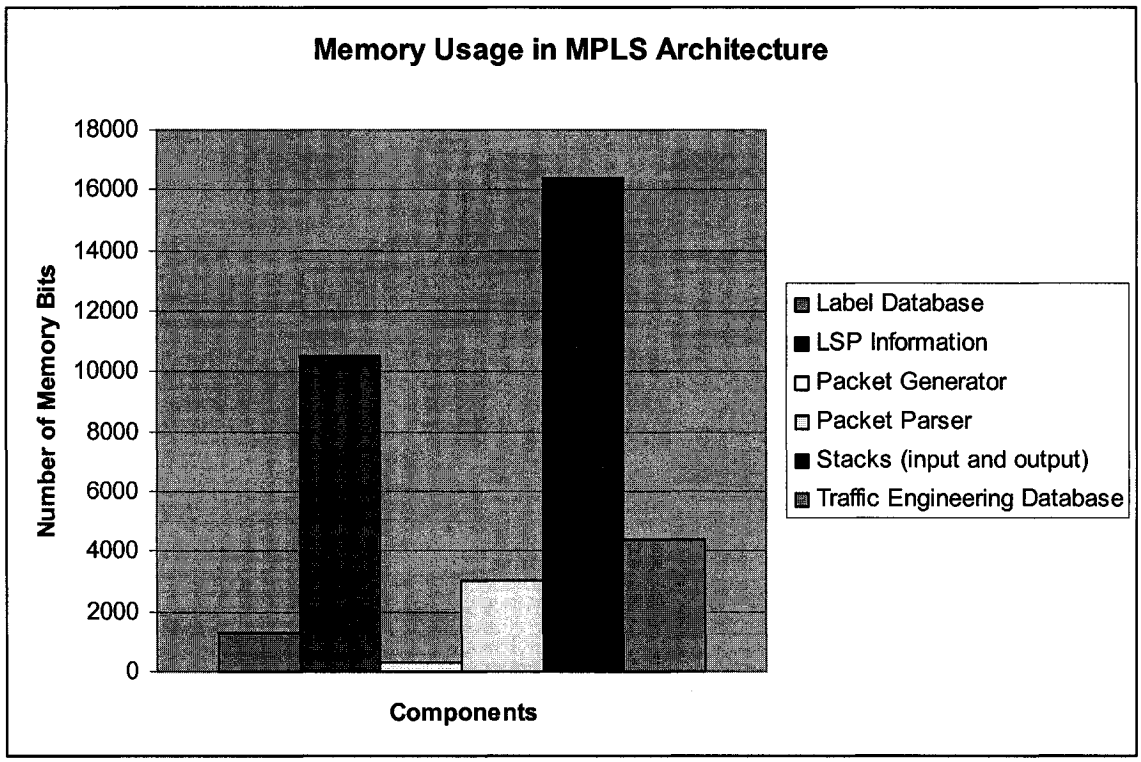


Figure 5-2. Memory Usage by Component in MPLS Architecture

More LSPs can be supported by increasing the memory in the LSP module and in a similar manner more links, packet characteristic, labels and packet types can be considered by increasing memory to the appropriate components.

5.2 Test Scenario

All the results described in this section will illustrate operations for the VPN illustrated in Figure 5-3. The VPN is for an organization with five remote locations connected through a public network using MPLS. Four routers (A, B, C, D) exist in the MPLS domain where A and D are LERs while B and C are LSRs. Six LSPs (A through F) are used for private communications among the sites. Interfaces are denoted with lower case letters (a through r) and all links have the total available bandwidth listed.

Table 5-3 describes the link information for the MPLS network. Each link is associated with a pair of interfaces and has a set amount of bandwidth. However it is not desirable

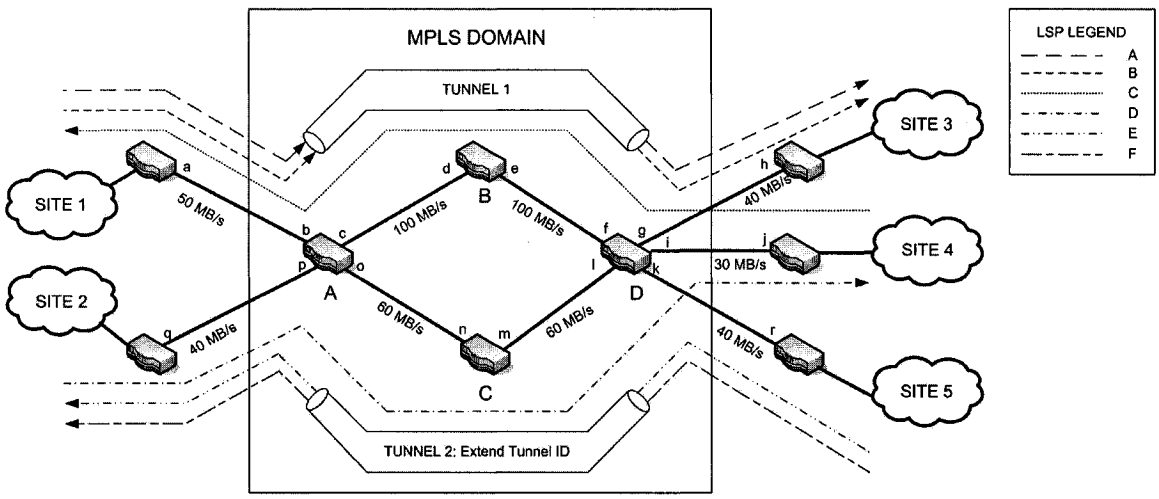


Figure 5-3. VPN Used to Generate Test Results

to have all the bandwidth on every link available for this VPN. The available percentage of bandwidth for LSPs is described below.

Table 5-4 lists all information for the LSPs for the VPN described below. Each LSP is uniquely identified with the source IP address, destination IP address, LSP ID, tunnel ID and extended tunnel ID collectively. LSPs A and B share the same tunnel (1) and consequently have the same tunnel ID. LSPs E and F share tunnel 2 and use an extended tunnel ID.

Interface information is listed in Table 5-5. Each interface has its own IP address and is associated with a router. Interfaces are internally referenced with port numbers to simplify tasks for reserving bandwidth and establishing LSPs.

5.3 Simulated Operations

This section describes how the processor performs a series of tasks in packet processing and generation. Consult Appendix A to see the test bench used to generate the simulated operations in this section. The tasks are performed as router A receives a PATH packet for LSP C and generates the corresponding RESV packet. For the purposes of this scenario it is presumed that there are no errors in either packet (although all necessary checks are performed). It is not possible to illustrate every task the architecture can

Table 5-3. Link Information for the Test Results VPN

Interface pair	Total Bandwidth (MB/s)	Total Percentage allocated to LSPs
a, b	50	80
c, d	100	50
e, f	100	50
g, h	40	75
i, j	30	80
k, r	40	75
l, m	60	50
n, o	60	50
p, q	40	75

Table 5-4. LSP Information for Test Results

Name	Source IP address	Destination IP address	LSP ID	Tunnel ID	Extended tunnel ID	Bandwidth (kB/s)	Refresh rate (min)
A	10.0.0.0	12.0.0.0	1	1	No	2,000	5
B	10.0.0.0	12.0.0.0	2	1	No	4,000	10
C	13.0.0.0	10.0.0.0	3	2	No	5,000	1
D	11.0.0.0	14.0.0.0	4	3	No	10,000	2
E	14.0.0.0	11.0.0.0	5	4	Yes	6,000	12
F	14.0.0.0	14.0.0.0	6	4	Yes	8,000	15

perform in one situation however this scenario will demonstrate how many of the key MPLS tasks are performed.

Each simulation has the high level signals for the architecture and a selected set of internal signals to show which components are operating at a given point in time. The state machine values for every component pertinent to the scenario are given and illustrate a non-zero value when that component is operational. Signals are also shown for the label database and stacks for data input by the user and data generated by the architecture.

Figure 5-4 illustrates a sequence of events when LSP data is being saved to the MPLS architecture. Among the high level signals we see that the wait request signal is high and the main module is operational while data is being saved. No other module is active as all LSP information is saved. Seven pieces of data must be saved per LSP so the entire process of saving LSP data varies depending on the number of LSPs that must be

Table 5-5. Interface Information for the Test Results VPN

Interface	IP address	MPLS Router ID	Local Port number
a	10.0.0.0	N/A	N/A
b	15.0.0.1	A	1
c	15.0.0.2	A	2
d	16.0.0.1	B	1
e	16.0.0.2	B	2
f	18.0.0.1	D	1
g	18.0.0.2	D	2
h	12.0.0.0	N/A	N/A
i	18.0.0.3	D	3
j	13.0.0.0	N/A	N/A
k	18.0.0.4	D	4
l	18.0.0.5	D	5
m	17.0.0.1	C	1
n	17.0.0.2	C	2
o	15.0.0.4	A	3
p	15.0.0.3	A	4
q	11.0.0.0	N/A	N/A
r	14.0.0.0	N/A	N/A

considered by the architecture. A similar process exists for saving link data but it is less time consuming because four pieces of data are saved per link as opposed to seven.

Figure 5-5 shows a cross section of the scenario where both LSP and link information are saved to the architecture. After having saved all parts of an LSP into the architecture, that information is cumulatively saved as an entire LSP. The LSP information module becomes active to record that information as the 'waitrequest' signal remains high to indicate that the architecture is active. After the LSP is saved we observed that all the information for a single link is saved to the architecture and that information is saved into the traffic engineering database (TE DB). No other component is active during that period of time and consequently there is no danger that the data being entered may be corrupted or otherwise altered during the process.

After all the LSP and link data has been correctly saved, packet data must then be entered as shown in Figure 5-6. Packet data is entered one thirty-two bit word at a time, similar to the manner that LSP and link data was entered. We observe that the main module is active while data is being entered however the input data stack is automatically activated

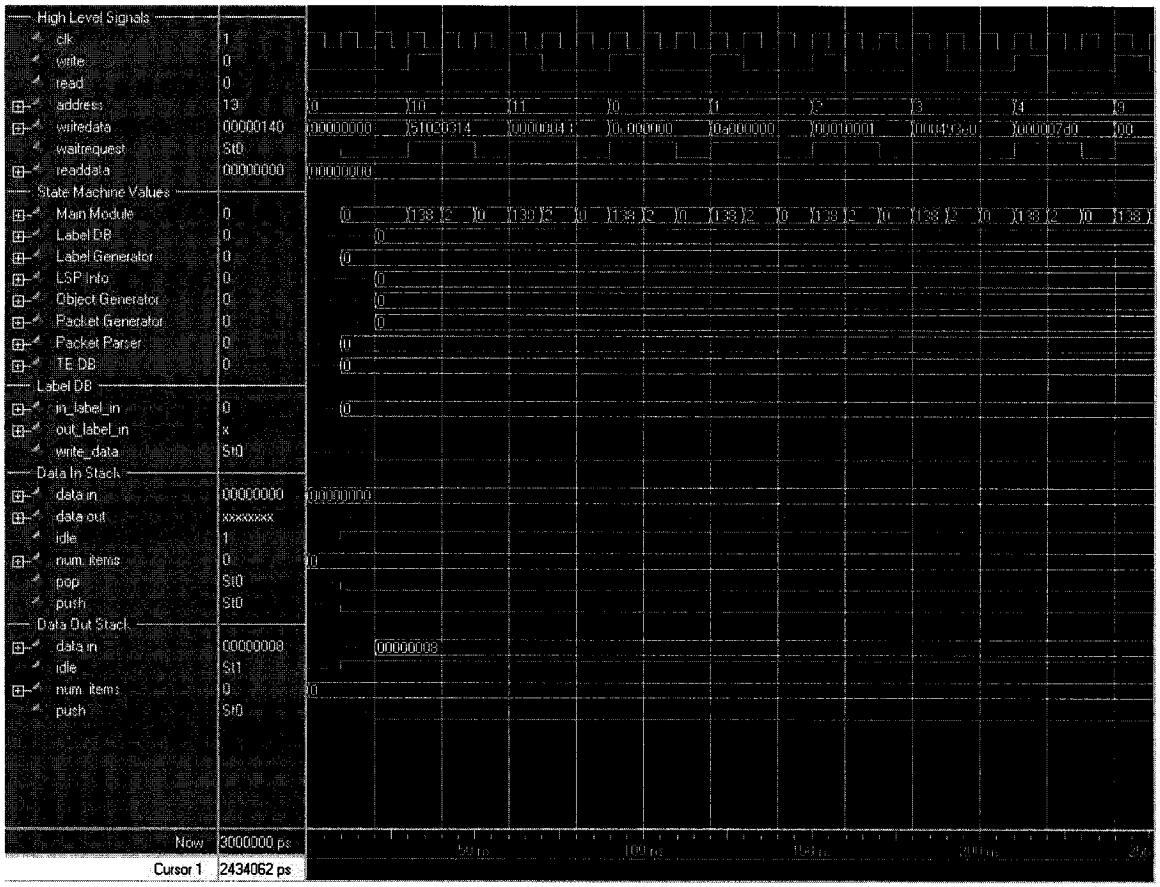


Figure 5-4. Simulation of LSP Data Saved to the Processor

to contain packet data. We see that as the 'push' signal goes high, the 'idle' signal goes low and data is saved. The total number of items (which also acts as a write address) is automatically updated as packet data is received by the architecture. In Figure 5-6, we see the number of stack items increase from two to three and from three to four. That total number increases as more packet data is entered.

