DESIGN AND ANALYSIS OF AN ATM NETWORK TRAFFIC SECURITY DEVICE

A Thesis

by

DAN CRISTIAN TEODOR

Submitted to the Office of Graduate Studies of
Texas A&M University
in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

August 1997

Major Subject: Computer Science

DESIGN AND ANALYSIS OF AN ATM NETWORK TRAFFIC SECURITY DEVICE

A Thesis

by

DAN CRISTIAN TEODOR

Submitted to Texas A&M University
in partial fulfillment of the requirements
for the degree of

MASTER OF SCIENCE

Approved as to style and content by:

[illegible]	[illegible]
Wei Zhao (Chair of Committee)	Pierce Cantrell (Member)
[illegible]	[illegible]
William Lively (Member)	Richard A. Volz (Head of Department)

August 1997

Major Subject: Computer Science

ABSTRACT

Design and Analysis of an ATM Network Traffic Security Device.
(August 1997)
Dan Cristian Teodor, B.S., State University of New York at Buffalo
Chair of Advisory Committee: Dr. Wei Zhao

Wide access distributed area network services are increasing in range and capacity at an exponential rate. With the continuation of this growth, the requirements of providing uniform security management will become more and more difficult to manage without occupying a significant portion of the network traffic capability available to the end-users the network is intended to service. Current methods rely on the network architecture itself to provide the mechanisms by which traffic is monitored and, when the situation warrants, suppressed in order to ensure that security methods are enforced. With the introduction of ATM/SONET technologies into this arena, the possibility of integrating every class of information service into a common transmission framework comes closer to reality through its high bandwidth capability and very large scalability. However, this expansion of types of services available and range offered complicates the task of minimizing the possibility that unauthorized persons may rely on covert traffic creation and reception in order to use the network in a manner not permitted by its controlling bodies.

To address this deficiency, this thesis presents the groundwork for the implementation of a dedicated security framework which should be able to accomplish the task of minimizing the potential for covert channels in such networks without creating the associated traffic overhead normally associated with such operations within the network itself. For this security framework, the system described presents a design which incorporates both the mechanisms for the detection and suppression of covert traffic, as well as, the implementation by which these mechanisms may be linked to a unifying control authority.

Performance analyses of the design show that it may be feasibly implemented with current levels of semiconductor manufacturing technology and incorporates elements that are readily available on the market. Secondly, these analyses show that the associated response delay experienced by transiting network traffic is minimal with respect to the overall time the information spends while en route through the network. Thirdly, the delays associated with connection management are constant under all global traffic conditions. Finally, the design is shown to incur no overhead in excess network traffic due to the enforcement functions which it implements.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER I

INTRODUCTION

As the size of wide-area public access backbones increases, so does the complexity of the task of monitoring and enforcing security standards across the entire range of services provided by these communications services. Current methods of providing a uniform framework for security enforcement across the network attempt to use the network itself as a tool for performing the necessary management functions. Their aim is to provide this uniform enforcement while providing the highest level of accessibility and responsiveness for which the network is equipped. The contributions of this thesis are as follows:

- Presentation of a framework within which uniform and rapid security enforcement may be provided while still offering the greatest amount of network bandwidth with as fast a response time as possible to the end user. This is presented in Chapters II and III.
- A complete design for the implementation of this security framework which is implementable using current semiconductor manufacturing technologies and relies on widely available and cost-effective components. This design is intended to minimize the network response degradation, as well as, the overhead incurred by the administrative functions of transferring management data. This is presented in Chapters IV and VI.
- A simulated implementation of the design which would allow for verification of correct operation of the device. Additionally, the simulation would allow the researcher to evaluate the creative process used in order to add future optimizations or expand the functionality of the device as component availabilities and technologies evolve. This is presented in Chapter V.

The main focus of the work for this thesis has been on creating a design suitable for use in evaluating the efficacy of a distributed security enforcement method for wide area ATM networks. The implementation is intended as a starting point for future expansion of the security enforcement capacity of a dedicated supervisory network for wide-area ATM networks.

The journal model is *IEEE Transactions on Computers*.

CHAPTER II

ISSUES IN ATM NETWORK SECURITY

At its very core, the ATM standard defines a wide-bandwidth network as an interconnection topology of digital switches that transport data between themselves in small packets of data called cells. Each cell consists of a 48 byte payload and a 5 byte header [1]. In order for a cell to be communicated from one computer to another it must first be transmitted to the nearest switch in the ATM network to the source computer. This switch then uses local information to send this packet to another switch in the ATM network. This process is repeated until the cell is transmitted to the switch closest to the destination computer and that switch, in turn, delivers the cell to that destination. The ATM network switches rely on a store-and-forward method of transmitting to another switch in the ATM network to which it has access. A switch to which another switch has access in an ATM network is defined to be one to which the second switch has a direct physical connection (by whatever physical medium chosen).

Switch to switch routing in an ATM network is performed based on local information at every switch in which a given cell arrives. This local information is stored in a look-up table which consists of *virtual path identifiers* (VPI) and *virtual connection identifiers* (VCI) [1]. These two pieces of information define to which switch the data should be forwarded on its next hop, based on the VPI/VCI information in the header of the arriving cell. When such a cell arrives, its VPI/VCI data is found in the switch's internal data and, based on this, the cell's header is updated and queued at the output that leads to the ATM switch that forms the next link along the logical path specified by the VPI/VCI pair.

At present, almost all of the available security enforcement methods for these network architectures involve software solutions in the switches and the network management functions found in the industry specifications. These software methods perform a series of transformations and tests on cells transiting through the network to ensure that they fulfill certain characteristics [2-16]. These software enforcement methods are designed to reside either in the ATM switches themselves or on the source / destination machines where the data originates or arrives.

Currently, there are two distinct approaches to network security enforcement. The first approach proactively attempts to prevent an intruder from ever gaining access to the system while the second approach reactively attempts to detect and track an intruder after they have gained unauthorized access.

The proactive approach attempts to solve the problem through five different methods:

- *Access Control* refers to a mechanism that restricts access to various subsystems only to those parties that are able to provide a key known only to the subsystem and the set of authorized users. It is the responsibility of the subsystem to allow only those transactions that are accompanied by an authorized key and, likewise, the responsibility of the users not to allow their keys to pass on to unauthorized parties. These approaches have been proven effective many times and are currently in use in most major operating systems. By using careful management of password keys, it has been shown that access control mechanisms may be effectively implemented in large distributed environments.

- *Encryption* requires all parties participating in communication across the network to use a well-known algorithm to hide their plain text data according to a key known by all parties and to unhide this data according to a key known only by themselves. This security method requires the communicating parties to be within a domain that is considered to be secure and it also implies the existence of a globally centralized key management authority that also exists within a secure domain. The only area that is assumed to be non-secure is the wide area network or inter-network itself. Overhead is generated within these systems in order to implement covert-channel free mechanisms in order to manage and transfer keys while, at the same time, ensuring the security of the centralized authority [3, 16]. However, through careful selection of an encryption algorithm and keys of reasonable length it is possible to apply encryption for data transfer in large distributed systems.

- *Physical Lockout* simply describes the physical infrastructure that protects the various hardware components of a distributed system and only allows authorized parties to access the various hardware terminals such as display terminals, keyboards and assorted input devices. Since there is no global information migration required to maintain the locks and gates to the access terminals this approach does not present any significant obstacles to its use for guarding distributed systems.

- *Neutral transmission patterns and modal operation* relies on a combination of generating traffic padding with meaningless packets, conditionally rerouting segments of traffic and controlling the creation and destruction of connections within the network. The objective of this method of security enforcement relies on the masking of actual traffic in order to reduce the possibility that an attacker will gain any useful information through the passive monitoring of network links [17, 18]. Here again, the end-user workstations are assumed to exist within a secure domain while the network or inter-network is the only component that is not secure. Overhead is generated both by padding traffic and by the extra work done by using inefficient routes to pass data. In addition, significant restrictions are generally placed on the capacity of client workstations to create connections with one another in order to perform meaningful work [19].

The reactive approach attempts collect information regarding system activities and analyze this information in order to establish patterns for "typical" system use. Once these patterns are established, the system then restricts the access of those users that initiate activities of a type or with a frequency that deviates from those calculated norms of "typical" system use [5-14]. The major problems encountered in attempting to design such systems have come from three sources. First, the quantity of information generated by tracking every system activity in a large system is orders of magnitude too large to store and not significantly impact system performance. Second, the computation time required to analyze and create system profiles can become significant, which can lead to a poor reaction time when attempting to detect users performing activities that do not fit the calculated system profile. Third, malicious users may attempt to fool these systems by performing activities that differ from the "typical" profile by a small amount. Through continued activity of this type, new system norms would be established that account for the patterns generated by the malicious activity. These systems may be categorized into two classes which differ by the method used to calculate their system profiles:

- *Heuristic profiling and expert systems* rely on the use of a system's past experience to create heuristics by which to judge the capability of a particular user to be an intruder. These heuristics can be either a set of learned rules based on a history of system security breakdowns or a pre-defined set of rules created by a system administrator. In either case, this class of intruder detection systems require large amounts of resources (both storage and computational) in order to track the necessary heuristics and to test all heuristics continually against the current system state. Lunt [13] showed that it is possible to accomplish this task, but only at tremendous expenditure of system resources of a type that would only be available on a large workstation.

- *Statistical profiling* makes the simplifying assumption that an intruder's activities should be detectable by only monitoring the statistical averages of various types of system activity. Therefore, intruders should be detectable if any given subset of the system's monitored activities deviate significantly from their normal profile. In this way, it is possible to eliminate the need to track a large set of heuristics and the computation time required is now bounded only by the number of system statistics which are tracked.

Out of all of the individual methods described, some do not lend themselves well to implementation in the distributed environment of a large number of semi-dumb switching stations which is envisioned for the current and future implementation of ATM networks. Those methods which rely heavily on being aware of the current global state of the distributed system are not suitable in an ATM environment since network bandwidth must be expended in order to maintain the accuracy of that information. Modal operation and neutral transmission patterns have been shown to require a large amount of information about the global state of the distributed system and, as such, their performance will degrade as networks become larger in size. Also, methods which require a large amount of memory or computation will also be difficult to implement effectively in the world of ATM networks since the necessary resources will need to be replicated many times over as the number of nodes in these networks grow. Intruder detection through heuristic profiling have been shown to require the resources provided by an entire workstation in order to provide timely information and, thus, would be difficult to scale to the number of nodes that would be required by a large network.

Of the methods discussed, intruder prevention through access control, data encryption and physical lockout show the most promise for implementation in a wide-area ATM environment. For intruder detection, statistical profiling has a definite performance advantage in a wide-area ATM environment. Since the capacity and range of services offered by global networks continues to grow at an astonishing pace the performance and cost advantages of these methods can only increase in the future [20, 21].

Data encryption, access control and statistical profiling all rely on the proper operation of a software module within the nodes of the ATM network. No matter how efficient or adaptive these approaches are, they all, as a whole, are susceptible to unfriendly attack by other software systems which may be connected to the network and which mimic the behavior of network nodes assigned to network management tasks [22]. Such attacks include the possibility of modifying virtual connection and path data in a switch in such a way that it is beneficial to the attacker (i.e. diversion or insertion of traffic in an unauthorized manner). Also, with the proliferation of inter-networking technologies, it becomes more and more difficult for an administrative body to manage and patrol the traffic of every node on the network due to the very large size and distributed nature of these inter-networks. Those software security methods which reside within the network nodes themselves are prone to uneven enforcement since every organization that controls individual machines connected to the inter-network apply their own standards and methods of security enforcement. Thus, an attacker may be able to use a combination of partial weaknesses that exist within the security management of different nodes across the network in order to perform unauthorized operations.

Therefore, in order to reach a more complete state of enforcement, the ATM network should not rely on the ATM switches or client computer nodes to perform these functions. A separate hardware entity that is controlled by one management authority, devoted solely to the purpose of security management and to which no network user or local manager has direct access would be the most appropriate solution. This centralized authority would be tasked with the responsibility of ensuring that all network traffic entering or exiting the secured backbone belongs to a virtual connection through that backbone that has been registered with the central securing authority. Thus, an attacker would be unable to use the network to carry traffic that has not been explicitly registered with the centralized authority body. Further, this supervisory network would be responsible for verifying not only the correlation of traffic with an existing, registered virtual connection, but also that this traffic is not exceeding any volume bounds placed on that connection. In this way, an attacker would be restricted from using an existing, authorized connection on which to piggyback covert traffic.

Aside from the assumed secure network on which the authorizing body would rely, hardware modules would be required at each entry and exit point into and from the backbone which would perform the actual monitoring of backbone traffic. This monitoring would be performed based on directives from the supervising authority body. Toward this goal, the design and implementation of application specific hardware has already been shown to be a cost-effective method of realizing such a security governance structure [23, 24]. A next step in providing ATM network security in a cost-effective manner is to encapsulate access control, encryption and statistical profiling for network traffic into application specific hardware and which will reside in the hardware modules at the entry and exit of every access path into the secure backbone.

CHAPTER III

SECURITY DEVICE SPECIFICATION

The ATM forum specifies two communication protocols by which cells are to be transferred across an ATM network. The first specification is called the Network to Network Interface (NNI) and describes the data formats to be used when two ATM switches in a public network communicate with one another [25]. The second of these is called the User to Network Interface (UNI) and describes the data formats to be used when communicating between a public service ATM switch and a private network ATM switch or between two private network ATM switches. Therefore, any given connection in an ATM inter-network forwards data according to the following sequence:

1. The data is relayed from the source computer to the first switch in the source private (local / organizational) ATM network using UNI.
2. The data is relayed from switch to switch within the source private ATM network using UNI.
3. The data is relayed from the last switch in the private ATM network to the first switch in the public ATM network using UNI.
4. The data is relayed from switch to switch within the public ATM network using NNI.
5. The data is relayed from the last switch in the public ATM network to the first switch in the destination private ATM network using UNI.
6. The data is relayed from switch to switch in the destination private ATM network using UNI.
7. The data is relayed from the last switch in the destination private ATM network to the destination node (computer) using UNI.

From this sequence of events it is possible to conclude that the majority of steps in the transmission rely on the UNI interface to transfer data. Further, since there are no user nodes (computers) connected directly to the public ATM network and, if we can ensure that no covert traffic exists among the nodes that communicate through the UNI, then we can also guarantee that all traffic in the public ATM network will also be covert element free. Therefore, the first specification of the external security device is that it correctly implements the data elements of the UNI in its network interfaces.

It is also necessary to address the method by which virtual connection information is maintained inside of each ATM switch for the purposes of routing information. The currently accepted method involves the transmission of specialized cells that contain "management data". These cells are originated by user nodes on the ATM inter-network for the purpose of setting up new connections. They inform the switches to which they are transmitted that a new connection is desired through that switch and that the switch should allocate a unique VPI/VCI pair in their internal data tables for that connection. Since it is this very method of new / existing connection management that is in question with regard to the detection of covert traffic, the external security device must rely on some other communication device that is external to the ATM network to acquire information about new connections as they are created within the network.

The design of this security device is intended to be applied to the current state of ATM network specifications. Therefore, the device should support placement within networks that utilize all of the currently published physical interface standards. In order to keep this requirement within reasonable bounds, those physical layer standards that are developed by any one organization and, therefore, considered "proprietary" will not be considered for support. Instead, those standard that were written to be "industry wide" and, supposedly, do not favor technologies controlled by any one specific manufacturer will be the basis for physical layer support in the design of this device. These standards are those physical layer interface specifications published by the ATM forum.

Currently, there are five standards published and officially recognized by the ATM Forum. These are (in order of increasing data rates):

- DS-1 (1.544 Mb/sec) physical interface specification [26]
- 25.6 Mb/sec over twisted pair physical interface specification [27]
- DS-3 (44.21 Mb/sec) physical interface specification [28]
- 155 Mb/sec over twisted pair physical interface specification [29]
- 622.08 Mb/sec Synchronous Optical Network (SONET) physical interface specification [30]

The device must perform the functions of detection, suppression and alert, when illegal traffic is found to be passing through the network, in a timely manner. Detection refers to determining if a cell being transmitted out of a particular port on a specific switch is in accordance with a VPI/VCI pair defined to be valid traffic for that switch's output. Detection also, optionally, involves verification if that cell is in accordance with a valid VPI/VCI pair but violates the traffic capacity of that channel. Suppression involves the discarding of the offending cell and alert refers to a method by which the security device reports the VPI/VCI pair of the offending cell and the switch output which produced it. Optionally, alert also refers to the reporting of the reason for which the cell is found to be in violation, whether it be due to an illegal VPI/VCI pair or due to a connection capacity violation. The issue of performing these functions in a timely manner is best described by setting a target of reporting a traffic infraction within one cell transmission time on the physical media of that network, regardless of what the transmission bandwidth may be.

The device must be able to perform effectively under periods of peak network traffic without hampering or significantly delaying the operation of the network itself. This means that, when a particular switch output is generating cells at its maximum rate for a sustained period of time, the device must be able to correctly process and retransmit those cells which are not found to cause any type of violation within a bounded delay of no more than one cell time.

Therefore, the device must conform to the following specifications:

- Support the ATM forum UNI data specification.
- Provide an external interface through which to report network traffic violations.
- Support all the physical interface specifications currently recognized by the ATM forum.
- Perform covert network traffic detection, suppression and alert.
- Perform its intended functions in a timely fashion even under peak traffic conditions.
- Be designed in such a fashion that its implementation is both cost effective and stable.

CHAPTER IV

SECURITY DEVICE DESIGN

IV.A Design Overview

The determining factor in the design was the need to implement the device with components that are widely available, inexpensive and of a proven stability. Because of the high data rates involved in the transmission of cells in ATM networks, it was necessary to use as much parallelization of functions as possible in hardware in order to implement the design with standard components and realizable clock speeds.

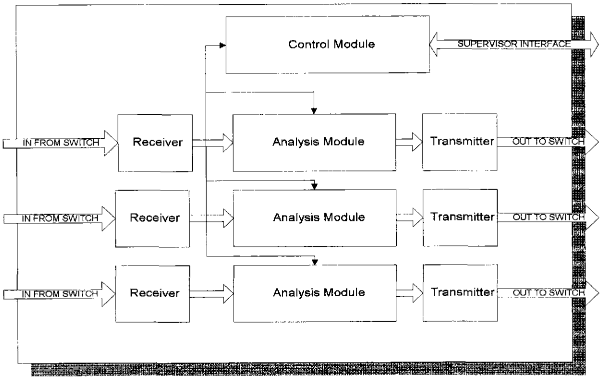


Fig. 1. Block diagram of ATM switch security device

As shown in figure 1, the device relies on three units functioning in tandem to handle the traffic produced by each ATM network switch output. These three units, labeled Receiver, Analysis Module and Transmitter, function in sequence to capture, analyze and retransmit the network traffic from one ATM network switch output port. The Receivers queue the incoming data from the ATM network switch and present the data to the Analysis Modules in manageable pieces. The Analysis Modules capture the data from the Receivers and perform the necessary functions of detection, suppression and alert and pass this data to the Transmitters if it is found to be valid. The Transmitters capture the outgoing data from the Analysis Modules and transmit it to the subsequent switch in the ATM network.

Overseeing the operation of the Receivers, Analysis Modules and Transmitters is the Control Module. It is the responsibility of this module to accept data from the Supervisory Interface regarding new connections that need to be admitted in the ATM network and pass this data to the appropriate Analysis Module. Additionally, the Control Module must detect a traffic alert from any one of the Analysis Modules and, when it occurs, must capture the data regarding the cell which caused the alert from the appropriate Analysis Module. Then, the Control Module must transmit this data to the supervisory interface.

When all of these units function correctly, the end result will be a device that can capture, analyze and retransmit the ATM network traffic on the multiple output ports of an ATM switch, update path information and report traffic infractions under conditions of peak data rate transmission. The analysis portion of the device's function may be of two types. Under the first variant, arriving network traffic will be checked for validity in terms of whether or not the connection with which that traffic is associated does indeed pass through the network switch and port from which the data originated. The second variant will perform exactly the same verification as the first variant and, in addition, will also verify that traffic that has been found to be traveling across a valid connection has not exceeded the traffic limits placed on that connection. The design of both variants is presented.

IV.B Transmitter and Receiver Design

The receivers and transmitters capture and send the cell data from and to the physical outputs and inputs of the ATM switches between which the device lies and process it according to the particular physical interface characteristics of those switches. This includes any functions of decryption, decompression and bit-level synchronization. The exact design of these units will be highly physical media dependent and beyond the scope of this design description. The physical blocks comprising these modules is not a matter of choice since

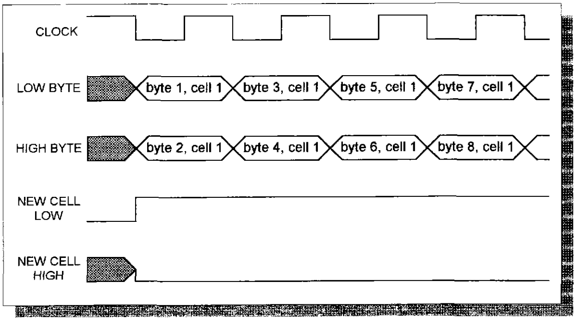


Fig. 2. Timing diagram for new cell arrival on low byte of Receiver output

it is already described in the ATM forum literature [26-30] and components for use in these modules have already been implemented as prototypes [24]. The only design issue which needs to be noted with regard to the function of the receivers and transmitters are that they present data to the Analysis Modules in parallel sixteen-bit words and synchronize the presentation of these 16-bit words to the Analysis Module clock. Receivers use two control lines, with one conductor each, carrying a digital signal, to indicate each of the following two conditions:

1. (*New Cell Low*) If asserted high on the rising edge of the Analysis Module's clock, it indicates that the data on the low-order eight bits of the outputs of the receiver is the first byte in a newly arriving cell.
2. (*New Cell High*) If asserted high on the rising edge of the Analysis Module's clock, it indicates that the data on the high-order eight bits of the outputs of the receiver is the first byte in a newly arriving cell. This will occur only when a cell arrives immediately after its predecessor. If this is not the case, then the Transmitter will present its data with the first byte on the lower eight bits of its outputs and use the signal *New Cell Low* to inform the Analysis Module of this status.

The graphical representation of the timing characteristics of these interface signals is shown in figure 2 and in figure 3.

In turn, the Analysis Modules use the same two one-conductor, digital signals to inform the transmitters of these same conditions in order to pass a cell which has been found to be valid to the Transmitter at the rate of one sixteen-bit word per Analysis Module clock cycle. The implication is that the output stage of the Receiver and the input stage of the Transmitter must be synchronized to the same clock as the Analysis Module.

The Receivers will present their data words and assert their control signals on the falling edge of the clock cycle within which the data arrives in order to allow the Analysis Modules to use positive edge triggered logic to sample this data. The same is true for the Transmitters which will capture the data being sent out by the Analysis Modules on the falling edge of the clock.

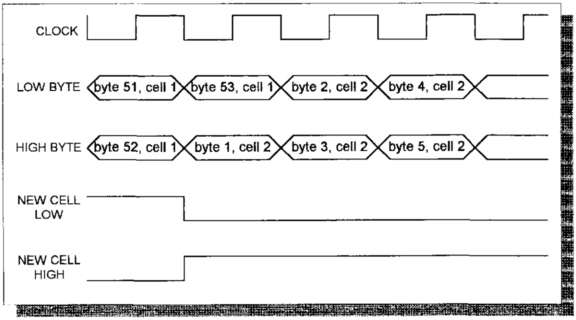


Fig. 3. Timing diagram for new cell arrival on high byte of Receiver output

The stipulation that data be presented to and read from the Analysis Modules in sixteen-bit words arises out of the need to have this device operate at clock speeds that are reasonable for implementation in integrated circuit designs that utilize the major logic families currently available. At the highest speed scenarios of data rates of 622.08 Mbps, it implies that 38.880 million sixteen-bit words will need to be processed by every Analysis Module which, in turn, implies a maximum clock rate of 38.880 MHz for the Analysis Modules.

IV.C Analysis Module Design

The Analysis Module will admit a new cell into a 16-bit shift register, word by word from the receiver. In parallel, as components of the VPI/VCI pair belonging to the cell in transit are received from the Receiver (contained in the cell header, consisting of the first five bytes of data) they will also be copied into six 4-bit latches. This transfer will occur in a stepwise fashion over the course of more than one clock cycle since different portions of the ATM cell header become visible at the Receiver's outputs on different 16-bit words. Multiplexers will be used to select which words of the header will be loaded into these latches based on whether the arriving cell entered the Analysis Module with its first byte in the low or the high order eight bits of the register input from the receiver.

Once all 24 bits of the VPI/VCI pair associated with the cell in transit have been captured in these 4-bit latches, the twenty four bits of output from them will be presented to the memory lookup module in two 12-bit words, with one word being presented at a time. The control to present these two 12-bit words will be performed by a 12-bit by 4-input multiplexer.

The two words that are presented to the memory lookup module will be interpreted by this module as an address which it uses to perform the actual analysis of the cell's validity. Depending on the version of the Analysis Module to be implemented, this function will change. Primarily, the memory lookup module will verify if the cell belongs to a connection that does indeed pass through the switch and port from which it originated. Optionally, the module will also verify if the network connection along which the cell in question is traveling has not exceeded the limits of traffic volume allowed for that connection.

This result will be used by the sequence / detect module to determine if the cell is valid or not. If the cell is valid, it will enable the output from the last set of latches in the 16-bit shift register to be sent out to the transmitter. If the cell is not valid, the sequence / detect module will suppress output of the cell from the shift register to the transmitter by simply presenting null data (all zero bits) to the input stage of the Receiver. In this case, the sequence / detect module will also will trigger interrupt logic in the Control Module. The Control Module will then know that an invalid cell has been detected and will perform the necessary operations to read the VPI/VCI pair of the offending cell from the outputs of the six 4-bit latches which have been storing this information throughout the entire process.

All of the devices used in this circuit are currently feasible in TTL and HC logic families. In addition, a number of tri-state buffers are implicitly being used in this design to allow the Control Module to select between the data inputs and outputs of the different Analysis Modules to which it is attached. The interconnection of the functional blocks of the Analysis Module is shown in figure 4.

The sequence / detect module which will be a simple sequential state machine with external decode logic will control all of the inputs and outputs required to perform the functions just described. This state machine will be designed using the same type of edge triggered D-type latches and combinational logic used to construct the other component blocks of the Analysis Module.

The reasoning behind the design of the Analysis Modules was to be able to take advantage of the large number of operations which can be performed in parallel in order to reduce the number of clock cycles necessary for the device to perform its function.

The effect on the performance of the physical communication link passing through this device will be that any cell in transit will be delayed by the amount of time necessary to read in the cell's header and perform the lookup of the VPI/VCI pair contained in these five bytes in the memory lookup module. This means that the controlling factor of the transmission delay a cell will experience in every security device through which it passes will be the sum of these two periods of time.

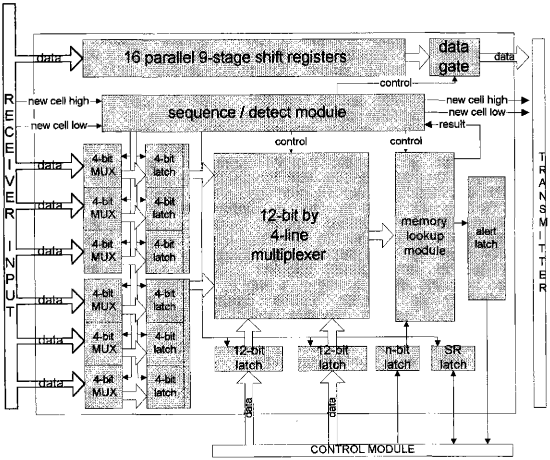


Fig. 4. Analysis Module block diagram

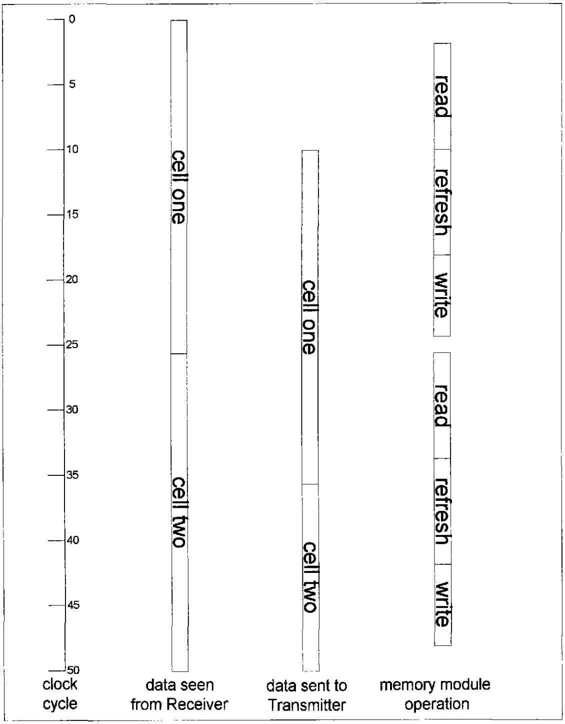


Fig. 5. Sequence of operations under full data load of the Analysis Module

The final issue is that the Control Module's interrupt logic will be triggered within less than one cell transmit time if the transiting cell is found to be invalid (nine clock cycles, to be precise). This means that the Control Module will know about the violation in less than one cell time and can begin sending data about the violation to its supervisory control interface within less than one cell time. The timing diagram for cell arrivals and departures from the Analysis Module is shown in figure 5.

The exact sequence of events within the Analysis Module will be:

1. Bytes 1 and 2 of cell one are presented on the inputs from the Receiver. These bytes are pushed into the shift register. The sequence / detect module is informed that a new cell has arrived through the assertion of the New Cell Low control line from the Receiver. The low order 4 bits of byte 1 and all 8 bits of byte 2 are presented and latched into the three high-order 4-bit latches.
2. Bytes 3 and 4 of cell one are presented and pushed into the shift register. All 8 bits of byte 3 and the high order four bits of byte 4 are presented and latched into the three low-order 4-bit latches.
3. Bytes 5 and 6 of cell one are presented and pushed into the shift register. The data in the three high order 4-bit latches is presented to the memory lookup module through the multiplexer.
4. Bytes 7 and 8 of cell one are presented and pushed into the shift register. The row address select is asserted on the memory lookup module.
5. Bytes 9 and 10 of cell one are presented and pushed into the shift register. The data in the three low order 4-bit latches is presented to the memory lookup module through the multiplexer.
6. Bytes 11 and 12 of cell one are presented and pushed into the shift register. The column address select is asserted on the memory lookup module.
7. Bytes 13 and 14 of cell one are presented and pushed into the shift register. The result concerning the validity of the cell will be read from the memory lookup module. If this result shows that the cell is invalid, the interrupt logic of the Control Module is triggered. Starting at this point, the Control Module may read the VPI/VCI pair stored in the six 4-bit latches in order to transmit this data about the traffic violation to the supervisory interface. The Control Module must complete the reading of this data within the next 20 clock cycles.
8. Bytes 15 and 16 of cell one are presented and pushed into the shift register. The sequence / detect module finishes the read cycle in the memory lookup module by deasserting the row address select line.
9. Bytes 17 and 18 of cell one are presented and pushed into the shift register. If the cell currently being received was found to be valid and the New Cell Low control input is asserted (meaning we are still receiving a cell with a starting byte on the low order bits of the Receiver input), the sequence / detect module sets the control on the low-order eight bits of the data gate to the Transmitter to reflect the inputs from the shift-register for the next 26 cycles. In this case, the sequence / detect module also sets the control on the high-order eight bits of the data gate to the Transmitter to reflect the inputs from the shift-register for the next 27 cycles. The sequence / detect module asserts the New Cell Low line to the Transmitter and bytes 1 and 2 of cell one are presented at the inputs of the Transmitter. If the cell currently being received was found to be valid and the New Cell High control input is asserted (meaning

we are receiving a cell with a starting byte on the high order bits of the Receiver input), the sequence / detect module sets the control on both the low-order and high-order eight bits of the data gate to the Transmitter to reflect the inputs from the shift-register for the next 27 cycles. The sequence / detect module asserts the New Cell High line to the Transmitter and byte 53 of cell one along with byte 1 of cell two are presented at the inputs of the Transmitter.