Once all data has been entered, the architecture is ready to begin parsing packet data and performing MPLS. The beginning of that process is shown in Figure 5-7 with packet data being parsed. We see that the main module is active but not iterating through any states. So the main module has been put into a wait state as the packet parse goes through several iterations to read and parse numerous amounts of data. The input data stack, previously shown to add data in Figure 5-6, is having its data removed by the parser. We see the 'pop' signal being asserted several times and the total number of

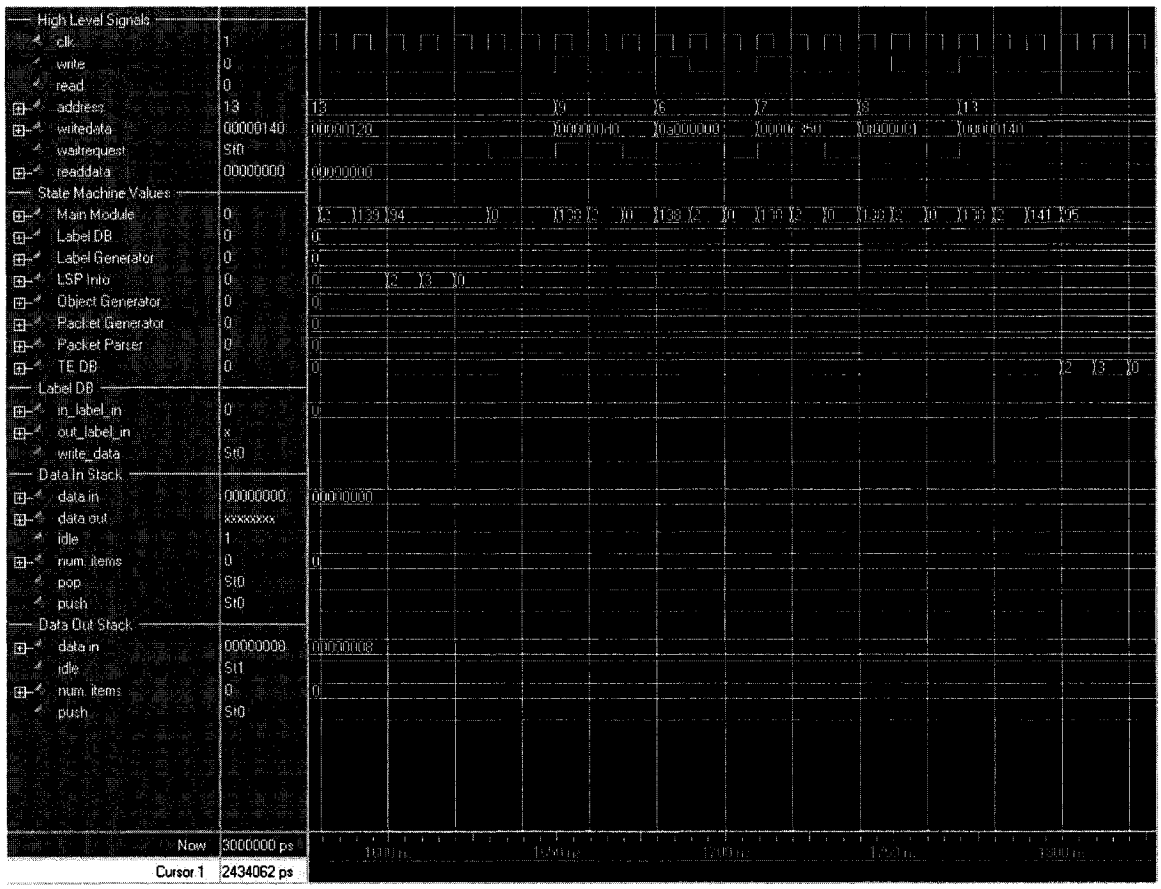


Figure 5-5. Simulation of LSP and Link Data Saved to the Processor

items decreasing. An error free scenario is being presented so all packet data is processed without interruption. However, if certain errors are found while processing packet data (e.g. an unsupported version number in the packet header), the entire process is automatically interrupted.

Once the packet has been parsed and now errors have been found, the first task is to verify that the LSP described in the packet matches an LSP that has been entered into the architecture. That task is performed in Figure 5-8. We see that the main module is still active but no longer in a wait state, as it was in Figure 5-7. A series of different states are being used to read packet data describing the LSP. The packet parser is also active retrieving the desired values. Both stacks have no data (as all the input stack data has now been processed by the parser) and no other component is active. The input for the output label at the label database changes because the output of the parser is connected to different components which include the label database. Since we are not reading label

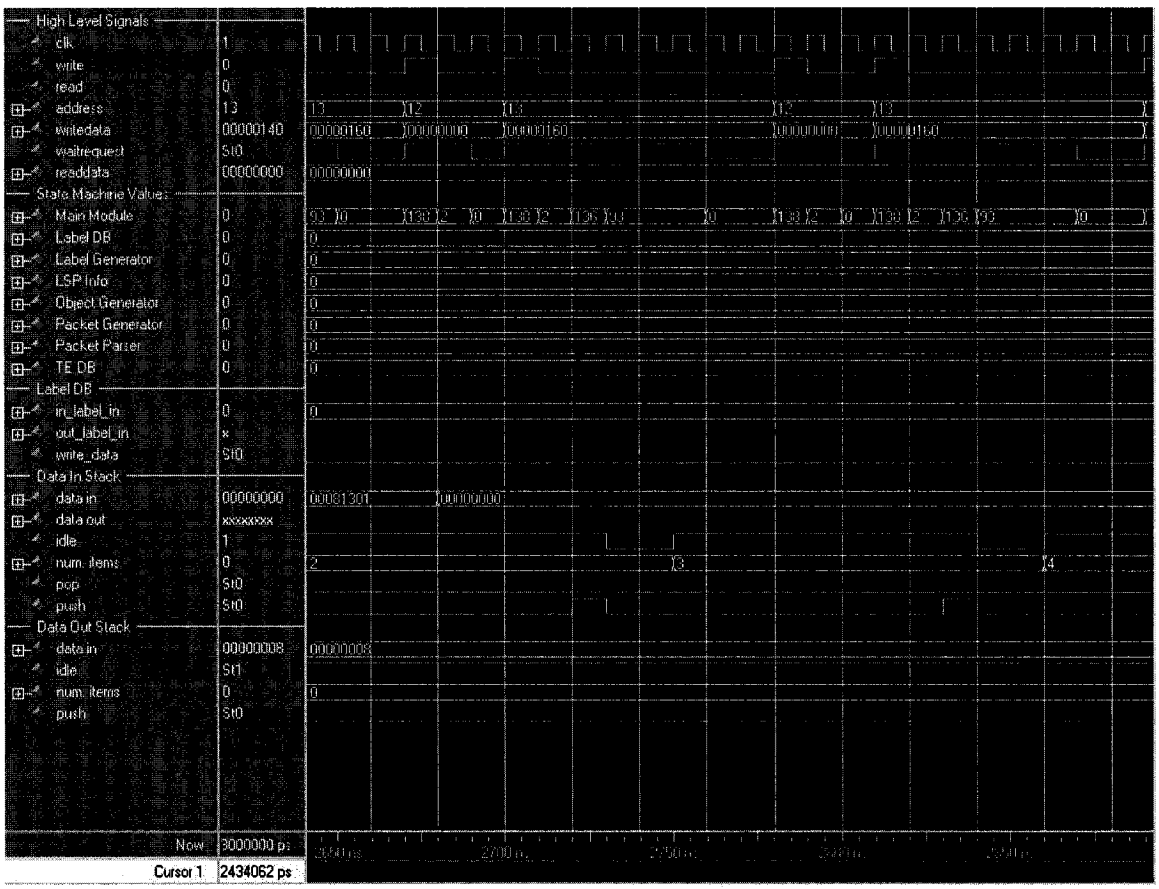


Figure 5-6. Simulation of Packet Data being Saved to the Processor

data the input to the label database is ignored at this time. The 'write' signal for the label database is never asserted while the LSP data is read from the packet parser.

When generating RESV packets, checks must be performed to determine if there are sufficient resources (e.g. bandwidth) to support the LSP on all pertinent links. Those tasks are performed in Figure 5-9 where the main module and the traffic engineering database are active during the process. Two links (downstream and upstream) are both checked to determine if there is enough bandwidth remaining to support the LSP. We see two wait states in the main module to accommodate each of those situations. Once the links have been verified and the bandwidth has been reserved, the label generator becomes active to create a label that will be used for the LSP in the architecture.

Figure 5-10 shows the RESV packet being generated after all necessary checks have been performed. The main module is in a prolonged wait state while the packet generator and