10. Bytes 19 and 20 of cell one are presented and pushed into the shift register. Either bytes 3 and 4 of cell one or bytes 2 and 3 of cell two are presented at the input of the Transmitter.
11. Bytes 21 and 22 of cell one are presented and pushed into the shift register. The sequence / detect module initiates a refresh cycle in the memory lookup module by asserting the row address select line. Either bytes 5 and 6 of cell one or bytes 4 and 5 of cell two are presented at the input of the Transmitter.
12. Bytes 23 and 24 of cell one are presented and pushed into the shift register. Either bytes 5 and 6 of cell one or bytes 4 and 5 of cell two are presented at the input of the Transmitter.
13. Bytes 25 and 26 of cell one are presented and pushed into the shift register. Either bytes 7 and 8 of cell one or bytes 6 and 7 of cell two are presented at the input of the Transmitter.
14. Bytes 27 and 28 of cell one are presented and pushed into the shift register. The sequence / detect module continues the refresh cycle of the memory lookup module by deasserting the row address select line. Either bytes 9 and 10 of cell one or bytes 8 and 9 of cell two are presented at the input of the Transmitter.
15. Bytes 29 and 30 of cell one are presented and pushed into the shift register. Either bytes 11 and 12 of cell one or bytes 10 and 11 of cell two are presented at the input of the Transmitter.
16. Bytes 31 and 32 of cell one are presented and pushed into the shift register. Either bytes 13 and 14 of cell one or bytes 12 and 13 of cell two are presented at the input of the Transmitter.
17. Bytes 33 and 34 of cell one are presented and pushed into the shift register. The sequence / detect module continues the refresh cycle of the memory lookup module by asserting the row address select line. Either bytes 15 and 16 of cell one or bytes 14 and 15 of cell two are presented at the input of the Transmitter.
18. Bytes 35 and 36 of cell one are presented and pushed into the shift register. The sequence / detect module finishes the refresh cycle of the memory lookup module by deasserting both the row address select line and the column address select line. Either bytes 17 and 18 of cell one or bytes 16 and 17 of cell two are presented at the input of the Transmitter.
19. Bytes 37 and 38 of cell one are presented and pushed into the shift register. If the Control Module needed to add or remove a valid path to or from the memory lookup module, it should have loaded a 24-bit latch with the appropriate VPI/VCI pair to be updated, an n-bit latch with the appropriate data about the path (the "n" bits depend on the design version chosen) and an SR-latch to indicate that path data needs to be updated by this point. If this SR-latch has been set, then the sequence / detect module begins a write cycle on the memory lookup module by setting the 12-bit line selector to reflect the high-order twelve bits of the 24-bit latch to the memory lookup module. Either bytes 19 and 20 of cell one or bytes 18 and 19 of cell two are presented at the input of the Transmitter.

20. Bytes 39 and 40 of cell one are presented and pushed into the shift register. If the SR-latch has been set, the sequence / detect module continues the write cycle by asserting the row address select line of the memory lookup module. Either bytes 21 and 22 of cell one or bytes 20 and 21 of cell two are presented at the input of the Transmitter.
21. Bytes 41 and 42 of cell one are presented and pushed into the shift register. If the SR-latch has been set, the sequence / detect module continues the write cycle on the memory lookup module by setting the 12-bit line selector to reflect the low-order twelve bits of the 24-bit latch. Either bytes 23 and 24 of cell one or bytes 22 and 23 of cell two are presented at the input of the Transmitter.
22. Bytes 43 and 44 of cell one are presented and pushed into the shift register. If the SR-latch has been set, the sequence / detect module continues the write cycle by asserting the column address select line of the memory lookup module. Either bytes 25 and 26 of cell one or bytes 24 and 25 of cell two are presented at the input of the Transmitter.
23. Bytes 45 and 46 of cell one are presented and pushed into the shift register. If the SR-latch has been set, both the column and row address select lines on the memory lookup module are deasserted. Either bytes 27 and 28 of cell one or bytes 26 and 27 of cell two are presented at the input of the Transmitter.
24. Bytes 47 and 48 of cell one are presented and pushed into the shift register. Either bytes 29 and 30 of cell one or bytes 28 and 29 of cell two are presented at the input of the Transmitter.
25. Bytes 49 and 50 of cell one are presented and pushed into the shift register. Either bytes 31 and 32 of cell one or bytes 30 and 31 of cell two are presented at the input of the Transmitter.
26. Bytes 51 and 52 of cell one are presented and pushed into the shift register. Either bytes 33 and 34 of cell one or bytes 32 and 33 of cell two are presented at the input of the Transmitter.
27. Byte 53 of cell one and byte 1 of cell two are presented and pushed into the shift register. The New Cell Low control input from the Receiver is deasserted and the New Cell High control input is asserted to inform the sequence / detect module that a new cell has arrived. The low-order 4 bits of byte 1 from cell two are presented and latched into the highest-order 4-bit latch. Either bytes 35 and 36 of cell one or bytes 34 and 35 of cell two are presented at the input of the Transmitter.
28. Bytes 2 and 3 of cell two are presented and pushed into the shift register. All bits from bytes 2 and 3 are presented and latched into the four 4-bit latches next to the highest-order 4-bit latch. Either bytes 37 and 38 of cell one or bytes 36 and 37 of cell two are presented at the input of the Transmitter.
29. Bytes 4 and 5 of cell two are presented and pushed into the shift register. The high-order 4 bits of byte 4 are presented and latched into the lowest order 4-bit latch. The data in the three high order 4-bit latches is presented to the memory lookup module through the multiplexer. Either bytes 39 and 40 of cell one or bytes 38 and 39 of cell two are presented at the input of the Transmitter.
30. Operation continues at step 4 with data continuing to be pumped out of the last stage of the shift register and into the Transmitter on every cycle and the remainder of cell two being pushed into the first stage of the shift register, two bytes at a time, on every cycle.

IV.C.1 Basic Building Blocks of the Analysis Module Design

The Analysis Module's design was conceived for gate-level implementation. Therefore, the atomic elements considered were NAND and NOR gates since they are the simplest building blocks for all the logic families currently in widespread digital semiconductor production. Due to the many different Boolean functions that needed to be implemented in order to make the design feasible, many versions of these gates were used, from the simplest two input gates up to six and seven input gates. The circuit symbol designations for the smallest of these basic gates are shown in figure 6.

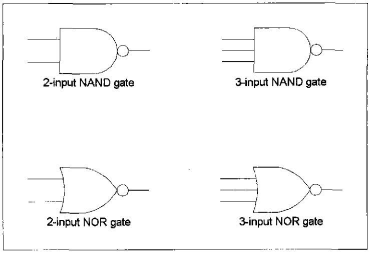


Fig. 6. Some of the atomic circuit units used in the design of the Analysis Module

Beginning with these basic units the first level of integration involved the construction of elementary latches and flip-flops. The nature of this design relies on two components for different elements of its operation. First, it relies, to a great extent, on positive edge-triggered latches for counters, state machines, storage elements and shift registers. Next, the design requires some SR type flip-flops for status tracking. Both of these devices were implemented and used as basic building blocks throughout the design. The circuit schemes for these devices are shown in figure 7 and figure 8.

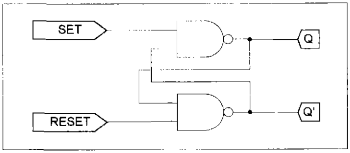


Fig. 7. SR-latch used in the design of the Analysis Module

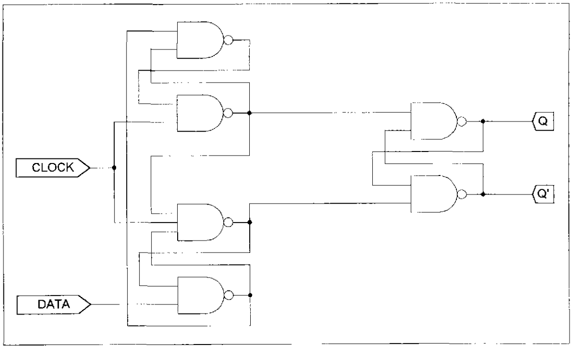


Fig. 8. Positive edge-triggered D-type flip-flops used in the design of the Analysis Module

IV.C.2 Shift Register Design

The Analysis Module relies on a shift register in order to temporarily store a portion of a cell that is received during the time required by the Analysis Module to determine if that cell can be further transmitted to the Transmitter block. In order for the Analysis Module to complete this task, 9 clock cycles are required. Therefore, this shift register must be able to delay incoming data by this same number of clock cycles. In addition, since data is presented from the Receiver in 16-bit words, the shift register must be able to capture all of this data on every clock cycle.

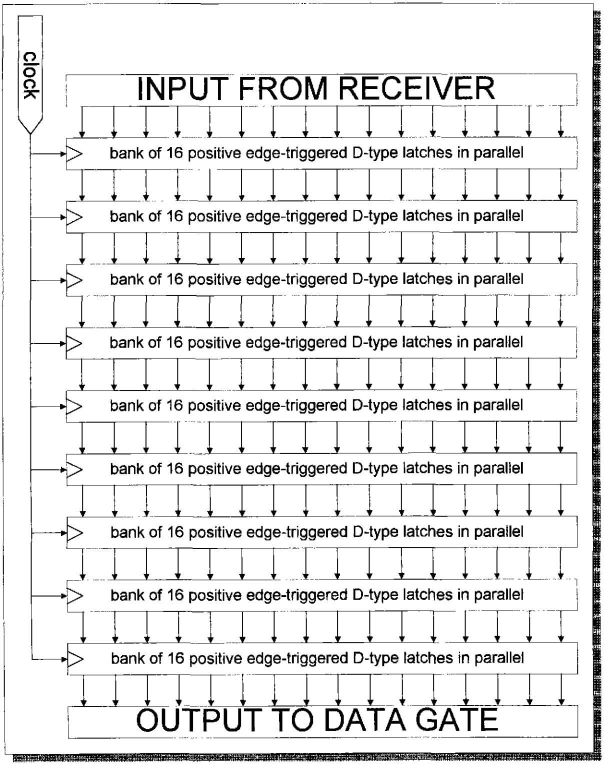

clock
INPUT FROM RECEIVER
bank of 16 positive edge-triggered D-type latches in parallel
bank of 16 positive edge-triggered D-type latches in parallel
bank of 16 positive edge-triggered D-type latches in parallel
bank of 16 positive edge-triggered D-type latches in parallel
bank of 16 positive edge-triggered D-type latches in parallel
bank of 16 positive edge-triggered D-type latches in parallel
bank of 16 positive edge-triggered D-type latches in parallel
bank of 16 positive edge-triggered D-type latches in parallel
bank of 16 positive edge-triggered D-type latches in parallel
OUTPUT TO DATA GATE

Fig. 9. Shift register used in the design of the Analysis Module

The shift register is built using positive edge-triggered D-type latches in 9 banks of 16 latches apiece. Each bank of 16 latches is linked in series to the next so that the output of a latch at position "n" will be captured and reflected at output "n+1" on the rising clock edge of the next clock cycle. The block diagram for this device is shown in figure 9.

IV.C.3 Multiplexer and Data Gate Design

The Analysis Module uses multiplexers in order to present different data to certain modules at the appropriate times. At its simplest, the multiplexer requires "n" inputs and "$\log_2 n$" control lines. Based on the state of the control lines, one of the inputs will be reflected at the output. A two-line multiplexer presents one of two inputs at its output, depending on the state of one control line. A four-line multiplexer presents one of four inputs at its output, depending on the state of two control lines. The Analysis Module relies on two-line and four-line multiplexers to parse the VPI/VCI pair from a newly arriving cell's header and to load this data into the memory lookup module. To implement these two forms of multiplexers, their output is expressed as a minimized Boolean expression. The Boolean expressions to implement these two devices are shown in figure 10.

Output = Input-1•Control′ + Input-2•Control	←	**two-line multiplexer**
Output = Input-1•Control-1′•Control-2′ + Input-2•Control-1′•Control-2 + Input-3•Control-1•Control-2′ + Input-4•Control-1•Control-2	←	**four-line multiplexer**

Fig. 10. Boolean expressions governing operation of multiplexers

The Analysis Module uses a data gate to control transmission of data from the shift register to the Transmitter. In its simplest form, the data gate requires an input and a control line. If the control line is asserted, the data gate's output will reflect what is presented at the input, otherwise, a value of zero will be presented. Once again, the output of this module can be expressed as a minimized Boolean expression. The Boolean expression to implement this device is shown in figure 11.

Output = Input•Control

Fig. 11. Boolean expressions governing operation of data gate

Both the multiplexers and data gates are expressed at the gate level as a series of NAND gates implementing the Boolean expressions governing the operation of these two devices. The circuit schemes governing the operation of the multiplexers and the data gate are shown in figure 12 and figure 13, respectively.

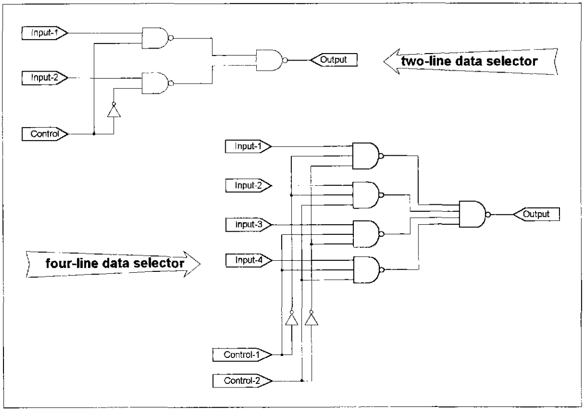


Fig. 12. Circuits governing operation of multiplexers

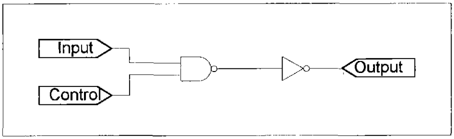


Fig. 13. Circuit governing operation of data gate

The design of the Analysis Module uses two types of multiplexers. The data latches used to store the VPI/VCI pair of the incoming cell rely on a series of six 4-bit by 2-line multiplexers to present different portions of this path information when new cells arrive with their first byte being presented on the low-order eight bits of the Receiver input and on the high-order eight bits of this input.

The second type of multiplexers are required to load data into the memory lookup module. There are four possible sources of data required by this module. Two of these sources are the high and low order bytes from the latches that store the VPI/VCI pair of the transiting cell. The remaining two sources are the high and low order bytes from the latches that store new path information that the Control Module requests to be loaded in the Analysis Module. Therefore, a 12-bit by 4-line data multiplexer is implemented.

The block diagrams for the implementation of the four-bit by two line multiplexer and the twelve-bit by four-line multiplexer are shown in figure 14 and figure 15, respectively.