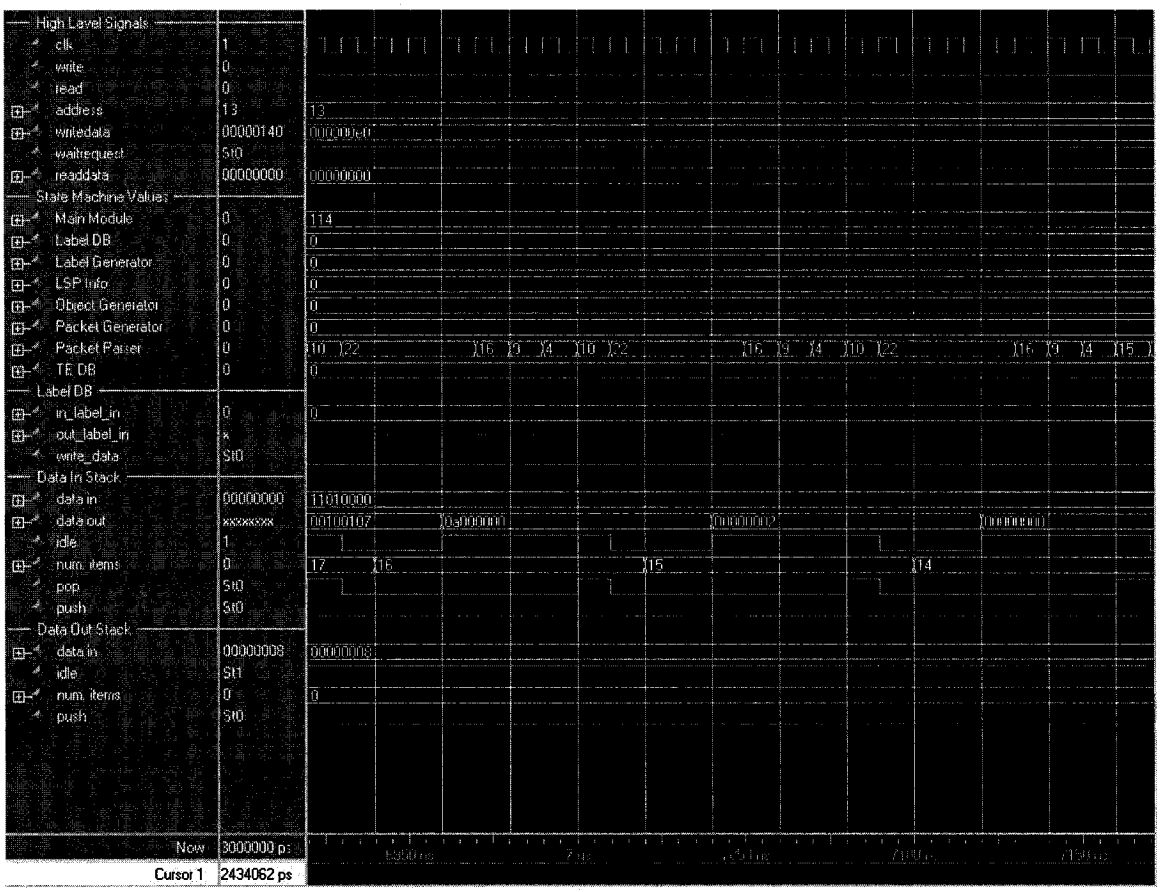


Figure 5-7. Simulation of Packet Data being Parsed by the Processor

object generator are active in producing the packet. The packet generator goes through a series of wait states since it is used to coordinate the sequence of packets that must be generated for the packet. Each packet has a specific set of objects it must contain so the packet generator logic must wait while each of those objects is generated. As the packet is generated the output data stack is active in saving the relevant data. The 'push' signal is asserted several times as data is saved and the total number of stack elements is automatically updated.

Once the packet has been generated, the architecture becomes idle as shown in Figure 5-11. The only action taking place once the packet is complete is an update of the status register so the architecture becomes idle almost immediately. From that point, the packet data can be read in addition to label information and the status register at the discretion of the user. When reading packet data, the output data stack will automatically be updated to reflect the loss of data as it's made available to the user.

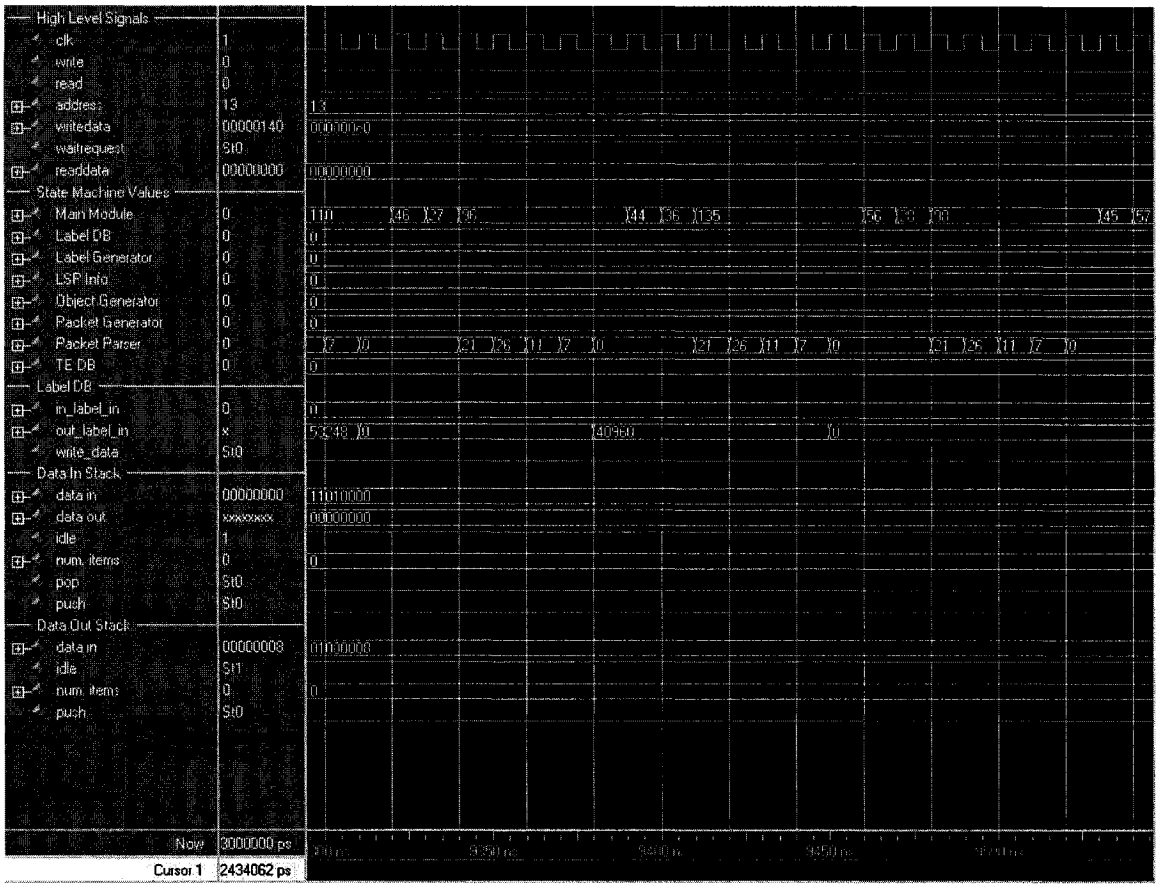


Figure 5-8. Simulation of LSP Data from a Packet being Verified by the Processor

Other actions possible in the architecture have not been explicitly shown because of their similarity to those that have been presented. Those actions include the generation and processing of PATHERR, PATHTEAR, RESVERR and RESVTEAR packets. PATHERR and PATHTEAR packets are generated when errors are found with PATH and RESV packet respectively. If an error had been found with the PATH packet presented in this scenario, a PATHERR packet would have been generated without the accompanying tasks (label generation, resource reservation and verification) instead of the RESV packet. PATHTEAR and RESVTEAR packets are generated based on a direct request by the user so the process is similar to that shown for generating the RESV packet alone. The only accompanying task in that regard is to specify the LSP for which to generate the packet. That task is accomplished in the same manner that a single piece of LSP or link data is written to the architecture.

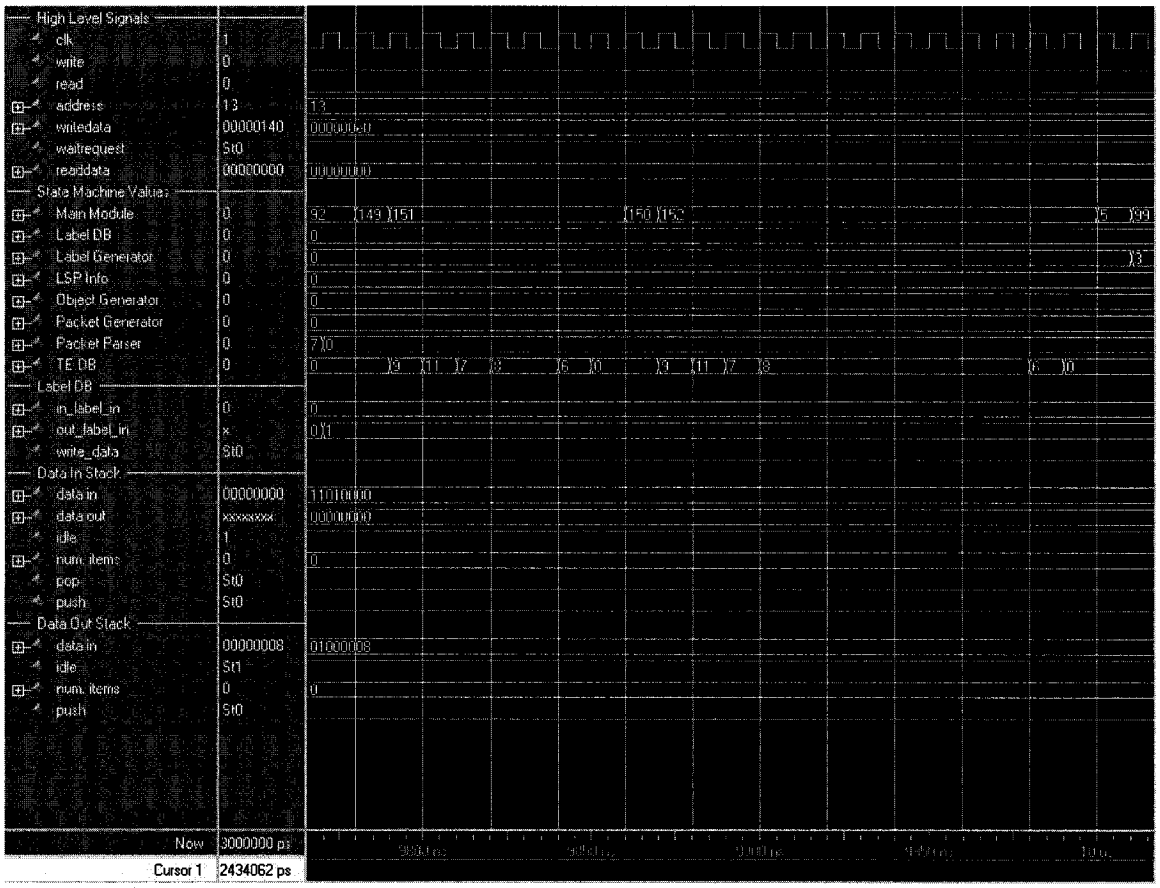


Figure 5-9. Simulation of Link Information being Updated in the Processor

5.4 Processor Performance

The performance of the processor can be evaluated by examining the number of cycles required to perform a series of tasks, most notably those illustrated in the previous section with simulations. Performance will be based only on the processing that occurs in the architecture and will not include the periods of time to detect that the architecture is idle. Performance analysis will be performed on a worst-case basis.

One task is to configure the processor with all relevant information (router configuration, LSPs and links) before it can process any packets. Two cycles are required to write a thirty-two bit word of data into the processor, regardless of its intended use. Two words are required for router configuration, six words are required for a single LSP and four words are necessary for a link. Furthermore, the task of collectively saving all LSP or

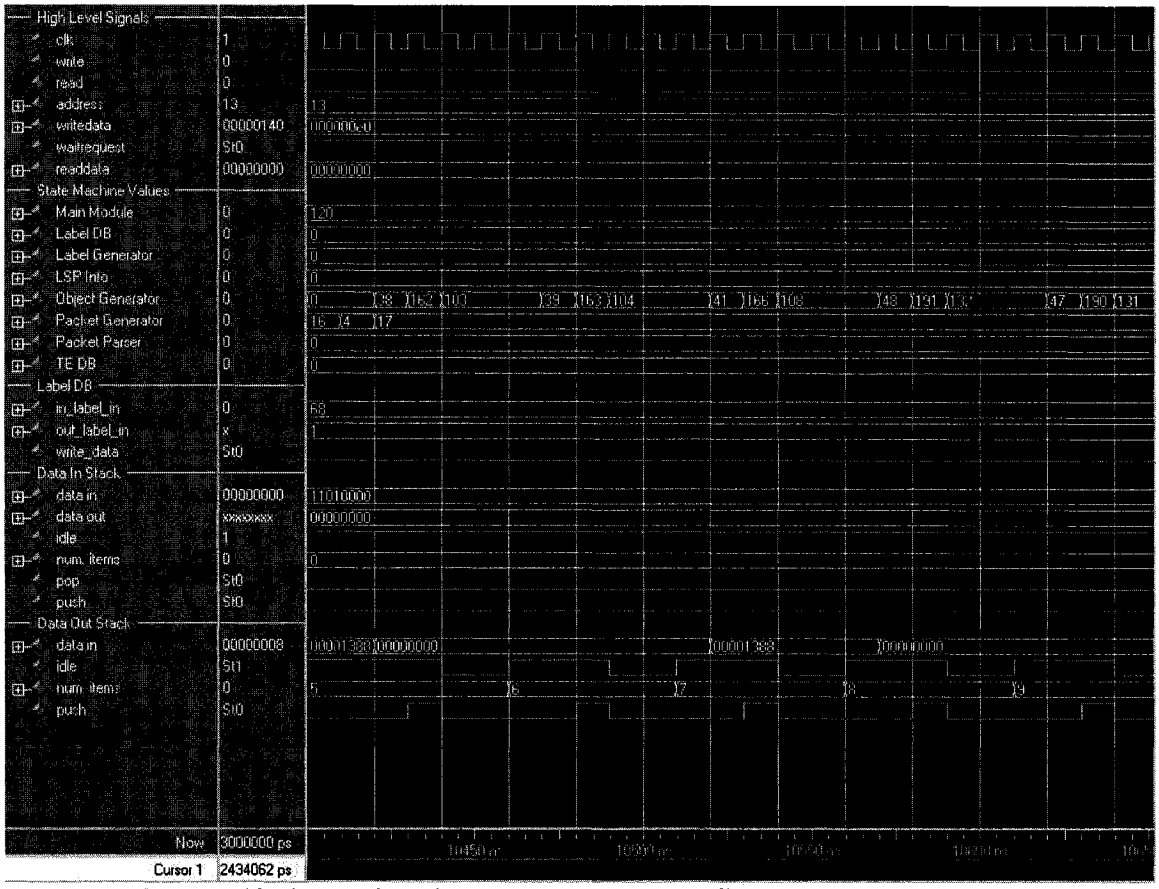


Figure 5-10. Simulation of an RESV Packet being Generated by the Processor

link information as a unit requires six cycles and must be performed after the correct information has been provided. Therefore, the number of cycles to configure the architecture with configuration, LSP and link information is shown in equation (6.1).

$$C_{config} = 4 + 18P + 14L \tag{6.1}$$

Variables P and L represent the numbers of LSPs and links respectively. The process for entering packet information is similar to that for configuration, LSP or link information. A thirty-two bit word must first be entered, requiring two cycles and it must then be saved as packet data, requiring six cycles. Therefore, the total number of cycles to enter a packet is expressed in equation (6.2).

$$C_{packet} = 8W \tag{6.2}$$

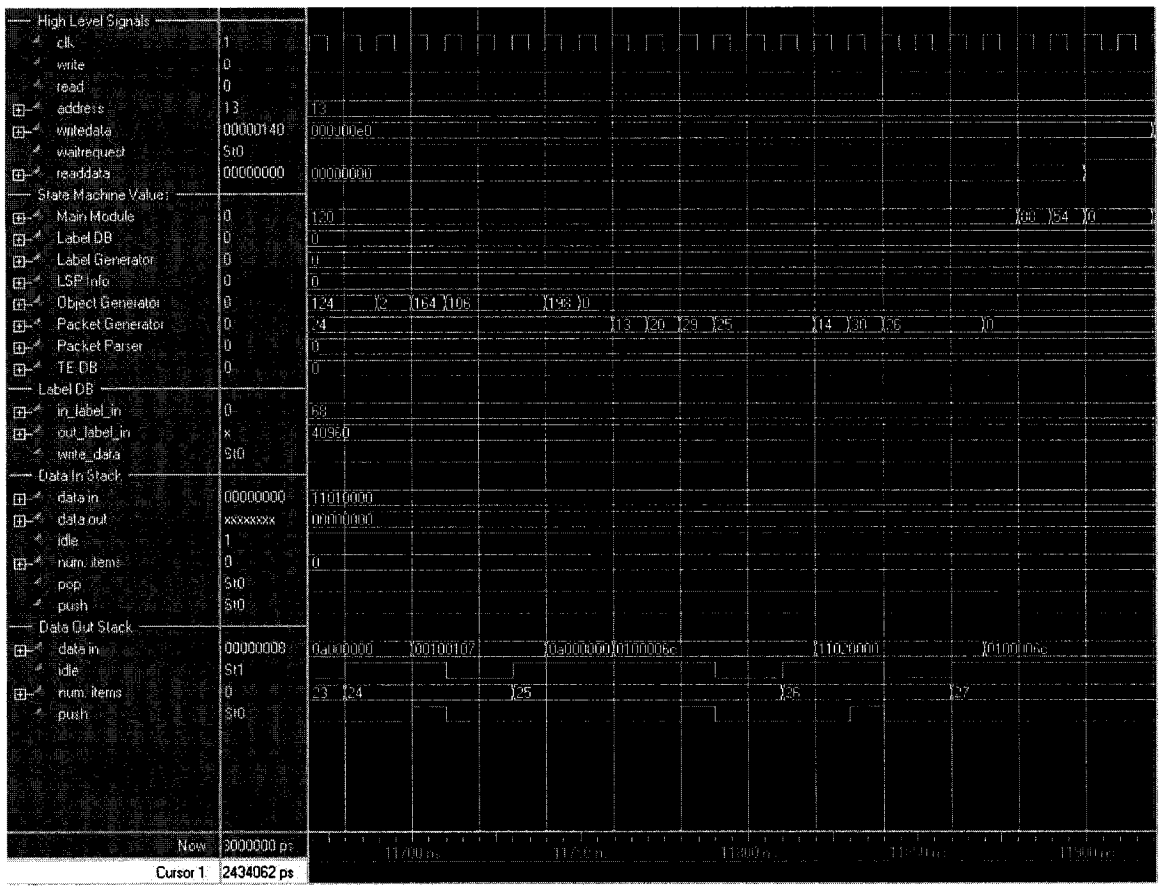


Figure 5-11. Simulation of when the RESV Packet is Complete

W represents the number of thirty-two bit words in the packet. The number of cycles to parse a packet varies greatly depending on numerous factors. Assuming the packet contains no errors and is supported by the architecture, processing times vary with the number of objects and the sequence of objects (the architecture can process objects in any order. Numerous searches are performed among packet types and object types that require different number of cycles to execute. A worst case number of cycles for packet parsing is observed by considering the time to generate the large packet supported by the architecture. RESV packets are the largest supported packets (twenty-seven thirty-two bit words) and it has been observed that no more than 420 cycles are required to parse any packet in the absence of errors. Parsing automatically stops in the presence of an error.

Verifying LSPs requires reading five pieces of information from the parser and search the LSP information module to determine if any LSP contains those characteristics. Seven

cycles are required to read a single element out of the packet parser and save it in a register for LSP searching. Three cycles are required to initialize the search and three cycles are required to determine if an LSP with a given set of characteristics exists in the MPLS architecture. Consequently equation (6.3) shows the worst case time for verifying LSP information based on packet data in the architecture.

$$C_{verify} = 38 + 3P \tag{6.3}$$

The number of cycles to update the downstream and upstream links of an LSP depends on the total number of links that have been entered in the traffic engineering database. The desired bandwidth can be read from the packet parser and that task, along with miscellaneous high level processing, requires eight cycles. A search must then occur to find the link that has the port number entered in the LSP information module. One cycle is required to reserve bandwidth once it has been determined that sufficient bandwidth exists and the desired link has been found. As with LSP verification three cycles are required to initialize begin the search and to determine if a given link meets the search criteria. Therefore equation (6.4) describes the number of cycles needed to update the bandwidth for the downstream and upstream links of an LSP.

$$C_{link} = 16 + 6L \tag{6.4}$$

A constant number of cycles are needed to update label information assuming there are no errors. Nine cycles are necessary to generate a label, ensure that it is unique and update the label database.

Generating a packet requires a variable number of cycles for successful completion. One reason is that each object being generated has a variable length that must be accounted for in the total length of the packet. So a variable number of objects will greatly affect the number of cycles to generate an entire packet. Another reason is that there are various sources (packet parser, label generator, LSP information, error generator, etc.) that may be needed for data, each using a different number of cycles to retrieve data. The

worst case number of cycles for packet generation was observed for the RESV packet is 250 cycles.

The previously described sequence of events is the longest that can occur at any time in the architecture. Therefore, the overall performance will be done by evaluating what happens in that sequence of events to enter and process a single packet. Equation (6.5) describes the total number of cycles required under those circumstances.

$$C_{total} = 8W + 20L + 21P + 737 \tag{6.5}$$

Figure 5-12 illustrates the total number of cycles required to enter and process a packet with varying numbers of LSPs and links to consider in the architecture. The numbers of LSPs and links have an approximately equal effect on the total number of cycles. Routers using fewer than 11 links will require less than 1200 cycles to perform all tasks related to a single packet using the architecture. For many LSPs on a router with few links, or a small VPN using many links on an router no more than 1800 cycles are require for a single packet. When both the number of LSPs and links increase to relatively large numbers a total of 2400 cycles are required given the limitations of the processor. Therefore, no task requires more than 2400 cycles, which is 48 μs with a 50 MHz clock.

In processing a packet, three tasks require a significant portion of processing time under all scenarios supported by the architecture. The first task is configuring the architecture with LSP and link information and the percentage of processing time required is shown in Figure 5-13. We see that the number of LSPs contributes more greatly to the number to the percentage of time required for configuration. A small number of LSPs and a large number of links may require up to 30% of the total processing time while the opposite scenario may require up to 40% processing time. Routers with few links in a VPN with few LSPs will typically require up to 20% of the total processing time but nearly half of the total time may be required for many LSPs and a large number of links.