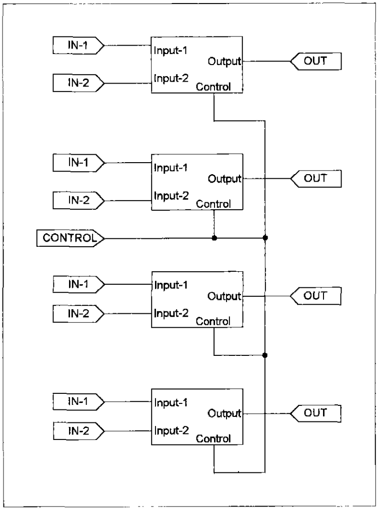


Fig. 14. Four-bit by two-line multiplexer used in the design of the Analysis Module

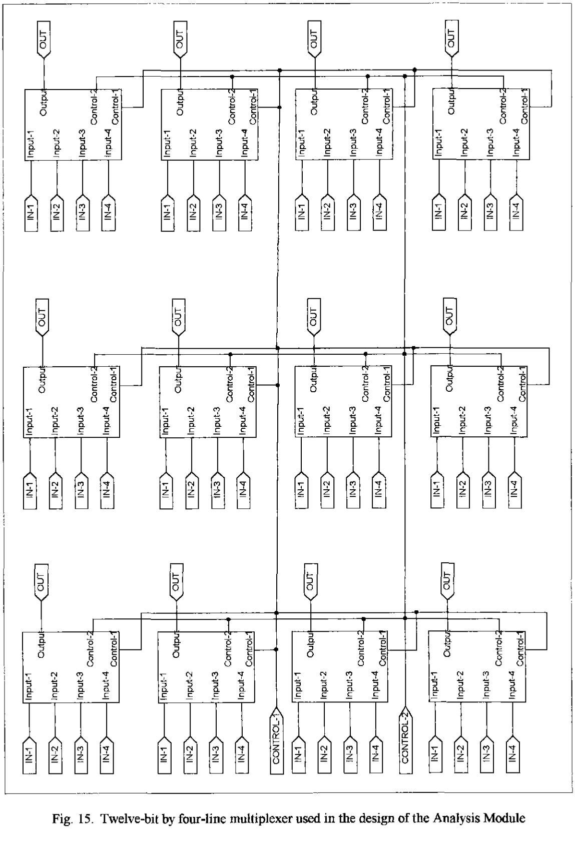


Fig. 15. Twelve-bit by four-line multiplexer used in the design of the Analysis Module

IV.C.4 Memory Lookup Module Design

The primary purpose of the *Analysis Module is to verify if network* traffic passing through it does not violate any path or, in the alternative implementation, path and volume restrictions. These two possible implementations are referred to as the "path-only" implementation and the "path and volume" implementation.

The key to performing the path validation of a transiting cell is to perform a lookup of its VPI/VCI pair in a table that associates a data field of one bit with each possible VPI/VCI pair. This table is implemented in a semiconductor memory external to the Analysis Module with an address bus that has the same width as a VPI/VCI pair and a data bus width of one bit. The ATM Forum's specification for the User Network Interface (version 3.0) requires that twenty-four bits be allocated for VPI/VCI information in the header of every cell. Therefore, this external memory must have a data bus width of twenty-four bits. Such a memory will have a total capacity of 16 megabits. The speed of the memory will dictate how many clock cycles the Analysis Module must wait before being able to determine if the transiting cell may be forwarded to the Transmitter.

With the current state of semiconductor technology, memories of the necessary density and speed have been implemented as monolithic integrated circuits. Once such product is the SMJ416100 dynamic random access memory (DRAM) from Texas Instruments. It offers an address bus width of twenty-four bits and a data bus width of one bit. After the necessary data has been presented on its address bus, the data for that address will become available within a maximum of 18 nanoseconds (for the SMJ416100-70 package). Since the Analysis Module clock is assumed to be operating at 38.88 MHz (thereby implying a clock period of 25.72 nanoseconds), we can guarantee that the data regarding whether or not a cell is valid will be available within one clock cycle *after* the VPI/VCI information has been presented.

Dynamic RAMs such as the SMJ416100 require that some maintenance be periodically performed in order to guarantee that the data stored within them will not become volatile. This maintenance consists of performing a series of refresh cycles within a specified period of time. For the SMJ416100 specifically, 4096 refresh cycles must be performed within every 35 millisecond time period in order to ensure that no data stored within the device will be altered inadvertently. This maintenance requirement can be resolved by combining the memory read operation required by the VPI/VCI pair lookup with one refresh cycle. Therefore, whenever a cell arrives and its path information has been parsed out of the header and presented to the memory, a read operation and a refresh operation can be performed in sequence. This operation is called a Hidden-Refresh-Read-Cycle in the literature of the SMJ416100.

The network data rate that the Analysis Module is required to support is 622.08 Mbits per second arriving in cells of 53 bytes apiece and with each byte consisting of 8 bits. This means that, under peak traffic conditions, 1.467 million cells will arrive per second. Since we are performing one read with refresh operation on every cell arrival, this implies that we will be performing the same number of refresh operations per second as there are cells that arrive. From this, it is possible to conclude that 46949 refresh operations will be performed every 35 milliseconds under peak traffic conditions, which well exceed the minimum number of 4096 established for this device.

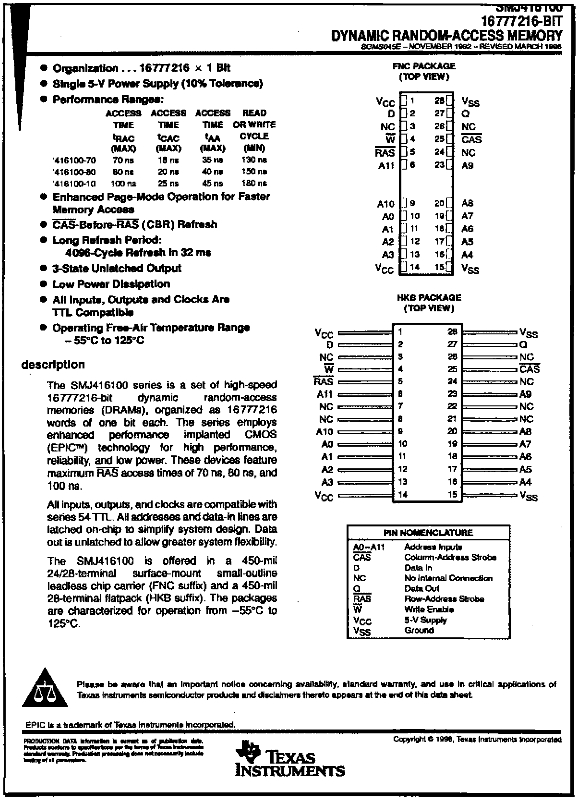
It is fairly evident that the number of refresh cycles that will be performed on the memory well exceeds the required minimum (by a factor of ten). However, reducing the number of refresh cycles performed to less than one for every cell arrival significantly complicates the design of the state machine inside of the memory lookup module, thereby, increasing the component count required for its implementation. Due to the fact that nothing in the literature about this device states or implies that performing such a large number of refresh cycles on it will lead to an increased chance of device failure before the expected end of its functional life, it was not seen as necessary to incorporate this reduction in refresh cycles within this design.

On every cell arrival, it is necessary to perform two operations. The first of these is the Hidden-Refresh-Read Cycle to verify the cell's validity and to perform the necessary maintenance on the memory. The second operation that must be performed is a Write Cycle to update valid path information that the Control Module has requested to be entered into the Analysis Module's local information. Each of these operations are initiated and carried out by following a sequence of events on the address bus (control lines A0 through A11) and data bus (control line D) of the memory and on the RAS' and CAS' control lines.

TABLE I
Timing characteristics as they appear in product data for the Texas Instruments SMJ416100-70 Dynamic RAM (DRAM)

[illegible]		MIN	MAX
t_{RC}	cycle time, random read or write	130 ns	
t_{RAS}	pulse duration, RAS′ low	70 ns	10,000 ns
t_{CSH}	delay time, RAS′ low to CAS′ going high	70 ns	
t_{RP}	pulse duration, RAS′ high	50 ns	
t_{CRP}	delay time, CAS′ high to RAS′ going low	5 ns	
t_{RCD}	delay time, RAS′ low to CAS′ low	20 ns	52 ns
t_{RSH}	delay time, CAS′ low to RAS′ going high	18 ns	
t_T	transition time	3 ns	30 ns
t_{CAS}	pulse duration, CAS′ low	18 ns	10,000 ns
t_{RAD}	delay time, RAS′ low to column address	15 ns	35 ns
t_{CP}	pulse duration, CAS′ high	10 ns	
t_{ASC}	setup time, column address before CAS′ going low	0 ns	
t_{ASR}	setup time, row address before RAS′ going low	0 ns	
t_{RAL}	delay time, column address to RAS′ going high	35 ns	
t_{RAH}	hold time, row address after RAS′ low	10 ns	
t_{CAL}	delay time, column address to CAS′ going high	35 ns	
t_{RCS}	setup time, W′ high before CAS′ going low	0 ns	
t_{CAH}	hold time, column address after CAS′ low	15 ns	
t_{RRH}	hold time, W′ high after RAS′ high	0 ns	
t_{CAH}	hold time, column address after CAS′ low	15 ns	
t_{CAC}	access time from CAS′ low		18 ns
t_{AA}	access time from column address		35 ns
t_{OFF}	output disable time after CAS′ high	0 ns	18 ns
t_{RAC}	access time from RAS' low		70 ns
t_{DS}	setup time, data	0 ns	
t_{CWL}	setup time, W′ low before CAS′ going high	18 ns	
t_{RWL}	setup time, W′ low before RAS′ going high	18 ns	
t_{WP}	pulse duration, W′ low	10 ns	
t_{DH}	hold time, data	15 ns	
t_{CHR}	delay time, RAS′ low to CAS′ going high	10 ns	
t_{WRH}	hold time, W′ high after RAS′ low	10 ns	
t_{WRP}	setup time, W′ high before RAS′ going low	10 ns	

The descriptions of the timing diagram designations for these dynamic RAMs, along with their minimum and maximum values, may be found in table I. Also, the refresh operation requirements are shown in figure 16.



SMJ416100
16777216-BIT
DYNAMIC RANDOM-ACCESS MEMORY
SGMS045E – NOVEMBER 1992 – REVISED MARCH 1996

- **Organization . . . 16777216 × 1 Bit**
- **Single 5-V Power Supply (10% Tolerance)**
- **Performance Ranges:**

	ACCESS TIME t_{RAC} (MAX)	ACCESS TIME t_{CAC} (MAX)	ACCESS TIME t_{AA} (MAX)	READ OR WRITE CYCLE (MIN)
'416100-70	70 ns	18 ns	35 ns	130 ns
'416100-80	80 ns	20 ns	40 ns	150 ns
'416100-10	100 ns	25 ns	45 ns	180 ns

- **Enhanced Page-Mode Operation for Faster Memory Access**
- **$\overline{CAS}$-Before-$\overline{RAS}$ (CBR) Refresh**
- **Long Refresh Period: 4096-Cycle Refresh in 32 ms**
- **3-State Unlatched Output**
- **Low Power Dissipation**
- **All Inputs, Outputs and Clocks Are TTL Compatible**
- **Operating Free-Air Temperature Range – 55°C to 125°C**

FNC PACKAGE
(TOP VIEW)

V_{CC}	1	28	V_{SS}
D	2	27	Q
NC	3	26	NC
$\overline{W}$	4	25	$\overline{CAS}$
$\overline{RAS}$	5	24	NC
A11	6	23	A9
A10	9	20	A8
A0	10	19	A7
A1	11	18	A6
A2	12	17	A5
A3	13	16	A4
V_{CC}	14	15	V_{SS}

HKB PACKAGE
(TOP VIEW)

V_{CC}	1	28	V_{SS}
D	2	27	Q
NC	3	26	NC
$\overline{W}$	4	25	$\overline{CAS}$
$\overline{RAS}$	5	24	NC
A11	6	23	A9
NC	7	22	NC
NC	8	21	NC
A10	9	20	A8
A0	10	19	A7
A1	11	18	A6
A2	12	17	A5
A3	13	16	A4
V_{CC}	14	15	V_{SS}

PIN NOMENCLATURE	
A0–A11	Address Inputs
$\overline{CAS}$	Column-Address Strobe
D	Data In
NC	No Internal Connection
Q	Data Out
$\overline{RAS}$	Row-Address Strobe
$\overline{W}$	Write Enable
V_{CC}	5-V Supply
V_{SS}	Ground

description

The SMJ416100 series is a set of high-speed 16777216-bit dynamic random-access memories (DRAMs), organized as 16777216 words of one bit each. The series employs enhanced performance implanted CMOS (EPIC™) technology for high performance, reliability, and low power. These devices feature maximum $\overline{RAS}$ access times of 70 ns, 80 ns, and 100 ns.

All inputs, outputs, and clocks are compatible with series 54 TTL. All addresses and data-in lines are latched on-chip to simplify system design. Data out is unlatched to allow greater system flexibility.

The SMJ416100 is offered in a 450-mil 24/28-terminal surface-mount small-outline leadless chip carrier (FNC suffix) and a 450-mil 28-terminal flatpack (HKB suffix). The packages are characterized for operation from –55°C to 125°C.

Please be aware that an important notice concerning availability, standard warranty, and use in critical applications of Texas Instruments semiconductor products and disclaimers thereto appears at the end of this data sheet.

EPIC is a trademark of Texas Instruments Incorporated.

PRODUCTION DATA information is current as of publication date. Products conform to specifications per the terms of Texas Instruments standard warranty. Production processing does not necessarily include testing of all parameters.

Copyright © 1996, Texas Instruments Incorporated

TEXAS INSTRUMENTS

Fig. 16. Product characteristics information as it appears in product data document for the Texas Instruments SMJ416100 Dynamic RAM (DRAM)

SMJ416100
16777216-BIT
DYNAMIC RANDOM-ACCESS MEMORY
SGMS045E – NOVEMBER 1992 – REVISED MARCH 1996

PARAMETER MEASUREMENT INFORMATION

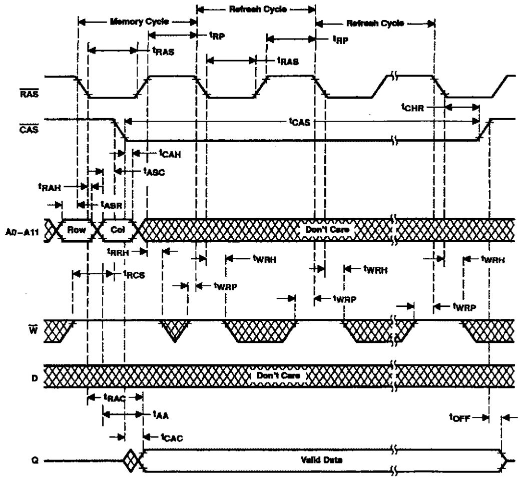


Figure 11. Hidden-Refresh-Cycle (Read) Timing

TEXAS INSTRUMENTS

Fig. 17. Hidden-Refresh-Read Cycle timing diagram as it appears in product data document for the Texas Instruments SMJ416100 Dynamic RAM (DRAM)

SMJ416100
16777216-BIT
DYNAMIC RANDOM-ACCESS MEMORY
SGMS045E – NOVEMBER 1992 – REVISED MARCH 1998

PARAMETER MEASUREMENT INFORMATION

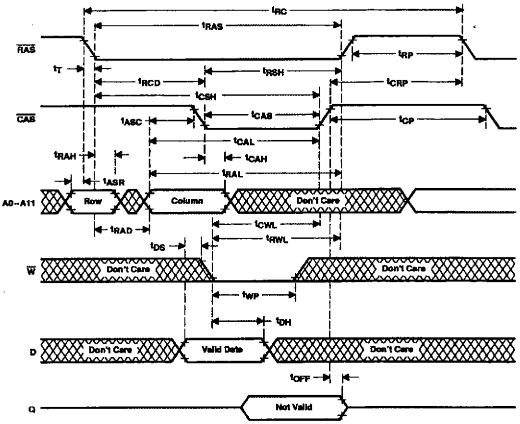


Figure 4. Write-Cycle Timing

TEXAS INSTRUMENTS

Fig. 18. Write Cycle timing diagram as it appears in product data document for the Texas Instruments SMJ416100 Dynamic RAM (DRAM)

The timing diagrams for the Hidden-Refresh-Read and Write operations on these dynamic RAMs are shown in figure 17 and figure 18, respectively.

In order to perform a Hidden-Refresh-Read Cycle, the following must occur in sequence (assuming that, initially, the memory's RAS′, CAS′ and W′ control lines are unasserted):

1. The high-order twelve bits of the address are presented on the address bus (read).
2. The RAS′ control line is asserted by being driven low and the data on the address bus continues to be held there for 10 nanoseconds more (read).
3. The low-order twelve bit of the address are presented on the address bus (read).
4. The CAS′ control line is asserted by being driven low and the data on the address bus continues to be held there for 15 nanoseconds more (read).
5. Wait for 3 nanoseconds to ensure that, at least, 18 nanoseconds have elapsed since the CAS′ line was asserted and read or latch the data at the address from the Q output (read).
6. Wait for 42 nanoseconds to ensure that, at least, 70 nanoseconds have elapsed since the RAS′ line was asserted and deassert the RAS′ control line by driving it high (read).
7. Wait for 50 nanoseconds and reassert the RAS′ control line by driving it low (refresh).
8. Wait for 70 nanoseconds and deassert the RAS′ control line by driving it high (refresh).
9. Wait for 50 nanoseconds and reassert the RAS′ control line by driving it low (refresh).
10. Wait for 10 nanoseconds and deassert both the RAS′ and CAS′ control lines by driving them both high (refresh).

In order to perform a Write Cycle, the following must occur in sequence (assuming that, initially, the memory's RAS′, CAS′ and W′ control lines are unasserted):

1. The high-order twelve bits of the address are presented on the address bus.
2. The RAS′ control line is asserted by being driven low and the data on the address bus continues to be held there for 10 nanoseconds more.
3. The low-order twelve bits of the address are presented on the address bus and the data to be written to that address is presented on the data bus.
4. The RAS′ and W′ control lines are asserted by driving them low and the data on the address bus continues to be held for 10 nanoseconds more while the data on the data bus continues to be held for 15 nanoseconds more.
5. Wait for 3 nanoseconds to ensure that, at least, 18 nanoseconds have elapsed since the CAS′ line was asserted and deassert both the RAS′ and CAS′ control lines.

With these details regarding the operation of the external dynamic RAM on which the memory lookup module will rely having been presented, the design of this module for the "path-only" version of the Analysis Module will simply direct the control lines of this external memory (pins RAS′, CAS′ and W′) into the sequence detect module, the address bus (pins A0 through A11) into the twelve-bit by four-line selector, the data bus (pin D) to the output of the "n-bit latch" and the data output (pin Q) to a latch within the sequence / detect module. Also, in the "path-only" implementation the "n-bit latch" will only be required to capture one bit of information since only this one bit is required to indicate whether or not the newly arriving cell is valid. The use of a dynamic RAM as a memory lookup module in the "path-only" version of the Analysis Module implementation is shown in figure 19.

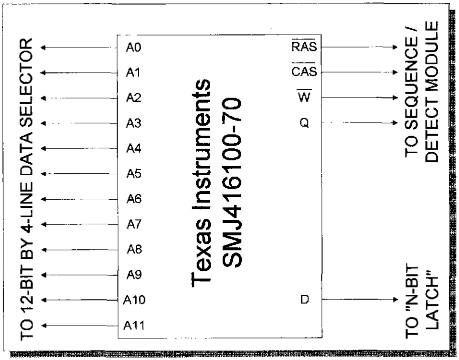


Fig. 19. Memory lookup module used in the design of the "path-only" version of the Analysis Module

The design of the memory lookup module for the "path and volume" version of the analysis module requires a significant amount of additional circuitry in order to ensure that transiting cells will experience a delay of less than one cell time. The "path and volume" version requires that we maintain two pieces of information regarding each possible network connection through the security device. First, as in the "path-only" implementation, it is necessary to determine whether or not a transiting cell belongs to a connection which is valid for the Analysis Module. Second, it is necessary to be able to store a short term history about every valid connection passing through a given Analysis Module. This short term history is the number of cells which have passed through the Analysis Module along a particular valid connection within a known window of time. Therefore, this requirement stipulates that no more than a certain known number of cells will be allowed to pass through the Analysis Module along a particular connection within that known window of time.

In order to implement this second requirement, a small storage unit must be used which will track the number of cells that are allowed to pass through the Analysis Module along a particular connection. Every single time that a cell belonging to that path passes through the module, the binary value stored within this storage unit will be decremented to reflect the event.

However, a mechanism must also exist by which this number of allowable cells can be replenished. Otherwise, the volume of traffic along a particular connection could only be monitored for an infinitely long window of time. Therefore, in addition to the ability to decrement the value in this storage unit, it is also necessary to be able to periodically increment this value. This implies a storage device with two control lines. If one control line is asserted, then, on the next clock cycle, the device must increment the value which it stores and, if the other control line in asserted, then, on the next clock cycle, the device must decrement the value which it stores.

This device can most easily be implemented as a simple counter with two sets of counting logic. If the decrement control line is asserted, then the count down logic is enabled on the next clock cycle. If the increment control line is asserted, then the count up logic is enabled on the next clock cycle. If both or neither the increment and decrement control lines are asserted, then the device will retain its current value on the next clock cycle.

According to the logic of its operation, this device will be able to alert the Analysis Module if a valid transiting cell has exceeded its traffic volume restrictions when the value stored internally reaches zero. However, as currently described, the device could conceivably also come to store an internal value of zero if, on some particular clock cycle, the internal value is the maximum value that the device can store (all bits set to one) and the increment control line is asserted. Likewise, a valid alarm due to a zero internal value may be stopped if another cell transits through the Analysis Module while the alarm is triggered because the internal state of the device would shift from all zero bits to all one bits, if the count down logic is operating properly. In order to prevent these two events from occurring, we must restrict the count up and count down logic from wrapping around to all zeroes or all ones when the extremes of the counting range have been reached.

This device will be called a "counter with control" for the purposes of this design and its block diagram is shown in figure 20. The operational states of the device are shown in table II.

TABLE II
Operational states of the "counter with control" to be implemented in the "path and volume" version of the memory lookup module

Increment Control	Decrement Control	Reason	Next State

asserted	unasserted	the replenishment time period has elapsed	if (current state ≠ all ones)
			current state + 1
			else
			current state

unasserted	asserted	a valid cell is transiting through the device	if (current state ≠ all zero)
			current state - 1
			else
			current state

asserted	asserted	a valid cell is transiting through the device and the replenishment time period has elapsed	current state
unasserted	unasserted	no event occurred	current state

The entire design of the Analysis Module centers around being able to perform the individual steps of capture and analysis of newly arriving cells within one cell time and synchronized to a clock whose period is equal to the time that is required to receive two bytes from the external network. Because of this requirement, this "counter with control" must be able to update its internal state within a bounded period of time. The reason this bounded period of time is important is that we cannot stipulate that the internal state of this counter consist of any certain fixed number of bits. Therefore, the implementation of this device as a ripple counter or any series of partial adders is ruled out due to the fact that the time required for complete state update from one clock cycle to the next in these devices is linearly dependent on the number of bits which make up the internal state of the device.

Additionally, it is necessary to determine if a valid cell has violated the volume restrictions for its connections without delaying that cell more than one cell transmission time. It has already been shown that the "path-only" version of the Analysis Module requires almost one complete cell time to perform its function leaving only seven byte times short of a complete cell, which translates to only 3½ clock cycles. Since the "counter with control" will be performing a function that is in addition to that of the "path-only" version, it must be able to provide this result within less than this period of time. In order to keep within this time restriction for state updates when the internal state of the counter is made up of an indeterminate number of bits, it is necessary to employ logic to fully decode the current state of the device on every state transition.

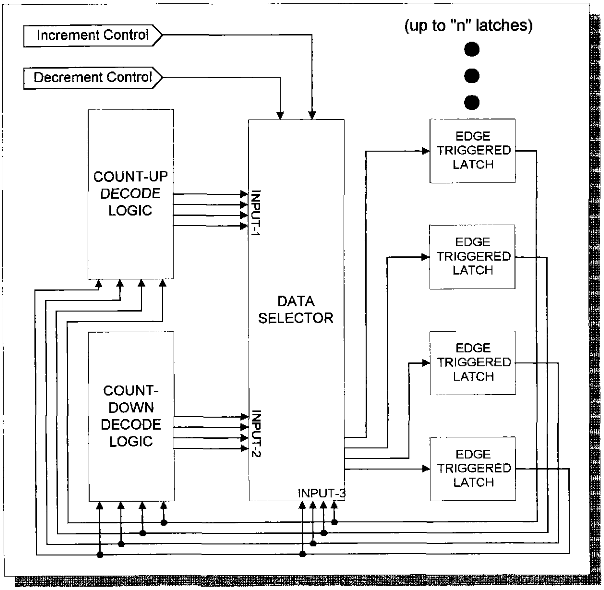


Fig. 20. "Counter with control" for the memory lookup module used in the design of the "path and volume" version of the Analysis Module

The logic necessary to implement the count-up and count-down logic blocks of the "counter with control" depends only on the number of latches that determine the internal state of the counter and whether or not the bit for which the decode is being performed is the highest order bit. This highest order bit will be labeled b_n and will be referred to as the terminal bit, with all of the other bits being labeled b_0 through b_{n-1} and being referred to as non-terminal bits. The Boolean expressions necessary to implement the decoding of both the terminal and non-terminal bits is shown in tables III, IV, V and VI for the five lowest order bits (the Boolean expressions for additional higher order bits may be generalized from these expressions). The ellipsis at the end of each expression indicates that the remaining higher order bit literals are ANDed or ORed with all the remaining bits available in the counter.

TABLE III
Boolean expressions necessary to implement the decode logic for the five lowest-order non-terminal bits of a fully decoded count-up counter with no roll-over

BIT	Expression (non-terminal bit of up-counter)
b_0	$(b_0') + (b_0)(b_1)(b_2)(b_3)(b_4)(b_5)(b_6)(b_7)\ldots$
b_1	$(b_0)(b_1') + (b_0')(b_1) + (b_0)(b_1)(b_2)(b_3)(b_4)(b_5)(b_6)(b_7)\ldots$
b_2	$(b_0)(b_1)(b_2') + (b_1')(b_2) + (b_0')(b_2) + (b_0)(b_1)(b_2)(b_3)(b_4)(b_5)(b_6)(b_7)\ldots$
b_3	$(b_0)(b_1)(b_2)(b_3') + (b_2')(b_3) + (b_1')(b_3) + (b_0')(b_3) + (b_0)(b_1)(b_2)(b_3)(b_4)(b_5)(b_6)(b_7)\ldots$
b_4	$(b_0)(b_1)(b_2)(b_3)(b_4') + (b_3')(b_4) + (b_2')(b_4) + (b_1')(b_4) + (b_0')(b_4) + (b_0)(b_1)(b_2)(b_3)(b_4)(b_5)(b_6)(b_7)\ldots$

TABLE IV
Boolean expressions necessary to implement the decode logic for the five lowest-order terminal bits of a fully decoded count-up counter with no roll-over

BIT	Expression (terminal bit of up-counter)
b_0	1
b_1	$b_0 + b_1$
b_2	$(b_0)(b_1) + b_2$
b_3	$(b_0)(b_1)(b_2) + b_3$
b_4	$(b_0)(b_1)(b_2)(b_3) + b_4$

TABLE V
Boolean expressions necessary to implement the decode logic for the five lowest-order non-terminal bits of a fully decoded count-down counter with no roll-over

BIT	Expression (non-terminal bit of down-counter)
b_0	$(b_0')(b_0 + b_1 + b_2 + b_3 + b_4 + b_5 + b_6 + b_7 +\ldots)$
b_1	$(b_0 + b_1')(b_0' + b_1)(b_0 + b_1 + b_2 + b_3 + b_4 + b_5 + b_6 + b_7 +\ldots)$
b_2	$(b_0 + b_1 + b_2')(b_1' + b_2)(b_0' + b_2)(b_0 + b_1 + b_2 + b_3 + b_4 + b_5 + b_6 + b_7 +\ldots)$
b_3	$(b_0 + b_1 + b_2 + b_3')(b_2' + b_3)(b_1' + b_3)(b_0' + b_3)(b_0 + b_1 + b_2 + b_3 + b_4 + b_5 + b_6 + b_7 +\ldots)$
b_4	$(b_0 + b_1 + b_2 + b_3 + b_4')(b_3' + b_4)(b_2' + b_4)(b_1' + b_4)(b_0' + b_4)(b_0 + b_1 + b_2 + b_3 + b_4 + b_5 + b_6 + b_7 \ldots)$

TABLE VI
Boolean expressions necessary to implement the decode logic for the five lowest-order terminal bits of a fully decoded count-down counter with no roll-over

BIT	Expression (terminal bit of down-counter)
b_0	0
b_1	$(b_0)(b_1)$
b_2	$b_0 + (b_1)(b_2)$
b_3	$(b_0 + b_1 + b_2)(b_3)$
b_4	$(b_0 + b_1 + b_2 + b_3)(b_4)$

The design of the multiplexer will not be presented here as this area has already been covered by other sections of the Analysis Module design.

To build on the functionality provided by the "counter with control", it is necessary to provide a method by which this counter may be incremented in order to continually replenish the bandwidth available to any valid connection. However, it is necessary to do this in such a fashion that every connection be allowed to maintain their own rate of replenishment and, in addition, to be able to update the replenishment rate for any new connection that is created. Therefore, the output of a simple fixed clock divisor will not be sufficient to drive the "Increment Control" on the "counter with control". For a connection that is valid, it is necessary to allow for this replenishment rate to be programmable.

This can be accomplished with a multiple-bit latch (called a register from this point on) and a simple counter with reset control which counts up on every clock cycle. When the internal state of the counter with reset control exactly matches the internal state of the register, an equality tester can be used to trigger an event. This event will reset the simple counter to an null internal state (all bits zero) and will also act as the "Increment Control" for the "counter with control". Once this collection of register, simple counter with reset, equality tester and "counter with control" has been implemented, the entire system should work in tandem to provide a unit that replenishes its allowable data at a rate which is programmable and sets of an alarm signal whenever the traffic rate (signaled by the "Decrement Control") has exceeded the rate allowed within the programmable window (i.e. the "counter with control" has reached an internal state consisting of all zero bits). This module will be referred to as a "window control module".

The simple counter with reset used in the window control module must also be a fully decoded counter since it too must change states in a fashion that is independent of the number of bits that make up its internal state. However, since there is no requirement that this device not roll over from a state of all one bits to all zero bits, the decode logic necessary for each bit is greatly simplified over that of the "counter with control". Moreover, there is no longer any difference in the logic required for terminal and non-terminal bits in the expressions describing the next state decode logic.

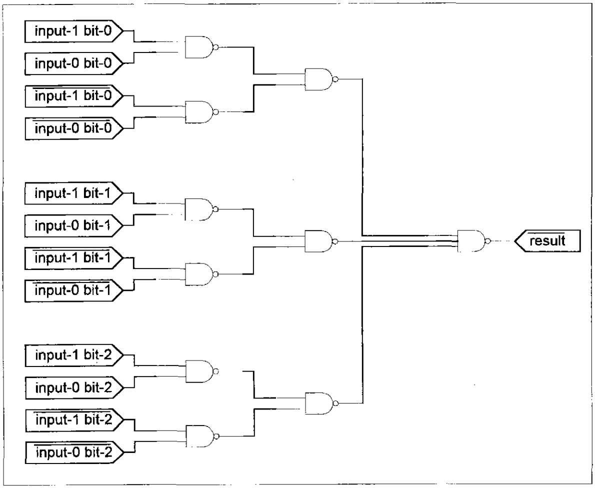


Fig. 21. Circuitry for an example three-bit equality tester for the memory lookup module used in the design of the "path and volume" version of the Analysis Module

The Boolean expressions necessary to implement the decoding of all bits in the simple counter with reset control is shown in table VII for the five lowest order bits (the Boolean expressions for additional higher order bits may be generalized from these expressions).

TABLE VII
Boolcan expressions necessary to implement the decode logic for the five lowest-order bits of a simple up-counter with reset control

Bit	Expression
b_0	b_0'
b_1	$(b_0)(b_1') + (b_0')(b_1)$
b_2	$(b_0)(b_1)(b_2') + (b_1')(b_2) + (b_0')(b_2)$
b_3	$(b_0)(b_1)(b_2)(b_3') + (b_2')(b_3) + (b_1')(b_3) + (b_0')(b_3)$
b_4	$(b_0)(b_1)(b_2)(b_3)(b_4') + (b_3')(b_4) + (b_2')(b_4) + (b_1')(b_4) + (b_0')(b_4)$

The zero tester unit of the window control module must simply assert its result if and only if all bits of its input are zeroes. This can very easily be accomplished with a multiple input NOR gate. Therefore, for an "n-bit" zero tester, all that is required is an n-input NOR gate.