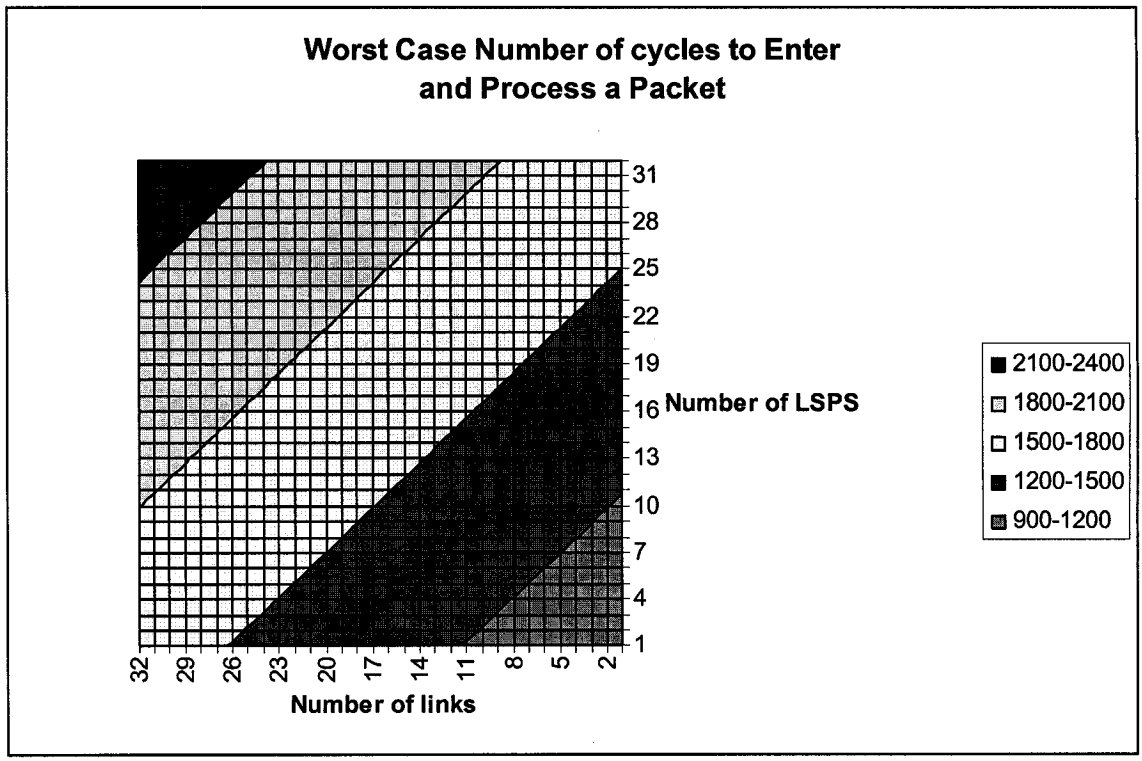


Figure 5-12. Worst Case Number of Cycles to Enter and Process a Packet

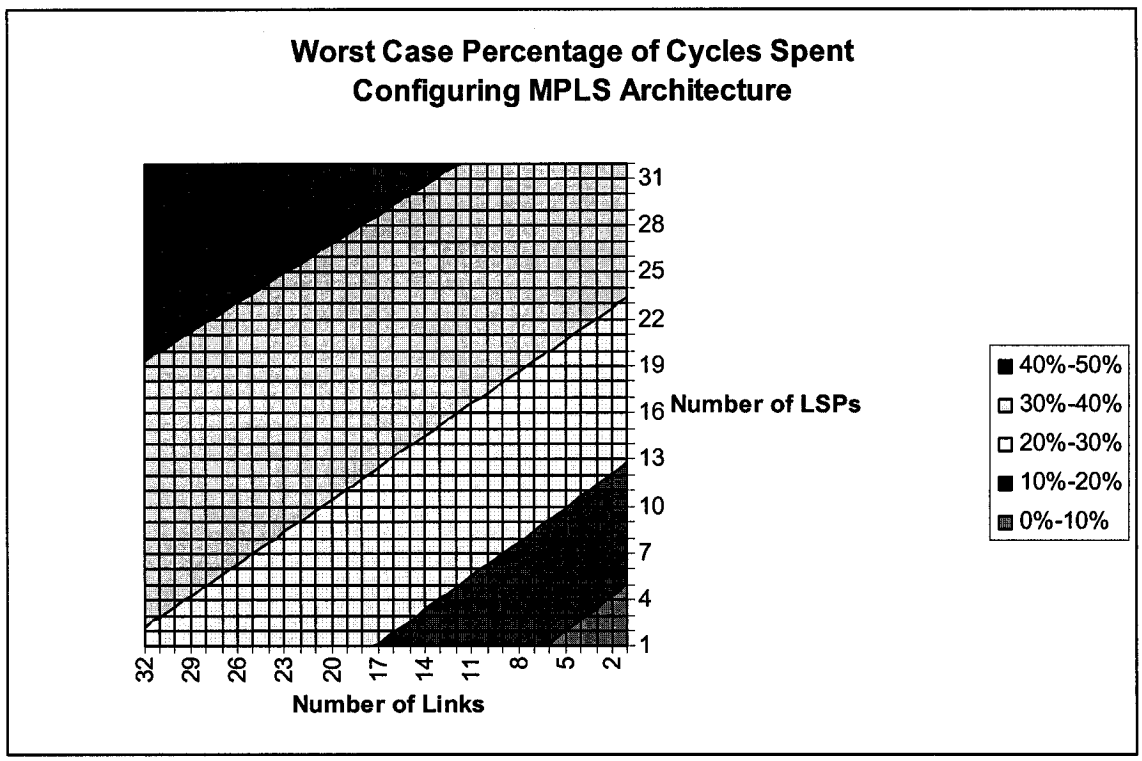


Figure 5-13. Worst Case Percentage of Cycles to Configure the MPLS Architecture

Figure 5-15 shows the percentage of cycles required to generate a packet after having performed all necessary tasks. It can be immediately seen that Figure 5-14 and Figure 5-15 are identical in the distribution of percentages among the number of LSPs and links. Therefore the number of LSPs and links has approximately the same effect. However the ranges and limits among percentages are different. Each range of data is 3% with packet generation where it is 5% with packet parsing so total percentage decrease for packet generation, as the number of LSPs and links becomes large, is less than what has been observed for packet parsing. The amount of time required to generate a packet ranges from 9% to 27% depending on the number of LSPs and links used.

The results and analysis described in this chapter demonstrate that the MPLS processor consumes no more than 2400 cycles to process an RSVP-TE packet, check for several errors, update bandwidth and label information and generate an appropriate response accounting for the router type and other factors. The processor would yield a worst case execution time of 48 μs with a 50 MHz clock. Path setup delays per switch are in the order of milliseconds [41]. Therefore, the MPLS processor can substantially improve the processing time required to establish LSPs.

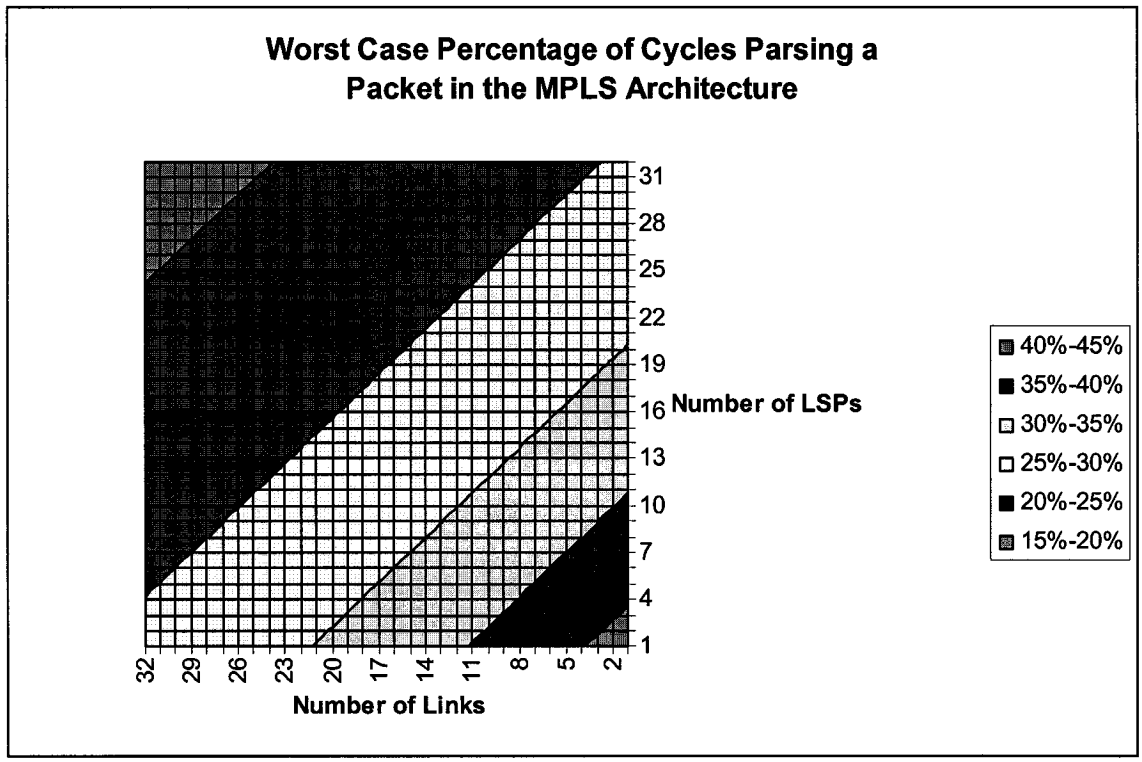


Figure 5-14. Worst Case Percentage of Cycles to Parse a Packet

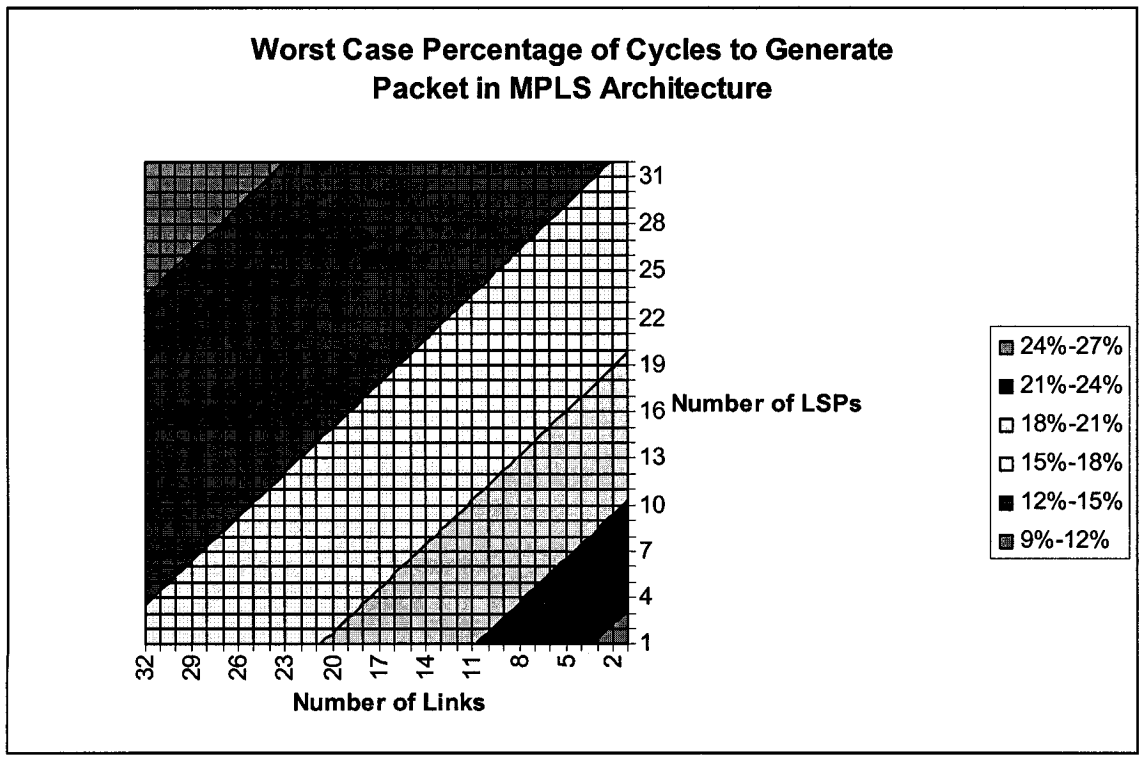


Figure 5-15. Worst Case Percentage of Cycles to Generate a Packet

Chapter 6

6 Conclusion

MPLS may also be used to provide the functionality of ATM to IP networks for the purposes of VPN implementation. MPLS can be used to set up LSPs with specific resource requirements. Traffic engineering (TE) and Quality of Service (QoS) can be used to prioritize network resources and provide a minimum level of performance. To achieve performance comparable to ATM/SONET, where 50 ms resilience times are possible, hardware implementations of MPLS must be used as opposed to software implementations prevalent today. A reconfigurable architecture and a hardware processor implementation are proposed in this thesis to improve MPLS performance. The processor was implemented using an FPGA. Results indicate that no task consumes more than 2400 cycles, requiring 48 μs to execute with a 50 MHz. Such an execution time represents a significant improvement when compared to path setup delays in the order of milliseconds.

6.1 Concepts Addressed in this Thesis

MPLS improves performance in IP networks by attaching labels to packets and switching packets based on the label alone. Each label value is associated with an LSP so several packets may follow the same path with a dedicated set of resources. Further performance improvements are possible by executing MPLS tasks directly in hardware. Hardware implementations exist in the literature for different aspects of MPLS. However, no implementation allows for a signaling protocol, label management and TE at the same time.

This thesis has introduced a hardware processor for the implementation of MPLS using RSVP-TE as its signaling protocol. The processor was specifically designed to interact

with software as part of an embedded system. The architecture was divided into several modules, each responsible for a set of tasks necessary for RSVP-TE and label management. A subset of RSVP-TE was implemented including functionality for basic packet types, error processing and tearing down established LSPs.

6.2 Contributions

The hardware processor proposed by this thesis introduced several significant research contributions, which can be summarized as follows:

- Improving performance for tasks to establish and teardown LSPs accounting for a subset of errors that may occur in the process, emphasizing processor intensive and common tasks for greatest improvement.
- Provide an infrastructure for TE at the hardware level, allowing for additional TE tasks to be added in the future without affecting other tasks.
- Provide an infrastructure for label management and retrieval at the hardware level.