The equality tester unit of the window control module must compare all of the bits of one input against all of the bits of the other input and assert its result output if and only if all of these bits match. The comparison of the individual bits would best be performed by an XNOR gate. However, due to the low level nature of this design, the XNOR function will be implemented with component NAND gates. Once each of the corresponding bits from the two inputs have been tested with the XNOR function, the resulting equality may be tested by applying the results of all the XNOR functions to a multiple input NAND gate. The circuitry for this device is shown in figure 21.

The interconnection of all of these sub-units in the make up of one window control module for the "path and volume" version of the Analysis Module is shown in figure 22. Once the window control module's functionality has been described, it becomes feasible to implement the "path and volume" version of the Analysis Module so that valid path enforcement and connection volume enforcement both occur with the cell in transit experiencing a delay of less than one cell time. The design will rely on the same dynamic RAMs on which the "path-only" version relied. However, more than one memory will now be used to provide more detailed information about the path along which an arriving cell is traveling.

Instead of simply using the DRAM to provide information about whether the connection to which the transiting cell belongs exists, this memory will now be used to provide a mapping from a VPI/VCI pair to one specific window control module within the Analysis Module that controls the volume information regarding that cell. As before, the first steps will be to present the VPI/VCI pair of the transiting cell as an address to the DRAM. However, the data provided by the memory will now be richer in content. If the data returned is null, then the VPI/VCI pair for the transiting cell will be assumed to belong to an invalid connection, an alert will be raised for this reason and the cell will be dropped. However, if the data presented by the memory for a particular VPI/VCI pair is not null, then the transiting cell is passing along a connection that is valid for the Analysis Module. This result data from the memory will be forwarded through a data gate to a demultiplexer which will, in turn, assert the "valid cell arrived" control line on one unique window control module.

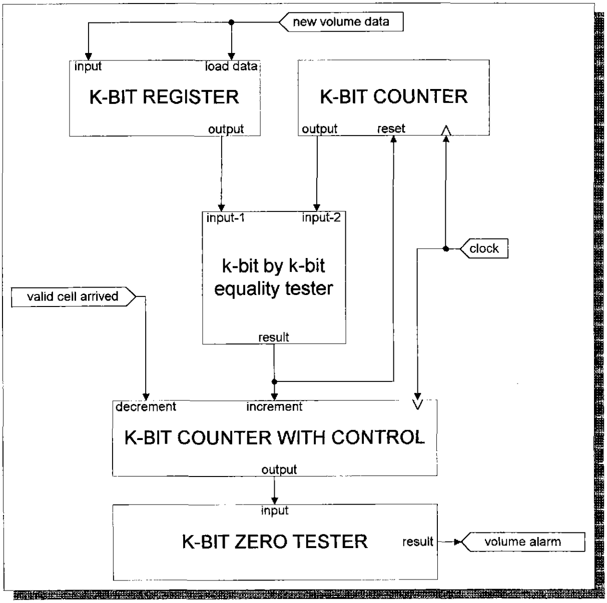


Fig. 22. Window control module for the memory lookup module used in the design of the "path and volume" version of the Analysis Module with "k" bits of control granularity

This use of DRAMs in the memory lookup module implies that more than one DRAM unit will be required. The total number of these memories that are required is a function of the total number of window control modules available within the Analysis Module. As an example, if the Analysis Module is equipped with sixteen window control modules, then four bits will be required to uniquely select one of these modules which, in turn, implies that four Texas Instruments SMJ416100 devices would be required in order to provide the necessary four bits of data. These four bits of data would then be passed to a four-to-sixteen demultiplexer which will, in turn, assert the "valid cell arrived" signal on one unique control module. The block diagram for a four-window control module "path and volume" Analysis Module is shown in figure 23.

As a general result, for every "W" window control modules available in an Analysis Module, a total of ($\log_2$ W) DRAMs will be required, each with a density of 16 Mbits, along with a "($\log_2$ W) to W" demultiplexer.

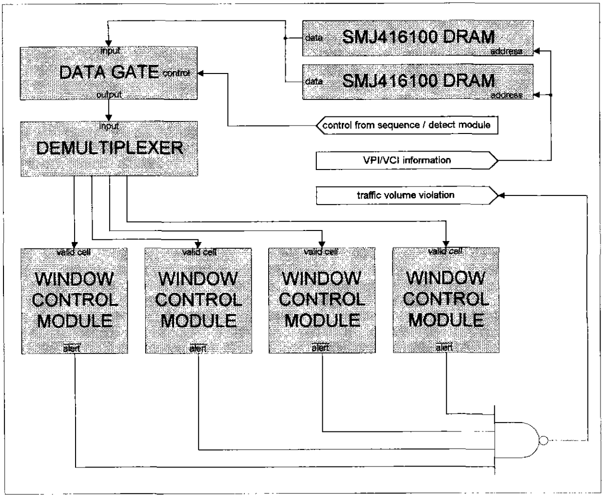


Fig. 23. Block diagram for the "path and volume" version of the Analysis Module

IV.C.5 Sequence / Detect Module Design

The sequence / detect module performs all of the necessary functions within the Analysis Module that guide it through the various functions it has to perform when receiving new cells from the Receiver. The sequence / detect module asserts and deasserts the control lines on the major logic blocks described in the top-level layout of the Analysis module and it does this only at the appropriate times. To be more specific, it sets the control lines to the dynamic memory or memories, the 4-bit multiplexers, the 4-bit data latches, the 12-bit by 4-line multiplexer, the memory lookup module, the traffic alert latch or latches and the New Cell High and New Cell Low lines to the Transmitter and the data gate that controls cell output to the Transmitter.

The core of the sequence / detect module is a state machine that cycles through a total of 28 states in order to perform all of the necessary operations on the control lines leading to the various blocks of logic. The actual signals that must be controlled are shown in table VIII.

TABLE VIII
Signal names and descriptions of the control lines set by the state machine internal to the sequence / detect module

Name	Signal Description
4BDS[1] 4BDS[2] 4BDS[3] 4BDS[4] 4BDS[5] 4BDS[6]	Controls which of the two possible inputs are reflected at the outputs of the 4-bit multiplexers. Each of the six bits control one multiplexer and each multiplexer is numbered #1 to #6 with #1 being the highest order and #6 being the lowest order multiplexer. (Found on both the "path-only" and the "path and volume" implementations of the Analysis Module.)
4BDL[1] 4BDL[2] 4BDL[3] 4BDL[4] 4BDL[5] 4BDL[6]	Controls the latching on the 4-bit latches that store the VPI/VCI information regarding a transiting cell. These are positive edge-triggered latches, therefore, latching occurs when these signals transition from low to high. Each of the six bits control one 4-bit latch and each latch is numbered #1 to #6 with #1 being the highest order and #6 being the lowest order latch. (Found on both the "path-only" and the "path and volume" implementations of the Analysis Module.)
12BDS[B] 12BDS[S]	Controls which of the four possible inputs are reflected at the outputs of the 12-bit by 4-line multiplexer. When the "B" signal is high, the high order twelve bits of the possible inputs will be reflected at the output and when the "B" signal is low, the low order twelve bits of the possible inputs will be reflect at the output. When the "S" signal is high, the VPI/VCI pair information will be reflected at the output and when the "S" signal is low, the new path information will be reflected at the output. (Found on both the "path-only" and the "path and volume" implementations of the Analysis Module.)
PVRL	Controls the latching on the latch or latches (depending on the implementation) which store the result of the memory read on the memory lookup module. These are positive edge-triggered latches, therefore, latching occurs when this signal transitions from low to high. (Found on both the "path-only" and the "path and volume" implementations of the Analysis Module.)

TABLE VIII
(continued)

Name	Signal Description
RSRL	When asserted high, this signal will reset the SR-latch that indicates to the Control Module whether the information about a new path that was last loaded has been stored in the memory lookup module. (Found on both the "path-only" and the "path and volume" implementations of the Analysis Module.)
LLODG26	When asserted high, this signal will load the counter that controls whether the data gate to the transmitter will allow to pass the low order byte from the shift register. When asserted, this signal will load that counter with a value of 26, indicating that the data gate will allow the next sequence of 26 bytes on the low-order byte output from the shift register to pass through it and on to the Transmitter. (Found on both the "path-only" and the "path and volume" implementations of the Analysis Module.)
LLODG27	When asserted high, this signal will load the counter that controls whether the data gate to the transmitter will allow to pass the low order byte from the shift register. When asserted, this signal will load that counter with a value of 27, indicating that the data gate will allow the next sequence of 27 bytes on the low-order byte output from the shift register to pass through it and on to the Transmitter. (Found on both the "path-only" and the "path and volume" implementations of the Analysis Module.)
LHODG27	When asserted high, this signal will load the counter that controls whether the data gate to the transmitter will allow to pass the high order byte from the shift register. When asserted, this signal will load that counter with a value of 27, indicating that the data gate will allow the next sequence of 27 bytes on the high-order byte output from the shift register to pass through it and on to the Transmitter. (Found on both the "path-only" and the "path and volume" implementations of the Analysis Module.)
RAS'	This is the row address select control line to the external dynamic RAM that is found in the memory lookup module. This line is used to latch address information when read and write cycles are being performed on the memory. This signal is asserted low when these two operations are being performed according to the data sheets describing these two procedures for the Texas Instruments SMJ416100 DRAM. (Found on both the "path-only" and the "path and volume" implementations of the Analysis Module.)
CAS'	This is the column address select control line to the external dynamic RAM that is found in the memory lookup module. This line is used to latch address information when read and write cycles are being performed on the memory. This signal is asserted low when these two operations are being performed according to the data sheets describing these two procedures for the Texas Instruments SMJ416100 DRAM. (Found on both the "path-only" and the "path and volume" implementations of the Analysis Module.)
W'	This is the read / write control line to the external dynamic RAM that is found in the memory lookup module. This line is used to indicate whether a write or a read operation is being performed on the memory. This signal is asserted low when a write operation is in progress and deasserted high when a read operation is in progress according to the data sheets describing these two procedures for the Texas Instruments SMJ416100 DRAM. (Found on both the "path-only" and the "path and volume" implementations of the Analysis Module.)
VVRL	Controls the latching on the latches which store the result of the window control module activity performed after the memory read in the memory lookup module. These are positive edge-triggered latches, therefore, latching occurs when this signal transitions from low to high. (Found on both the "path-only" and the "path and volume" implementation of the Analysis Module.)
SDDG	When asserted high, this signal instructs the data gate at the input to the demultiplexer in the window control module to reflect the data on the data bus of the memories at its output. When not asserted, the data gate will reflect null values at its outputs (all zero bits).

The state machine guiding the operation of the sequence / detect module requires some external information about the events of new cell arrivals. New cells can arrive at the Analysis Module with either the first byte being presented on the low-order eight bits of the Receiver's output stage or with the first byte being presented on the high-order eight bits of the Receiver's output stage. In order to be able to determine when these events are occurring, the state machine uses, as external controls, the New Cell High and New Cell Low signals which are generated by the Receiver. The method in which these signals behave is described in the section devoted to the Receiver's design. The designations for these signals are shown in table IX.

TABLE IX
Signal names and descriptions of the external signals the state machine internal to the sequence / detect module requires

Name	Signal Description
NCHIN	When asserted high, it indicates the arrival of a new cell from the Receiver with the first byte of that cell being presented on the high-order eight bits of the Receiver's output stage.
NCLIN	When asserted high, it indicates the arrival of a new cell from the Receiver with the first byte of that cell being presented on the high-order eight bits of the Receiver's output stage.

The exact states of each of the signals controlled by the state machine for each state in the state diagram are shown in table X (don't care conditions appear as blank fields in the table). Internal values which have not been assigned to any specific state are naturally assumed to produce a don't care condition for all signals. The state diagram for the state machine guiding the actions of the sequence / detect module are shown in figure 24.

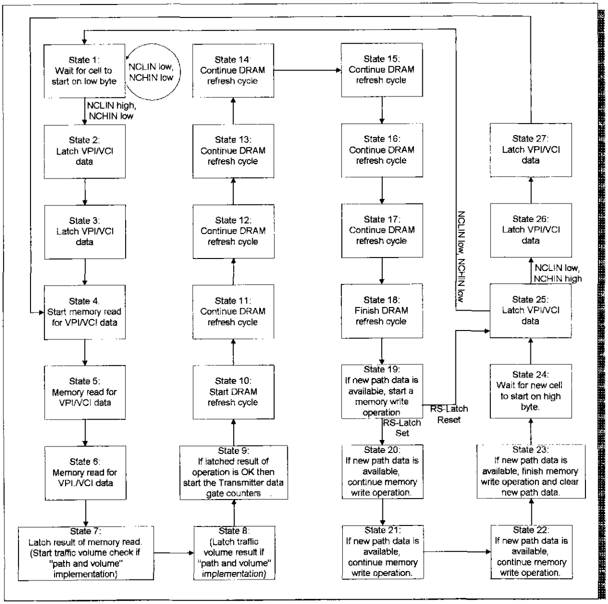
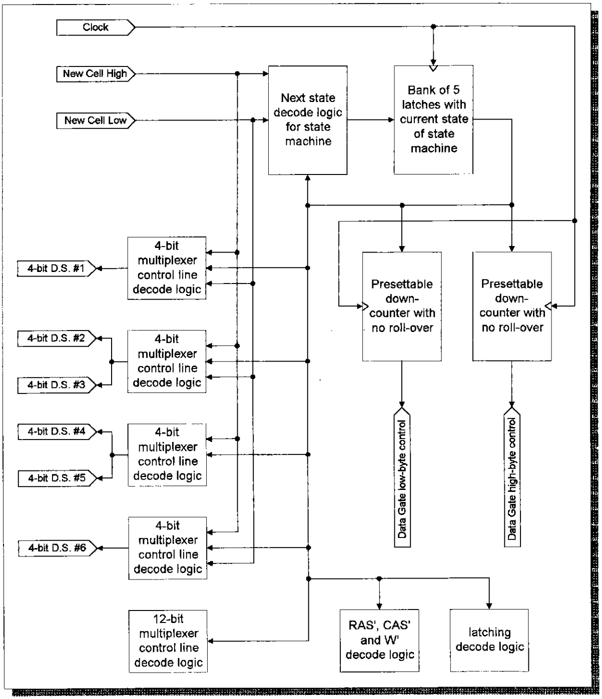

State 1: Wait for cell to start on low byte
NCLIN low, NCHIN low
NCLIN high, NCHIN low
State 2: Latch VPI/VCI data
State 3: Latch VPI/VCI data
State 4. Start memory read for VPI/VCI data
State 5: Memory read for VPI/VCI data
State 6: Memory read for VPI./VCI data
State 7: Latch result of memory read. (Start traffic volume check if "path and volume" implementation)
State 8: (Latch traffic volume result if "path and volume" implementation)
State 9: If latched result of operation is OK then start the Transmitter data gate counters
State 10: Start DRAM refresh cycle
State 11: Continue DRAM refresh cycle
State 12: Continue DRAM refresh cycle
State 13: Continue DRAM refresh cycle
State 14: Continue DRAM refresh cycle
State 15: Continue DRAM refresh cycle
State 16: Continue DRAM refresh cycle
State 17: Continue DRAM refresh cycle
State 18: Finish DRAM refresh cycle
State 19: If new path data is available, start a memory write operation
RS-Latch Set
RS-Latch Reset
State 20: If new path data is available, continue memory write operation.
State 21: If new path data is available, continue memory write operation.
State 22: If new path data is available, continue memory write operation.
State 23: If new path data is available, finish memory write operation and clear new path data.
State 24: Wait for new cell to start on high byte.
State 25: Latch VPI/VCI data
NCLIN low, NCHIN high
State 26: Latch VPI/VCI data
State 27: Latch VPI/VCI data
NCLIN low, NCHIN low

Fig. 24. State diagram for the state machine internal to the sequence / detect module within the Analysis Module

TABLE X
Signal states for every valid state in the state machine diagram for the sequence / detect module of the Analysis Module

State	Value	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	RAS	CAS	W	[illegible]	[illegible]
1	00000	↓↓↓↓↓↓	↓↓↓↓↓↓		↓	↓	↓	↓	↓	↑	↑	↑	↓	↓
2	00001	↓↓↓↓↓↓	↑↑↑↓↓↓		↓	↓	↓	↓	↓	↑	↑	↑	↓	↓
3	00011	↓↓↓↓↓↓	↓↓↓↑↑↑	↑↓	↓	↓	↓	↓	↓	↑	↑	↑	↓	↓
4	00010		↓↓↓↓↓↓	↑↓	↓	↓	↓	↓	↓	↓	↑	↑	↓	↓
5	00110		↓↓↓↓↓↓	↓↓	↓	↓	↓	↓	↓	↓	↑	↑	↓	↓
6	00111		↓↓↓↓↓↓	↓↓	↓	↓	↓	↓	↓	↓	↓	↑	↓	↓
7	00101		↓↓↓↓↓↓		↑	↓	↓	↓	↓	↓	↓	↑	↓	↓
8	00100		↓↓↓↓↓↓		↓	↓	↓	↓	↓	↑	↓	↑	↑	↑
9	01100		↓↓↓↓↓↓		↓	↓	↑	↑	↑	↑	↓	↑	↓	↓
10	01101		↓↓↓↓↓↓		↓	↓	↓	↓	↓	↑	↓	↑	↓	↓
11	01111		↓↓↓↓↓↓		↓	↓	↓	↓	↓	↓	↓	↑	↓	↓
12	01110		↓↓↓↓↓↓		↓	↓	↓	↓	↓	↓	↓	↑	↓	↓
13	01010		↓↓↓↓↓↓		↓	↓	↓	↓	↓	↓	↓	↑	↓	↓
14	01011		↓↓↓↓↓↓		↓	↓	↓	↓	↓	↑	↓	↑	↓	↓
15	01001		↓↓↓↓↓↓		↓	↓	↓	↓	↓	↑	↓	↑	↓	↓
16	01000		↓↓↓↓↓↓		↓	↓	↓	↓	↓	↑	↓	↑	↓	↓
17	11000		↓↓↓↓↓↓		↓	↓	↓	↓	↓	↓	↓	↑	↓	↓
18	11001		↓↓↓↓↓↓		↓	↓	↓	↓	↓	↑	↑	↑	↓	↓
19	11011		↓↓↓↓↓↓	↑↑	↓	↓	↓	↓	↓	↑	↑	↑	↓	↓
20	11010		↓↓↓↓↓↓	↑↑	↓	↓	↓	↓	↓	↓	↑	↑	↓	↓
21	11110		↓↓↓↓↓↓	↓↑	↓	↓	↓	↓	↓	↓	↑	↑	↓	↓
22	11111		↓↓↓↓↓↓	↓↑	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓
23	11101		↓↓↓↓↓↓		↓	↑	↓	↓	↓	↑	↑	↑	↓	↓
24	11100	↑↑↑↑↑↑	↓↓↓↓↓↓		↓	↓	↓	↓	↓	↑	↑	↑	↓	↓
25	10100	↑↑↑↑↑↑	↑↓↓↓↓↓		↓	↓	↓	↓	↓	↑	↑	↑	↓	↓
26	10101	↑↑↑↑↑↑	↓↑↑↑↑↓		↓	↓	↓	↓	↓	↑	↑	↑	↓	↓
27	10111	↑↑↑↑↑↑	↓↓↓↓↓↑		↓	↓	↓	↓	↓	↑	↑	↑	↓	↓

Aside from this central state machine and decode logic blocks guiding the actions of the control signals, there are also some additional logic units that provide for support operation which allows the state machine to continue on with other activities. One such unit is the logic that controls the data gate which feeds a cell's contents to the input stage of the Transmitter (and controls the state of the New Cell High and New Cell Low lines to the Transmitter). The state machine only devotes one state to setting in motion the chain of events which will present an entire cell to the Transmitter. Once the state machine has passed through this state, a count-down with no roll-over counter takes over and continues holding the data gate in pass mode until all of the bytes of the transiting cell have been passed on to the Transmitter.



Clock
New Cell High
New Cell Low
Next state decode logic for state machine
Bank of 5 latches with current state of state machine
4-bit D.S. #1
4-bit multiplexer control line decode logic
4-bit D.S. #2
4-bit D.S. #3
4-bit multiplexer control line decode logic
4-bit D.S. #4
4-bit D.S. #5
4-bit multiplexer control line decode logic
4-bit D.S. #6
4-bit multiplexer control line decode logic
12-bit multiplexer control line decode logic
Presettable down-counter with no roll-over
Presettable down-counter with no roll-over
Data Gate low-byte control
Data Gate high-byte control
RAS', CAS' and W' decode logic
latching decode logic

Fig. 25. Logic blocks which make up the sequence / detect module of the Analysis Module

Other blocks of logic necessary for the correct functioning of the analysis module are those that control the six 4-bit multiplexers which present the VPI/VCI information to the 4-bit latches. These additional blocks are shown in the block diagram of the sequence / detect module in figure 25.

TABLE XI
Next state decode logic for each bit of the state machine controlling the operation of the sequence / detect module

BIT	Expression
b_0	$(b_3)(b_2)(b_1') + (b_4')(b_3)(b_2')(b_1) + (b_4')(b_3')(b_2)(b_1) + (b_3')(b_2')(b_1')(b_0) + (b_3')(b_2')(b_1')(b_0')$(New Cell Low) $+ (b_4)(b_3)(b_2') + (b_4)(b_1')(b_0) + (b_4)(b_3)(b_0') + (b_4')(b_3')(b_1')(b_0')$(New Cell High)
b_1	$(b_3')(b_2')(b_0) + (b_4')(b_3)(b_2)(b_0) + (b_4)(b_3')(b_0) + (b_4)(b_2')(b_1')(b_0) + (b_4)(b_2')(b_1)(b_0)$(Latch Set) $+ (b_1)(b_0')$
b_2	$(b_4')(b_3')(b_2) + (b_4')(b_2)(b_1') + (b_4')(b_2)(b_0) + (b_3')(b_1)(b_0') + (b_4)(b_1)(b_0') + (b_4)(b_3)(b_2) + (b_4)(b_3')(b_1')(b_0')$(New Cell High) $+ (b_4)(b_2')(b_1)(b_0)$(Latch Reset) $+ (b_2)(b_1')(b_0)$
b_3	$(b_4')(b_3) + (b_4')(b_2)(b_1')(b_0') + (b_4)(b_2')(b_1)(b_0)$(Latch Set) $+ (b_3)(b_2)(b_1) + (b_3)(b_1)(b_0') + (b_3)(b_1')(b_0) + (b_3)(b_2')(b_1')$
b_4	$(b_4)(b_3) + (b_4)(b_1')(b_0) + (b_4)(b_3')(b_0')$(New Cell High) $+ (b_3)(b_2')(b_1')(b_0')$

TABLE XII
Decode logic for the control lines which the sequence / detect module uses to operate sub-units of the Analysis Module

BIT	Expression
PVRL	$(b_4')(b_3')(b_2)(b_1')(b_0)$
RSRL	$(b_4)(b_3)(b_2)(b_1')(b_0)$
LLODG26 LHODG27	$(b_4')(b_3)(b_2)(b_1')(b_0')$(New Cell Low)
LLODG27	$(b_4')(b_3)(b_2)(b_1')(b_0')$(New Cell High)
W'	$(b_4') + (b_3') + (b_2') + (b_1') + (b_0')$
RAS'	$(b_4')(b_1')(b_0') + (b_3)(b_1')(b_0) + (b_2')(b_0) + (b_4)(b_3') + (b_4)(b_2)(b_1')$
CAS'	$(b_3')(b_2') + (b_3')(b_1)(b_0') + (b_4)(b_3)(b_1)(b_0') + (b_4)(b_3)(b_1')(b_0) + (b_4)(b_3)(b_2')(b_1)$
VVRL SDDG	$(b_4')(b_3')(b_2)(b_1')(b_0')$(Memory Lookup Result)
4BDS [1…6]	(b_2)
4BDL [0]	$(b_4')(b_3')(b_2')(b_1')(b_0) + (b_4)(b_3')(b_2)(b_1')(b_0')$
4BDL [2…3]	$(b_4')(b_3')(b_2')(b_1')(b_0) + (b_4)(b_3')(b_2)(b_1')(b_0)$
4BDL [4…5]	$(b_4')(b_3')(b_2')(b_1)(b_0) + (b_4)(b_3')(b_2)(b_1')(b_0)$
4BDL [6]	$(b_4')(b_3')(b_2')(b_1)(b_0) + (b_4)(b_3')(b_2)(b_1)(b_0)$
12BDS [B]	(b_2')
12BDS [L]	(b_4)

TABLE XIII
Next state decode logic for each bit of the presettable down-counters with no roll-over in the sequence / detect module

Bit	Expression
b_0	$(b_0' + Set_{27})(b_4 + b_3 + b_2 + b_1 + b_0 + Set_{27})(Set_{26}' + Set_{27})$
b_1	$(b_1' + b_0 + Set_{26} + Set_{27})(b_1 + b_0' + Set_{26} + Set_{27}) (b_4 + b_3 + b_2 + b_1 + b_0 + Set_{26} + Set_{27})$
b_2	$(b_2' + b_1 + b_0) (b_2 + b_1')(b_2 + b_0')(b_4 + b_3 + b_2 + b_1 + b_0)(Set_{26}')(Set_{27}')$
b_3	$(b_3' + b_2 + b_1 + b_0 + Set_{26} + Set_{27}) (b_3 + b_2' + Set_{26} + Set_{27})(b_3 + b_1' + Set_{26} + Set_{27})$ $(b_3 + b_0' + Set_{26} + Set_{27})(b_4 + b_3 + b_2 + b_1 + b_0 + Set_{26} + Set_{27})$
b_4	$(b_3 + b_2 + b_1 + b_0 + Set_{26} + Set_{27})(b_4 + Set_{26} + Set_{27})$

After minimization, table XI shows that the logic necessary to decode the next state of each of the bit inputs to the down-counters, the state machine and the control lines to the other modules is considerable. Also, as is shown in tables XII and XIII, the decode logic for the presettable down-counters and external control lines is not trivial either.

IV.D Control Module Design

The Control Module must do its work asynchronously from the Analysis Modules. It's job is to handle communication with the supervisory interface and with the hardware in the Analysis Modules to which it is attached. To the supervisor interface, the Control Module must report traffic violations detected and read from the Analysis Modules and get information about new valid data paths that have been created in the network. When the Control Module receives data about a new valid path, it must be able to distinguish through which Analysis Module the path passes and must update the valid path information within this module. Implied in this responsibility is the job of maintaining the coherency of all the DRAMs in the Analysis Modules, as well as getting the data from a given security module as quickly as possible once a traffic violation has occurred.

A Motorola 68PM302 Integrated Multiprotocol Processor would be an ideal candidate in this design because of its current availability at reasonable cost and its capability to provide a broad range of built-in features that closely match the needs of this application. It provides sufficient I/O to be able to perform all the necessary read and write operations to and from the Analysis Module hardware. It offers the interrupt circuitry necessary for the Analysis Module to alert the Control Module of a traffic violation. Finally, it provides a high-speed serial interface which could be used in conjunction with a DS1 compliant transceiver in order to communicate with the Control Module's supervisory interface. Those transactions which are considered necessary are:

- Informing the security device by the supervisor hardware that a new VPI/VCI pair is valid on one of the ports of the ATM switch which the device is monitoring.
- Informing the supervisor hardware that a traffic violation has occurred, on which port it has occurred and what the VPI/VCI pair of the offending cell was.

The event which is considered crucial to the operation of the device will be assigned to the interrupt logic of the MC68PM302. This is the presence of a traffic violation on one of the data streams passing through an Analysis Module. Since the timing requirements of the design are so stringent, it would be recommended that no interrupt be shared within these tasks because the time required to perform the additional job of determining exactly which module triggered a particular interrupt is likely to make the processor unable to capture the necessary information in a timely fashion (a timely fashion is defined as one that is performed sufficiently fast that data due to a traffic alarm in one of the Analysis Modules is lost) due to the entry of a new cell into that Analysis Module. Therefore, since the MC68PM302 has 8 interrupt levels, this design should be able to handle the security requirements of traffic originating from an ATM switch with up to 8 outputs (one interrupt per switch output).

While it is not the express purpose of this discussion to describe how the interconnection between the Control Module and Analysis Module should be created, in order to show that it is feasible to have a Control Module consisting of one MC68PM302 controlling up to eight Analysis Modules, one possible arrangement for the assignment of the microcontroller's available external I/O controls follows:

- I/O ports A and C are assigned to load VPI/VCI information into the latches that store data about new connections within the Analysis Modules. This data will become the physical address which will be updated in the dynamic RAM of the Analysis Modules in order to track the new information about that data path.
- I/O port B, upper nibble is assigned to loading the state of a new path into the latches that store data about new connections within the Analysis Modules (this, together with the VPI/VCI information forms the complete data regarding a new path that needs to be set up).
- I/O port B, lower nibble is assigned to select the Analysis Modules whose data needs to be updated. These signals are used to ensure that only one Analysis Module out of those to which the Control Module is attached, will latch the data regarding a new connection which needs to be created.

This interconnection is shown as a block diagram in figure 26. It should be noted that a good portion of the reaction time and efficiency of this module relies on the manner in which the controlling software within the microcontroller behaves. It is possible to code this software in such a way that it reacts to traffic violations extremely quickly while information coming in from the supervisory interface regarding new connections that must be created is allowed to wait a lengthy period of time. Alternatively, the reverse could be true, where new path information is applied to the Analysis Modules very quickly, while traffic violations not always be picked up or may wait for a period of time before being reported to the supervisory interface.

It is beyond the scope of this document to make statements regarding how the software in this module should be coded. The exact mechanics and operating characteristics of the supervisory interface and the format in which it transmits data are only outlined in broad strokes.

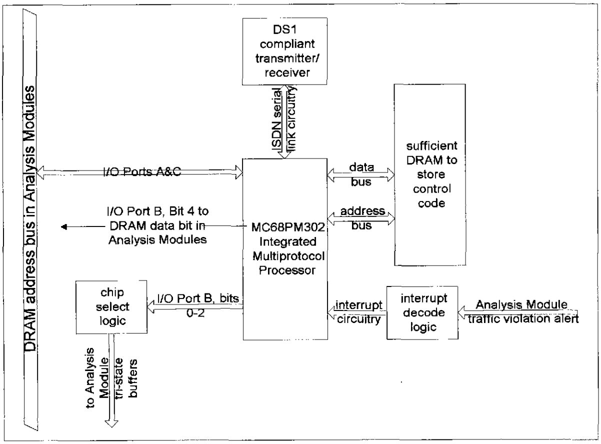

DRAM address bus in Analysis Modules
DS1 compliant transmitter/ receiver
ISDN serial link circuitry
I/O Ports A&C
I/O Port B, Bit 4 to DRAM data bit in Analysis Modules
MC68PM302 Integrated Multiprotocol Processor
data bus
address bus
sufficient DRAM to store control code
chip select logic
I/O Port B, bits 0-2
to Analysis Module tri-state buffers
interrupt circuitry
interrupt decode logic
Analysis Module traffic violation alert

Fig. 26. A possible configuration for using a Motorola 68PM302 microcontroller as a Control Module for multiple Analysis Modules

For the implementation of this Control Module with "path-only" Analysis Modules, the functions described thus far do not require anything more involved than the functionality necessary to receive commands along an ISDN compliant serial communications link, decode it and write the new path information to the registers on the appropriate Analysis Module. Since the only information regarding any virtual path that is required is whether the path is valid or not, there is no prolonged processing involved in order to decide what information needs to be written to the Analysis Module. This means that the time required to begin allowing cells to pass along a newly created valid connection is bounded by only those clock cycles in the 68PM302 necessary to decode the command from the serial link and write it to the Analysis Module.

However, in the case of the Control Module's implementation with "path and volume" Analysis Modules, the situation changes significantly. With the "path and volume" implementation, the data that the Control Module must write to the Analysis Modules is no longer a simple statement describing the validity of a particular path. In this case, this data describes a mapping from the VPI/VCI pair of an arriving cell to one of the "window control modules" on the target Analysis Module. This implies that the Control Module must know, a priori, which "window control modules" on which Analysis Modules have already been assigned to existing paths. When information arrives along the serial link regarding a new connection, the Control Module must be able to determine which "window control module" in the target Analysis Module to map to the new connection.