6.3 Future Research

The encouraging results of this thesis leave several areas of future research. Those areas include but are not limited to the following:

- Adding an existing routing protocol (OSPF) implementation to the processor so a user need only specify the network topology and a series of constraints to establish LSPs and manage labels.
- Adding more basic features to RSVP-TE in hardware to increase its performance most notably a checksum and the ability to generate packets after the refresh period has expired.
- Support additional packet and object types in RSVP-TE.
- Explicit support for 'shared explicit' and 'wildcard filter' LSPs and the use of label stacking.

Appendix A. MPLS Processor Test Bench

This section illustrates a test bench to demonstrate how interaction with the MPLS processor is performed to execute various tasks. The reset signal of the MPLS processor was internally grounded for testing purposes and is not illustrated below.

```
// Designer: Raymond Peterkin
// University of Ottawa
// File Name: mpls_establish_lsps_tb.v
//
// This file contains the logic necessary to configure router A will
// all LSP information (LSPs A through F) and link information
(b,c,o,p)

// Change time scale
`timescale 1ns/1ps

// Define relevant macros
// Macros to define the types of data that can be saved directly to
registers
// from the user
`define DEST_ADDR 5'b00000            // LSP info
`define SRC_ADDR 5'b00001             // LSP info
`define TUNNEL_LSP_ID 5'b00010        // LSP info
`define REFRESH_DATA 5'b00011         // LSP info
`define PEAK_DATA_RATE_USER 5'b00100  // LSP info
`define QOS_INFO 5'b00101             // LSP info
`define NEXT_HOP_ADDR 5'b00110        // Link info
`define BANDWIDTH 5'b00111            // Link info
`define PORT_ADDR 5'b01000            // Link info
`define PORT_PCT_DATA 5'b01001        // LSP and Link info
`define COS_FLAGS_TTL_DATA 5'b01010   // One time config. info
`define INITIAL_LABEL 5'b01011        // One time config. info
`define STACK_DATA 5'b01100           // Packet data for input stack
`define CONTROL_REGISTER 5'b01101     // User commands

`define INPUT_STACK_ADDRESS 5'b01110
`define PACKET_DATA_ADDRESS 5'b01111
`define STATUS_ADDRESS 5'b10000
`define IP_DEST_ADDR_ADDRESS 5'b10001
`define IP_SRC_ADDR_ADDRESS 5'b10010
`define IN_LABEL_ADDRESS 5'b10011
`define OUT_LABEL_ADDRESS 5'b10100

`define NO_DATA 5'b11111

// Define macros describing the commands associated with the control
register
```

```
`define RESET 32'h00000000
`define CLEAR_INPUT_BUFFER 32'h00000020
`define CLEAR_OUTPUT_BUFFER 32'h00000040
`define CLEAR_LSP_INFO 32'h00000060
`define CLEAR_TE_DB_INFO 32'h00000080
`define CREATE_PATH_TEAR 32'h000000a0
`define CREATE_RESV_TEAR 32'h000000c0
`define PROCESS_PACKET 32'h000000e0
`define ESTABLISH_LSPS 32'h00000100
`define SAVE_LSP_INFO 32'h00000120
`define SAVE_TE_DB_INFO 32'h00000140
`define SAVE_PACKET_DATA 32'h00000160
module mpls_sim_establish_lsps;

// Define input signals
reg   clk;
reg   write;
reg   read;
reg   [4:0] address;
reg   [31:0] writedata;

// Define output signals
wire  waitrequest;
wire  [31:0] readdata;

MPLS dut(
.clk(clk),
.write(write),
.read(read),
.address(address),
.writedata(writedata),
.waitrequest(waitrequest),
.readdata(readdata)
);

initial
begin
  // Set initial values
  clk = 1'b1;
  write = 1'b0;
  read = 1'b0;
  address = `DEST_ADDR;
  writedata = 32'h00000000;
end

always
#5 clk = !clk;

// Perform testing for router A of test scenario
initial
begin

  // ***** BEGIN WRITING CONFIGURATION INFORMATION *****

  // Enter config information for test case
  // Set write data for the following:
  // R = 0 (LER)
```

```
// CoS = 5
// Flags pkt = 1
// Flags error = 2
// Flags style = 3
// Initial TTL = 20
#30 address = `COS_FLAGS_TTL_DATA;
    read = 1'b0;
    write = 1'b1;
    writedata = 32'h51020314;

// Deassert write signal
#10 address = `COS_FLAGS_TTL_DATA;
    read = 1'b0;
    write = 1'b0;
    writedata = 32'h51020314;

// Initial label: 67
#20 address = `INITIAL_LABEL;
    read = 1'b0;
    write = 1'b1;
    writedata = 32'h00000043;

// Deassert write signal
#10 address = `INITIAL_LABEL;
    read = 1'b0;
    write = 1'b0;
    writedata = 32'h00000043;

// ***** END WRITING CONFIGURATION INFORMATION *****

// ***** BEGIN WRITING LSP INFORMATION *****

// Begin writing all information for LSP A
// Write destination address: 12.0.0.0
#20 address = `DEST_ADDR;
    read = 1'b0;
    write = 1'b1;
    writedata = 32'h0c000000;

// Deassert write signal
#10 address = `DEST_ADDR;
    read = 1'b0;
    write = 1'b0;
    writedata = 32'h0c000000;

// Write source address: 10.0.0.0
#20 address = `SRC_ADDR;
    read = 1'b0;
    write = 1'b1;
    writedata = 32'h0a000000;

// Deassert write signal
#10 address = `SRC_ADDR;
    read = 1'b0;
    write = 1'b0;
    writedata = 32'h0a000000;
```

```
// Tunnel ID: 1, LSP ID: 1
#20 address = `TUNNEL_LSP_ID;
    read = 1'b0;
    write = 1'b1;
    writedata = 32'h00010001;

// Deassert write signal
#10 address = `TUNNEL_LSP_ID;
    read = 1'b0;
    write = 1'b0;
    writedata = 32'h00010001;

// Refresh time: 5 min (300000 ms)
#20 address = `REFRESH_DATA;
    read = 1'b0;
    write = 1'b1;
    writedata = 32'h000493e0;

// Deassert write signal
#10 address = `REFRESH_DATA;
    read = 1'b0;
    write = 1'b0;
    writedata = 32'h000493e0;

// Peak data rate: 2000 kb/s
#20 address = `PEAK_DATA_RATE_USER;
    read = 1'b0;
    write = 1'b1;
    writedata = 32'h000007d0;

// Deassert write signal
#10 address = `PEAK_DATA_RATE_USER;
    read = 1'b0;
    write = 1'b0;
    writedata = 32'h000007d0;

// Extend tunnel ID = 0
// Upstream port = 1
// Downstream port = 2
// TE DB port ID = 0
// Percentage = 0
#20 address = `PORT_PCT_DATA;
    read = 1'b0;
    write = 1'b1;
    writedata = 32'h00009000;

// Deassert write signal
#10 address = `PORT_PCT_DATA;
    read = 1'b0;
    write = 1'b0;
    writedata = 32'h00009000;

// Save LSP information
#20 address = `CONTROL_REGISTER;
    read = 1'b0;
    write = 1'b1;
    writedata = `SAVE_LSP_INFO;
```

```
// Deassert write signal
#10 address = `CONTROL_REGISTER;
    read = 1'b0;
    write = 1'b0;
    writedata = `SAVE_LSP_INFO;

// Wait for LSP information to be written
wait(waitrequest == 1'b0);

// End writing all information for LSP A

// Begin writing all information for LSP B
// Write destination address: 12.0.0.0
#20 address = `DEST_ADDR;
    read = 1'b0;
    write = 1'b1;
    writedata = 32'h0c000000;

// Deassert write signal
#10 address = `DEST_ADDR;
    read = 1'b0;
    write = 1'b0;
    writedata = 32'h0c000000;

// Write source address: 10.0.0.0
#20 address = `SRC_ADDR;
    read = 1'b0;
    write = 1'b1;
    writedata = 32'h0a000000;

// Deassert write signal
#10 address = `SRC_ADDR;
    read = 1'b0;
    write = 1'b0;
    writedata = 32'h0a000000;

// Tunnel ID: 1, LSP ID: 2
#20 address = `TUNNEL_LSP_ID;
    read = 1'b0;
    write = 1'b1;
    writedata = 32'h00010002;

// Deassert write signal
#10 address = `TUNNEL_LSP_ID;
    read = 1'b0;
    write = 1'b0;
    writedata = 32'h00010002;

// Refresh time: 10 min (600000 ms)
#20 address = `REFRESH_DATA;
    read = 1'b0;
    write = 1'b1;
    writedata = 32'h000927c0;

// Deassert write signal
#10 address = `REFRESH_DATA;
```

```
    read = 1'b0;
    write = 1'b0;
    writedata = 32'h000927c0;

// Peak data rate: 4000 kb/s
#20 address = `PEAK_DATA_RATE_USER;
    read = 1'b0;
    write = 1'b1;
    writedata = 32'h00000fa0;

// Deassert write signal
#10 address = `PEAK_DATA_RATE_USER;
    read = 1'b0;
    write = 1'b0;
    writedata = 32'h00000fa0;

// Extend tunnel ID = 0
// Upstream port = 1
// Downstream port = 2
// TE DB port ID = 0
// Percentage = 0
#20 address = `PORT_PCT_DATA;
    read = 1'b0;
    write = 1'b1;
    writedata = 32'h00009000;

// Deassert write signal
#10 address = `PORT_PCT_DATA;
    read = 1'b0;
    write = 1'b0;
    writedata = 32'h00009000;

// Save LSP information
#20 address = `CONTROL_REGISTER;
    read = 1'b0;
    write = 1'b1;
    writedata = `SAVE_LSP_INFO;

// Deassert write signal
#10 address = `CONTROL_REGISTER;
    read = 1'b0;
    write = 1'b0;
    writedata = `SAVE_LSP_INFO;

// Wait for LSP information to be written
wait(waitrequest == 1'b0);

// End writing all information for LSP B

// Begin writing all information for LSP C
// Write destination address: 10.0.0.0
#20 address = `DEST_ADDR;
    read = 1'b0;
    write = 1'b1;
    writedata = 32'h0a000000;

// Deassert write signal
```

```
#10 address = `DEST_ADDR;
    read = 1'b0;
    write = 1'b0;
    writedata = 32'h0a000000;

// Write source address: 13.0.0.0
#20 address = `SRC_ADDR;
    read = 1'b0;
    write = 1'b1;
    writedata = 32'h0d000000;

// Deassert write signal
#10 address = `SRC_ADDR;
    read = 1'b0;
    write = 1'b0;
    writedata = 32'h0d000000;

// Tunnel ID: 2, LSP ID: 3
#20 address = `TUNNEL_LSP_ID;
    read = 1'b0;
    write = 1'b1;
    writedata = 32'h00020003;

// Deassert write signal
#10 address = `TUNNEL_LSP_ID;
    read = 1'b0;
    write = 1'b0;
    writedata = 32'h00020003;

// Refresh time: 1 min (60000 ms)
#20 address = `REFRESH_DATA;
    read = 1'b0;
    write = 1'b1;
    writedata = 32'h0000ea60;

// Deassert write signal
#10 address = `REFRESH_DATA;
    read = 1'b0;
    write = 1'b0;
    writedata = 32'h0000ea60;

// Peak data rate: 5000 kb/s
#20 address = `PEAK_DATA_RATE_USER;
    read = 1'b0;
    write = 1'b1;
    writedata = 32'h00001388;

// Deassert write signal
#10 address = `PEAK_DATA_RATE_USER;
    read = 1'b0;
    write = 1'b0;
    writedata = 32'h00001388;

// Extend tunnel ID = 0
// Upstream port = 2
// Downstream port = 1
// TE DB port ID = 0
```

```
// Percentage = 0
#20 address = `PORT_PCT_DATA;
    read = 1'b0;
    write = 1'b1;
    writedata = 32'h00010800;

// Deassert write signal
#10 address = `PORT_PCT_DATA;
    read = 1'b0;
    write = 1'b0;
    writedata = 32'h00010800;

// Save LSP information
#20 address = `CONTROL_REGISTER;
    read = 1'b0;
    write = 1'b1;
    writedata = `SAVE_LSP_INFO;

// Deassert write signal
#10 address = `CONTROL_REGISTER;
    read = 1'b0;
    write = 1'b0;
    writedata = `SAVE_LSP_INFO;

// Wait for LSP information to be written
wait(waitrequest == 1'b0);

// End writing all information for LSP C

// Begin writing all information for LSP D
// Write destination address: 14.0.0.0
#20 address = `DEST_ADDR;
    read = 1'b0;
    write = 1'b1;
    writedata = 32'h0e000000;

// Deassert write signal
#10 address = `DEST_ADDR;
    read = 1'b0;
    write = 1'b0;
    writedata = 32'h0e000000;

// Write source address: 11.0.0.0
#20 address = `SRC_ADDR;
    read = 1'b0;
    write = 1'b1;
    writedata = 32'h0b000000;

// Deassert write signal
#10 address = `SRC_ADDR;
    read = 1'b0;
    write = 1'b0;
    writedata = 32'h0b000000;

// Tunnel ID: 3, LSP ID: 4
#20 address = `TUNNEL_LSP_ID;
    read = 1'b0;
```

```
    write = 1'b1;
    writedata = 32'h00030004;

// Deassert write signal
#10 address = `TUNNEL_LSP_ID;
    read = 1'b0;
    write = 1'b0;
    writedata = 32'h00030004;

// Refresh time: 2 min (120000 ms)
#20 address = `REFRESH_DATA;
    read = 1'b0;
    write = 1'b1;
    writedata = 32'h0001d4c0;

// Deassert write signal
#10 address = `REFRESH_DATA;
    read = 1'b0;
    write = 1'b0;
    writedata = 32'h0001d4c0;

// Peak data rate: 10000 kb/s
#20 address = `PEAK_DATA_RATE_USER;
    read = 1'b0;
    write = 1'b1;
    writedata = 32'h00002710;

// Deassert write signal
#10 address = `PEAK_DATA_RATE_USER;
    read = 1'b0;
    write = 1'b0;
    writedata = 32'h00002710;

// Extend tunnel ID = 0
// Upstream port = 4
// Downstream port = 3
// TE DB port ID = 0
// Percentage = 0
#20 address = `PORT_PCT_DATA;
    read = 1'b0;
    write = 1'b1;
    writedata = 32'h00021800;

// Deassert write signal
#10 address = `PORT_PCT_DATA;
    read = 1'b0;
    write = 1'b0;
    writedata = 32'h00021800;

// Save LSP information
#20 address = `CONTROL_REGISTER;
    read = 1'b0;
    write = 1'b1;
    writedata = `SAVE_LSP_INFO;

// Deassert write signal
#10 address = `CONTROL_REGISTER;
```

```
    read = 1'b0;
    write = 1'b0;
    writedata = `SAVE_LSP_INFO;

// Wait for LSP information to be written
wait(waitrequest == 1'b0);

// End writing all information for LSP D

// Begin writing all information for LSP E
// Write destination address: 11.0.0.0
#20 address = `DEST_ADDR;
    read = 1'b0;
    write = 1'b1;
    writedata = 32'h0b000000;

// Deassert write signal
#10 address = `DEST_ADDR;
    read = 1'b0;
    write = 1'b0;
    writedata = 32'h0b000000;

// Write source address: 14.0.0.0
#20 address = `SRC_ADDR;
    read = 1'b0;
    write = 1'b1;
    writedata = 32'h0e000000;

// Deassert write signal
#10 address = `SRC_ADDR;
    read = 1'b0;
    write = 1'b0;
    writedata = 32'h0e000000;

// Tunnel ID: 4, LSP ID: 5
#20 address = `TUNNEL_LSP_ID;
    read = 1'b0;
    write = 1'b1;
    writedata = 32'h00040005;

// Deassert write signal
#10 address = `TUNNEL_LSP_ID;
    read = 1'b0;
    write = 1'b0;
    writedata = 32'h00040005;

// Refresh time: 12 min (720000 ms)
#20 address = `REFRESH_DATA;
    read = 1'b0;
    write = 1'b1;
    writedata = 32'h000afc80;

// Deassert write signal
#10 address = `REFRESH_DATA;
    read = 1'b0;
    write = 1'b0;
    writedata = 32'h000afc80;
```

```
// Peak data rate: 6000 kb/s
#20 address = `PEAK_DATA_RATE_USER;
    read = 1'b0;
    write = 1'b1;
    writedata = 32'h00001770;

// Deassert write signal
#10 address = `PEAK_DATA_RATE_USER;
    read = 1'b0;
    write = 1'b0;
    writedata = 32'h00001770;

// Extend tunnel ID = 1
// Upstream port = 3
// Downstream port = 4
// TE DB port ID = 0
// Percentage = 0
#20 address = `PORT_PCT_DATA;
    read = 1'b0;
    write = 1'b1;
    writedata = 32'h0009a000;

// Deassert write signal
#10 address = `PORT_PCT_DATA;
    read = 1'b0;
    write = 1'b0;
    writedata = 32'h0009a000;

// Save LSP information
#20 address = `CONTROL_REGISTER;
    read = 1'b0;
    write = 1'b1;
    writedata = `SAVE_LSP_INFO;

// Deassert write signal
#10 address = `CONTROL_REGISTER;
    read = 1'b0;
    write = 1'b0;
    writedata = `SAVE_LSP_INFO;

// Wait for LSP information to be written
wait(waitrequest == 1'b0);

// End writing all information for LSP E

// Begin writing all information for LSP F
// Write destination address: 11.0.0.0
#20 address = `DEST_ADDR;
    read = 1'b0;
    write = 1'b1;
    writedata = 32'h0b000000;

// Deassert write signal
#10 address = `DEST_ADDR;
    read = 1'b0;
    write = 1'b0;
```

```
    writedata = 32'h0b000000;

// Write source address: 14.0.0.0
#20 address = `SRC_ADDR;
    read = 1'b0;
    write = 1'b1;
    writedata = 32'h0e000000;

// Deassert write signal
#10 address = `SRC_ADDR;
    read = 1'b0;
    write = 1'b0;
    writedata = 32'h0e000000;

// Tunnel ID: 4, LSP ID: 6
#20 address = `TUNNEL_LSP_ID;
    read = 1'b0;
    write = 1'b1;
    writedata = 32'h00040006;

// Deassert write signal
#10 address = `TUNNEL_LSP_ID;
    read = 1'b0;
    write = 1'b0;
    writedata = 32'h00040006;

// Refresh time: 15 min (900000 ms)
#20 address = `REFRESH_DATA;
    read = 1'b0;
    write = 1'b1;
    writedata = 32'h000dbb80;

// Deassert write signal
#10 address = `REFRESH_DATA;
    read = 1'b0;
    write = 1'b0;
    writedata = 32'h000dbb80;

// Peak data rate: 8000 kb/s
#20 address = `PEAK_DATA_RATE_USER;
    read = 1'b0;
    write = 1'b1;
    writedata = 32'h00001f40;

// Deassert write signal
#10 address = `PEAK_DATA_RATE_USER;
    read = 1'b0;
    write = 1'b0;
    writedata = 32'h00001f40;

// Extend tunnel ID = 1
// Upstream port = 3
// Downstream port = 4
// TE DB port ID = 0
// Percentage = 0
#20 address = `PORT_PCT_DATA;
    read = 1'b0;
```

```
    write = 1'b1;
    writedata = 32'h0009a000;

// Deassert write signal
#10 address = `PORT_PCT_DATA;
    read = 1'b0;
    write = 1'b0;
    writedata = 32'h0009a000;

// Save LSP information
#20 address = `CONTROL_REGISTER;
    read = 1'b0;
    write = 1'b1;
    writedata = `SAVE_LSP_INFO;

// Deassert write signal
#10 address = `CONTROL_REGISTER;
    read = 1'b0;
    write = 1'b0;
    writedata = `SAVE_LSP_INFO;

// Wait for LSP information to be written
wait(waitrequest == 1'b0);

// End writing all information for LSP F

// ***** END WRITING LSP INFORMATION *****

// ***** BEGIN WRITING LINK INFORMATION *****

// Begin writing information for port 1

// Extend tunnel ID = 0
// Upstream port = 0
// Downstream port = 0
// TE DB port ID = 1
// Percentage = 80
#20 address = `PORT_PCT_DATA;
    read = 1'b0;
    write = 1'b1;
    writedata = 32'h000000d0;

// Deassert write signal
#10 address = `PORT_PCT_DATA;
    read = 1'b0;
    write = 1'b0;
    writedata = 32'h000000d0;

// Write next hop address: 10.0.0.0
#20 address = `NEXT_HOP_ADDR;
    read = 1'b0;
    write = 1'b1;
    writedata = 32'h0a000000;

// Deassert write signal
#10 address = `NEXT_HOP_ADDR;
    read = 1'b0;
```

```
    write = 1'b0;
    writedata = 32'h0a000000;

// Bandwidth: 50 Mb/s (50000 kb/s)
#20 address = `BANDWIDTH;
    read = 1'b0;
    write = 1'b1;
    writedata = 32'h0000c350;

// Deassert write signal
#10 address = `BANDWIDTH;
    read = 1'b0;
    write = 1'b0;
    writedata = 32'h0000c350;

// Port address: 15.0.0.1
#20 address = `PORT_ADDR;
    read = 1'b0;
    write = 1'b1;
    writedata = 32'h0f000001;

// Deassert write signal
#10 address = `PORT_ADDR;
    read = 1'b0;
    write = 1'b0;
    writedata = 32'h0f000001;

// Save TE DB information
#20 address = `CONTROL_REGISTER;
    read = 1'b0;
    write = 1'b1;
    writedata = `SAVE_TE_DB_INFO;

// Deassert write signal
#10 address = `CONTROL_REGISTER;
    read = 1'b0;
    write = 1'b0;
    writedata = `SAVE_TE_DB_INFO;

// Wait for TE DB information to be written
wait(waitrequest == 1'b0);

// End writing information for port 1

// Begin writing information for port 2
// Extend tunnel ID = 0
// Upstream port = 0
// Downstream port = 0
// TE DB port ID = 2
// Percentage = 50
#20 address = `PORT_PCT_DATA;
    read = 1'b0;
    write = 1'b1;
    writedata = 32'h00000132;

// Deassert write signal
#10 address = `PORT_PCT_DATA;
```

```
    read = 1'b0;
    write = 1'b0;
    writedata = 32'h00000132;

// Write next hop address: 16.0.0.1
#20 address = `NEXT_HOP_ADDR;
    read = 1'b0;
    write = 1'b1;
    writedata = 32'h10000001;

// Deassert write signal
#10 address = `NEXT_HOP_ADDR;
    read = 1'b0;
    write = 1'b0;
    writedata = 32'h10000001;

// Bandwidth: 100 Mb/s (100000 kb/s)
#20 address = `BANDWIDTH;
    read = 1'b0;
    write = 1'b1;
    writedata = 32'h000186a0;

// Deassert write signal
#10 address = `BANDWIDTH;
    read = 1'b0;
    write = 1'b0;
    writedata = 32'h000186a0;

// Port address: 15.0.0.2
#20 address = `PORT_ADDR;
    read = 1'b0;
    write = 1'b1;
    writedata = 32'h0f000002;

// Deassert write signal
#10 address = `PORT_ADDR;
    read = 1'b0;
    write = 1'b0;
    writedata = 32'h0f000002;

// Save TE DB information
#20 address = `CONTROL_REGISTER;
    read = 1'b0;
    write = 1'b1;
    writedata = `SAVE_TE_DB_INFO;

// Deassert write signal
#10 address = `CONTROL_REGISTER;
    read = 1'b0;
    write = 1'b0;
    writedata = `SAVE_TE_DB_INFO;

// Wait for TE DB information to be written
wait(waitrequest == 1'b0);

// End writing information for port 2
```

```
// Begin writing information for port 3
// Extend tunnel ID = 0
// Upstream port = 0
// Downstream port = 0
// TE DB port ID = 3
// Percentage = 50
#20 address = `PORT_PCT_DATA;
    read = 1'b0;
    write = 1'b1;
    writedata = 32'h000001b2;

// Deassert write signal
#10 address = `PORT_PCT_DATA;
    read = 1'b0;
    write = 1'b0;
    writedata = 32'h000001b2;

// Write next hop address: 17.0.0.2
#20 address = `NEXT_HOP_ADDR;
    read = 1'b0;
    write = 1'b1;
    writedata = 32'h11000002;

// Deassert write signal
#10 address = `NEXT_HOP_ADDR;
    read = 1'b0;
    write = 1'b0;
    writedata = 32'h11000002;

// Bandwidth: 60000 kb/s
#20 address = `BANDWIDTH;
    read = 1'b0;
    write = 1'b1;
    writedata = 32'h0000ea60;

// Deassert write signal
#10 address = `BANDWIDTH;
    read = 1'b0;
    write = 1'b0;
    writedata = 32'h0000ea60;

// Port address: 15.0.0.4
#20 address = `PORT_ADDR;
    read = 1'b0;
    write = 1'b1;
    writedata = 32'h0f000004;

// Deassert write signal
#10 address = `PORT_ADDR;
    read = 1'b0;
    write = 1'b0;
    writedata = 32'h0f000004;

// Save TE DB information
#20 address = `CONTROL_REGISTER;
    read = 1'b0;
    write = 1'b1;
```

```
    writedata = `SAVE_TE_DB_INFO;

// Deassert write signal
#10 address = `CONTROL_REGISTER;
    read = 1'b0;
    write = 1'b0;
    writedata = `SAVE_TE_DB_INFO;

// Wait for TE DB information to be written
wait(waitrequest == 1'b0);

// End writing information for port 3

// Begin writing information for port 4
// Save information for port 4
// Extend tunnel ID = 0
// Upstream port = 0
// Downstream port = 0
// TE DB port ID = 4
// Percentage = 75
#20 address = `PORT_PCT_DATA;
    read = 1'b0;
    write = 1'b1;
    writedata = 32'h0000024B;

// Deassert write signal
#10 address = `PORT_PCT_DATA;
    read = 1'b0;
    write = 1'b0;
    writedata = 32'h0000024B;

// Write next hop address: 11.0.0.0
#20 address = `NEXT_HOP_ADDR;
    read = 1'b0;
    write = 1'b1;
    writedata = 32'h0b000000;

// Deassert write signal
#10 address = `NEXT_HOP_ADDR;
    read = 1'b0;
    write = 1'b0;
    writedata = 32'h0b000000;

// Bandwidth: 40 Mb/s (40000 kb/s)
#20 address = `BANDWIDTH;
    read = 1'b0;
    write = 1'b1;
    writedata = 32'h00009c40;

// Deassert write signal
#10 address = `BANDWIDTH;
    read = 1'b0;
    write = 1'b0;
    writedata = 32'h00009c40;

// Port address: 15.0.0.3
#20 address = `PORT_ADDR;
```

```
        read = 1'b0;
        write = 1'b1;
        writedata = 32'h0f000003;

    // Deassert write signal
    #10 address = `PORT_ADDR;
        read = 1'b0;
        write = 1'b0;
        writedata = 32'h0f000003;

    // Save TE DB information
    #20 address = `CONTROL_REGISTER;
        read = 1'b0;
        write = 1'b1;
        writedata = `SAVE_TE_DB_INFO;

    // Deassert write signal
    #10 address = `CONTROL_REGISTER;
        read = 1'b0;
        write = 1'b0;
        writedata = `SAVE_TE_DB_INFO;

    // Wait for TE DB information to be written
    wait(waitrequest == 1'b0);

    // End writing information for port 4

    // ***** END WRITING LINK INFORMATION *****

    // ***** END WRITING LSP AND LINK INFORMATION *****

    // ***** BEGIN ESTABLISHING LSPs *****

    // Establish LSPs
    #20 address = `CONTROL_REGISTER;
        read = 1'b0;
        write = 1'b1;
        writedata = `ESTABLISH_LSPS;

    // Deassert write signal
    #10 address = `CONTROL_REGISTER;
        read = 1'b0;
        write = 1'b0;
        writedata = `ESTABLISH_LSPS;

    // Wait for packet to establish LSPs to be generated
    wait(waitrequest == 1'b0);
end
endmodule
```

References

[1] Alcatel Corporation, "The Alcatel MPLS Solution: A Key Technology to Enable Profitable Next Generation Networks", 2003.

[2] Alcatel Corporation, "Making IP/MPLS Operations Plug-and-Play with Alcatel Service Access Manager (SAM)", 2004.

[3] Altera Corporation, "Application Note 132: Implementing Multiprotocol Switching with Altera PLDs", version 1.0, January 2001.

[4] Altera Corporation, "Avalon Interface Specification", April 2005.

[5] Altera Corporation, "Introduction to Megafunctions", version 1, January 1998.

[6] Altera Corporation, "Nios Development Board Reference Manual, Stratix Professional Edition", September 2004.

[7] C. D'Souza, "MPLS Traffic Engineering with Avici Converged Core Routers", Avici Systems, 2004.

[8] Cisco Systems Inc., "Ethernet over MPLS for the Cisco 7600 Series Internet Routers", 2004.

[9] D. Awduche, L. Berger, D. Gan, T. Li, V. Srinivasan, G. Swallow, "RFC 3209: RSVP-TE: Extensions to RSVP for LSP Tunnels", December 2001.

[10] D. Gluskin and I. Cidon, "A Hardware Signaling Paradigm for Fine-Grained Resource Reservation", Technion, CCIT Publication # 433, June 2003, based on D. Gluskin "Hardware Oriented Resource reservation scheme for integrated networks", Technion EE Master Thesis, March 2001.

[11] D. Oran, "RFC 1142: OSI IS-IS Intra-domain Routing Protocol", February 1990.

[12] E. Rosen, A. Viswanathan, R. Callon, "RFC 3031: Multiprotocol Label Switching Architecture", January 2001.

[13] E. Rosen, D. Tappan, G. Fedorkow, Y. Rekhter, D. Farinacci, T. Li, A. Conta, "RFC 3032: MPLS Label Stack Encoding", January 2001.

[14] E. Rosen, Y. Rekhter, "RFC 4364: BGP/MPLS IP Virtual Private Networks (VPNs)", February 2006.

[15] F. Vahid, T. Givaris, "Embedded System Design: A Unified Hardware/Software Introduction", John Wiley and Sons Inc., 2002.

[16] Foundry Networks, "Offering Scalable Layer 2 Services with VPLS and VLL", 2006.

[17] G. Hågård, M, Wolf, "Multiprotocol label switching in ATM networks", Ericsson Mobile Communications, 1998.

[18] G. Pildish, "Cisco ATM Solutions: Master ATM implementation on Cisco networks", Cisco Press, 2000.

[19] Hammerhead Systems, "HSX 6000 Layer 2.5 Aggregation Switch", 2006.

[20] H. Wang, M. Veeraraghavan, R. Karri, "A Hardware Implementation of a Signaling Protocol", Opticomm 2002, July 2002.

[21] H. Wang, M. Veeraraghavan, R. Karri, T. Li, "A Hardware-Accelerated Implementation of the RSVP-TE Signaling Protocol", 2004 IEEE International Conference on Communications, Volume 3, 20-24 June 2004 Page(s):1609 – 1614.

[22] J. Ash, Y. Lee, P. Ashwood-Smith, B. Jamoussi, D. Fedyk, D. Skalecki, L. Li, "RFC 3214: LSP Modification Using CR-LDP", January 2002.

[23] J. Chin, "Cisco Frame Relay Solutions Guide", Cisco Press, 2004.

[24] J. Guichard, I. Pepelnjak, "MPLS and VPN Architectures", Cisco Press, 2000.

[25] J. Moy, "RFC 2328: OSPF Version 2", April 1998.

[26] J. Wroclawski, "RFC 2210: The Use of RSVP with IETF Integrated Services", September 1997.

[27] L. Andersson, P. Doolan, N. Feldman, A. Fredette, B. Thomas, "RFC 3036: LDP Specification", January 2001.

[28] L. Sha, "System Architecture and Hardware Implementations for a Reconfigurable MPLS Router", University of Saskatchewan Thesis for Master of Science in the Department of Electrical Engineering, August 2003.

[29] M. Abou-Gabal, R. Peterkin, D. Ionescu "An Architecture for a Hardware Implementation of the OSPF Protocol", CAINE 2004 - 17th International Conference on Computer Applications in Industry and Engineering, Orlando, Florida, USA, November 17-19, 2004.

[30] M. Abou-Gabal, "Architecture for a Hardware Implementation of the OSPF Protocol", University of Ottawa Thesis for Master of Applied Science in Electrical Engineering.

[31] M. Abou-Gabal, R. Peterkin, D. Ionescu: "IS-IS protocol Hardware Architecture for VPN solutions", in Proceedings of the 7th WSEAS International Conference on Communications, Athens, Greece, July 12-15, 2004.

[32] M. Anderson, A. Donnell, "Testing Juniper Networks M40 Router MPLS Interoperability with Cisco Systems 7513 and 12008 Routers", Juniper Networks Inc., 2000.

[33] Mangrove Systems, "Piranha 100: Intelligent Access to the Packet Core", 2005.

[34] Nortel Networks Inc., "Multiservice Provider Edge (MPE) 9000 portfolio", 2005.

[35] P. Agarwal, B. Akyol, "RFC 3443: Time To Live (TTL) Processing in Multi-Protocol Label Switching (MPLS) Networks", January 2003.

[36] P. Pan, Ed., G. Swallow, Ed., A. Atlas, Ed., "RFC 4090: Fast Reroute Extensions to RSVP-TE for LSP Tunnels", May 2005.

[37] R. Braden, Ed., L. Zhang, S. Berson, S. Herzog, S. Jamin, "RFC 2205: Resource ReSerVation Protocol (RSVP) -- Version 1 Functional Specification", September 1997.

[38] R. Venkateswaran, "Virtual Private Networks," IEEE Potentials, Mar. 2001.

[39] S. Blake, D. Black, M. Carlson, E. Davies, Z. Wang, W. Weiss, "RFC 2475: An Architecture for Differentiated Service", December 1998.

[40] S. Lee, "Design of Computers and Other Complex Digital Devices", Prentice Hall, November 1999.

[41] S. K. Long, R. R. Pillai, J. Biswas, T. C. Khong, "Call Performance Studies on the ATM Forum UNI Signalling", http://www.krdl.org.sg/Research/Publications/ Papers/pillai_uni_perf.pdf

[42] S. Kent, K. Seo, "RFC 4301: Security Architecture for the Internet Protocol", December 2005.

[43] S. Shenker, C. Partridge, R. Guerin, "RFC 2212: Specification of Guaranteed Quality of Service", September 1997.

[44] S. Shenker, J. Wroclawski, "RFC 2215: General Characterization Parameters for Integrated Service Network Elements", September 1997.

[45] T. Li, Z. Tao, H. Wang, M. Veeraraghavan, "Specification of a Subset of CR-LDP for Hardware Implementation", January 2005.

[46] T. Stern, K. Bala, "Multiwavelength Optical Networks: A Layered Approach", Addison-Wesley, 1999.

[47] V. Alwayn, "Advanced MPLS Design and Implementation", Cisco Systems, 2002.

[48] W. Townsley, A. Valencia, A. Rubens, G. Pall, G. Zorn, B. Palter, "RFC 2661: Layer Two Tunneling Protocol L2TP", August 1999.

[49] Y. Rekhter, T. Li, "RFC 1654: A Border Gateway Protocol 4 (BGP-4)", July 1994.