It becomes evident from this situation that, if the 68PM302's software were to be allowed to handle the mapping of a new path onto a particular "window control module" in the appropriate Analysis Module, then a search algorithm must be implemented in order to ensure that the new path is not being mapped onto a "window control module" already assigned to another path. The table that would need to be searched would contain the state of every "window control module" in a particular Analysis Module. Each element in this table would indicate the assignment state of one particular "window control module". Further, this search space would be a linear function of the number of "window control modules" in each Analysis Module. In order to have this search execute in constant time, the software would be required to create a last-in, first-out (LIFO) queue for every Analysis Module. It is assumed that the operations of pushing and pulling new values onto and from these queues, respectively, would require constant computational time.

The length of each of these queues would be the number of "window control modules" on each Analysis Module. Initially, the Control Module, would completely fill each queue with the indexes of all the "window control modules" on every Analysis Module (indicating that no valid paths exist in any Analysis Module). Then, as new paths are created through commands from the serial link, the 68PM302 would need to pull the first index off the LIFO queue which pertains to the appropriate Analysis Module and write this value to the Analysis Module, along with the VPI/VCI pair of the new connection. In this way, the connection setup time will be constant, since the critical operation of pulling the first value off a LIFO queue is assumed to be accomplished in constant time.

When commands arrive along the serial link which invalidate existing paths, the actions necessary to invalidate the path can no longer be accomplished in constant time according to this approach. In order to invalidate an existing path, the Control Module must first determine to which "window control module", in the appropriate Analysis Module, the path had originally been mapped. Using the design described thus far, this would need to be accomplished through a series of tables which would have been updated at connection setup time. The action of determining which "window control module" has been freed by the path just invalidated would be accomplished by searching this table. Since there are no constraints being placed on the order in which new connections are created and invalidated, then the most optimal search of this table could only be accomplished in computational time that is a linear function of the size of each table, i.e., the number of "window control modules" implemented in each Analysis Module. Once the appropriate "window control module" has been determined, then its index in the appropriate table would be updated and that index would be pushed back onto the LIFO queue which pertains to the Analysis Module on which that "window control module" resides.

However, this linear search time does not imply that the time required to invalidate an existing connection is, itself, linear and not constant. When a command arrives along the serial link to invalidate a path, the command will contain the exact VPI/VCI pair which needs to be invalidated. This means that all of the information necessary to invalidate that path in the appropriate Analysis Module is already available. The 68PM302, would write to that Analysis Module indicating that the VPI/VCI pair in question just maps to a null value (this null value being zero, as described in the Analysis Module's design). Once this is accomplished, then the previously mentioned table search could be performed without the danger that a cell will transit through that Analysis Module along the newly invalidated connection.

Therefore, both new connection creation and invalidation may both be accomplished in constant time within the Control Module's software for the "path and volume" implementation of the Analysis Module.

CHAPTER V

SIMULATION OF SECURITY DEVICE

As already stated, the determining factor in the design was the need to implement the device with circuits of a proven stability and which are inexpensive, in terms of transistor count. Because of the high data rates involved in the transmission of cells in ATM networks, it was necessary to use as much parallelization of functions as possible in hardware in order to implement the design with these stable circuits and at realizable clock speeds.

The objective of this design was to determine whether it is possible to assemble all of the necessary logic units into one or a few monolithic ASICs which will comply with all of the specifications set forth in Chapter 3. Therefore, the deciding factor in the technology chosen for this simulation was that technology which would allow for an accurate determination of the necessary component count and also provide a rough estimate of its operating characteristics for some known value of the signal delay intrinsic in each gate in the circuit. A secondary issue involved was the simulation cost in terms of simulation development time and computational complexity of the simulation itself. Finally, due to the nature of the conclusions to be drawn, the simulation had to be free of logic family specific manufacturing issues. One such example is the difference in the importance of correct transistor sizing between different logic families. In bipolar logic families the sizing of individual transistors is much more important than it would be in a design relying on FET technologies due to the large difference in bipolar base currents drawn versus those of field-effect transistor gates. Another example would be the differences in component counts which may be mounted on an emitter-coupled logic die versus other logic families. Since the devices in emitter coupled logic circuits are not intended to ever be driven into their saturation region during normal circuit operation, these circuits typically reach the die package heat dissipation limits at much lower component counts than would similar circuits in other logic families.

In order to cover these issues, the simulation was laid out as a behavioral description of the circuits involved, with individual logic gates as the atomic element. The design of the entire security device was simulated using the Verilog hardware description language in order to verify that the circuit indeed performs its intended function. The actual circuit is expected to operate with a clock period of 25.88 nanoseconds. For the purposes of this implementation, this clock period was approximated to 26 nanoseconds, in order to make simulation of the device less computationally intensive. This approximation may be found in the Verilog definition of the "ClockGen" module.

The Receivers, Transmitters, Control Module and dynamic RAM of the device use a purely behavioral description. The procedure used for the Receivers, Transmitters and Control Module is a direct extension of the earlier discussion regarding their design. The procedure used for the dynamic RAM is drawn from the manufacturer's product data and implements the functions of "Read with Hidden Refresh" and "Write" according to the timing specifications described therein.

The Analysis Module was modeled as a series of behaviorally described "NAND" and "NOR" gates whose operation is assumed to be ideal except for a known, fixed signal propagation delay. As shown in the source code included in Appendices A and B, this propagation delay is set to one nanosecond, this is to say, 1/26 of the clock cycle time.

Based on these gate description, all of the remaining sub-modules necessary to construct the Analysis Module were simulated, with latches and multiplexers appearing as the most basic building blocks and continuing all the way up to complete state machines at the highest orders of complexity.

The results of this simulation indicate that the device will, indeed, perform its function satisfactorily for a range of gate delays, with the highest acceptable delay being 2/26 of the clock period of the Analysis Module. The device operation breaks down at some point between a gate delay of 2/26 and 3/26 of the clock period.

The entire simulation for both versions of the Analysis Module was implemented with a modular approach in order to make debugging, testing and compilation feasible. The resulting simulation consists of a large number of functional module units which are interdependent among themselves. These modules and the submodules upon which they depend are shown in table XIV for the "path-only" version of the Analysis Module.

The logic which makes up each of the individual blocks of logic described in the design sections were grouped as closely as possible within one complete circuit module with the same name. This was not precisely possible in all cases due to the interdependence of similar reusable blocks that could be used as submodules for different design components. However, every simulation module is a faithful representation of the exact circuit logic and Boolean expressions described in the section concerning the design of the device.

Component count optimizations within each of the logic blocks were implemented as far as possible without having the logic block deviate from the circuit described in the design section. While a careful analysis will reveal certain optimizations still left unimplemented, these optimizations will have a non-significant impact on the overall component count of the entire device. For example, the number of literals in the decode logic of many of the counters could have been reduced somewhat by implementing the states of the counters as Gray-code counters. However, this optimization would reduce overall component count by less than one percent.

The correctness of the simulation's operation, and the subsequent inference that the device, if constructed would operate properly, relies on the generation of test pattern cell arrivals and new path updates from the modules entitled "ControlModule" and "Receiver". Both of these modules are behaviorally defined with the "Receiver" logic block generating an alternating sequence of non-unique complete cell arrivals and the "ControlModule" generating a continuous stream of new path updates for loading into the device's memory.

For the purposes of this simulation, it was not feasible to implement the entire sixteen megabit memory space of all of the memories involved. Instead, in the "path-only" version of the Analysis Module, the dynamic RAM's procedural definition specifies that it will recognize a cell's path as being valid if the last bit of the address presented to it is asserted. In the "path and volume" version of the Analysis Module, an arriving cell's path will be mapped to an existing path if the lower three bits of the address correspond to a sequence that has already been generated by the "ControlModule".

It is evident that some simplifications had to be made in order to encapsulate the entire design into the simulation environment chosen, however, the artificial data sequence created by the test modules simulated show cases of the device operating under all possible combinations of circumstances. This is to say that the data streams generated by the "ControlModule" and the "Receiver" force the Analysis Module to show its behavior both when a valid cell arrives and when an invalid cell arrives, on either starting byte of the "Receiver" module's output. In addition, it shows that the Analysis Module will correctly load new path data within one cell time of the "ControlModule" block's having presented it.

TABLE XIV
Module names and the submodules of which they consist for the simulation of the "path-only" Analysis Module

Module Name	Sub-module Components
Inverter	Procedurally defined.
TwoInputNANDGate	Procedurally defined.
ThreeInputNANDGate	Procedurally defined.
FourInputNANDGate	Procedurally defined.
FiveInputNANDGate	Procedurally defined.
SixInputNANDGate	Procedurally defined.
SevenInputNANDGate	Procedurally defined.
NineInputNANDGate	Procedurally defined.
TwoInputNORGate	Procedurally defined.
ThreeInputNORGate	Procedurally defined.
FourInputNORGate	Procedurally defined.
FiveInputNORGate	Procedurally defined.
SixInputNORGate	Procedurally defined.
SevenInputNORGate	Procedurally defined.
SRLatch	2•(TwoInputNANDGate)
PosEdgeTrigLatch	5•(TwoInputNANDGate) + 1•(ThreeInputNANDGate)
FourBitRegister	4•(PosEdgeTrigLatch)
EightBitRegister	8•(PosEdgeTrigLatch)
SixteenBitRegister	16•(PosEdgeTrigLatch)
ShiftRegister	9•(SixteenBitRegister)
DataGate	1•(TwoInputNANDGate) + 1•(Inverter)
FourBitDataGate	4•(DataGate)
EightBitDataGate	2•(FourBitDataGate)
TwoLineSelector	3•(TwoInputNANDGate)
FourBitTwoLineSelector	4•(TwoLineSelector) + 1•(Inverter)
FourLineSelector	4•(ThreeInputNANDGate) + 1•(FourInputNANDGate)
TwelveBitFourLineSelector	12•(FourLineSelector) + 2•(Inverter)
ClockGen	Procedurally defined.
NewPathStore	3•(EightBitRegister) + 1•(PosEdgeTrigLatch) + 1•(SRLatch)
DynamicRAM	Procedurally defined.
NetworkReceiver	Procedurally defined.
NetworkTransmitter	Procedurally defined.
ControlModule	Procedurally defined.
DownCounterWithPreset	5•(PosEdgeTrigLatch) + 2•(Inverter) + 5•(TwoInputNORGate) + 3•(ThreeInputNORGate) + 5•(FourInputNORGate) + 2•(FiveInputNORGate) + 4•(SixInputNORGate) + 2•(SevenInputNORGate)

TABLE XIV
(continued)

Module Name	Submodule Components
StateControl	5•(Inverter) + 7•(TwoInputNANDGate) + 4•(ThreeInputNANDGate) + 3•(FourInputNANDGate) + 11•(FiveInputNANDGate) + 2•(SevenInputNANDGate)
StateMachine	5•(PosEdgeTrigLatch) + 5•(ResetControl) + 3•(TwoInputNANDGate) + 18•(ThreeInputNANDGate) + 9•(FourInputNANDGate) + 6•(FiveInputNANDGate) + 1•(SevenInputNANDGate) + 2•(NineInputNANDGate)
SequenceDetect	1•(StateMachine) + 1•(StateControl) + 2•(DownCounterWithPreset) + 2•(FiveInputNANDGate)
NetworkSecurity	1•(ClockGen) + 1•(NetworkReceiver) + 1•(NetworkTransmitter) + 1•(ControlModule) + 1•(ShiftRegister) + 2•(EightBitDataGate) + 1•(SequenceDetect) + 1•(NewPathStore) + 6•(FourBitTwoLineSelector) + 6•(FourBitRegister) + 1•(TwelveBitFourLineSelector) + 1•(DynamicRAM)

The modules and the submodules upon which they depend are shown in table XV for the "path and volume" version of the Analysis Module.

It must be noted that, to simulate the entire device as one complete unit requires a very considerable amount of computation time. The device's simulation code presented here had, itself, to be broken down into component sections. Each of those sections were simulated with a known generator pattern of signals, their results captured and then passed along to the standalone simulation of the next logic block in the sequence.

However, the Verilog code presented here allows a designer to look at the behavior of the signals of every phase of the design in order to analyze where improvements could be made. Therefore, the simulation's purpose as a "proof of concept" has been realized.

TABLE XV
Module names and the submodules of which they consist for the simulation of the "path and volume" Analysis Module

Module Name	Sub-module Components
Inverter	Procedurally defined.
TwoInputNANDGate	Procedurally defined.
ThreeInputNANDGate	Procedurally defined.
FourInputNANDGate	Procedurally defined.
FiveInputNANDGate	Procedurally defined.
SixInputNANDGate	Procedurally defined.
SevenInputNANDGate	Procedurally defined.
NineInputNANDGate	Procedurally defined.
TwoInputNORGate	Procedurally defined.
ThreeInputNORGate	Procedurally defined.
FourInputNORGate	Procedurally defined.
FiveInputNORGate	Procedurally defined.
SixInputNORGate	Procedurally defined.
SevenInputNORGate	Procedurally defined.
SRLatch	2•(TwoInputNANDGate)
PosEdgeTrigLatch	5•(TwoInputNANDGate) + 1•(ThreeInputNANDGate)
FourBitRegister	4•(PosEdgeTrigLatch)
EightBitRegister	8•(PosEdgeTrigLatch)
SixteenBitRegister	16•(PosEdgeTrigLatch)
ShiftRegister	9•(SixteenBitRegister)
DataGate	1•(TwoInputNANDGate) + 1•(Inverter)
FourBitDataGate	4•(DataGate)
EightBitDataGate	2•(FourBitDataGate)
TwoLineSelector	3•(TwoInputNANDGate)
FourBitTwoLineSelector	4•(TwoLineSelector) + 1•(Inverter)
FourLineSelector	4•(ThreeInputNANDGate) + 1•(FourInputNANDGate)
TwelveBitFourLineSelector	12•(FourLineSelector) + 2•(Inverter)
ClockGen	Procedurally defined.
NewPathStore	3•(EightBitRegister) + 7•(PosEdgeTrigLatch) + 1•(SRLatch)
DynamicRAM	Procedurally defined.
NetworkReceiver	Procedurally defined.
NetworkTransmitter	Procedurally defined.
ControlModule	Procedurally defined.
DownCounterWithPreset	5•(PosEdgeTrigLatch) + 2•(Inverter) + 5•(TwoInputNORGate) + 3•(ThreeInputNORGate) + 5•(FourInputNORGate) + 2•(FiveInputNORGate) + 4•(SixInputNORGate) + 2•(SevenInputNORGate)

TABLE XV
(continued)

Module Name	Submodule Components
CounterGate	2•(Inverter) + 4•(ThreeInputNANDGate) + 1•(FourInputNANDGate)
BitEqualTest	3•(TwoInputNANDGate)
ThreeBitDataGate	3•(DataGate)
ThreeBySevenDemux	10•(Inverter) + 7•(ThreeInputNANDGate)
ResetControl	Procedurally defined.
CounterWithZeroTest	4•(PosEdgeTrigLatch) + 4•(CounterGate) + 6•(TwoInputNANDGate) + 6•(TwoInputNORGate) + 3•(ThreeInputNANDGate) + 3•(ThreeInputNORGate) + 2•(FourInputNANDGate) + 3•(FourInputNORGate)
CounterWithReset	4•(PosEdgeTrigLatch) + 4•(DataGate) + 3•(Inverter) + 5•(TwoInputNANDGate) + 3•(ThreeInputNANDGate)
WindowCounter	4•(PosEdgeTrigLatch) + 1•(CounterWithReset) + 4•(BitEqualTest) + 1•(Inverter) + 1•(FourInputNANDGate)
WindowControl	1•(WindowCounter) + 1•(CounterWithZeroTest) + 1•(Inverter)
StateControl	5•(Inverter) + 7•(TwoInputNANDGate) + 4•(ThreeInputNANDGate) + 3•(FourInputNANDGate) + 11•(FiveInputNANDGate) + 2•(SevenInputNANDGate)
StateMachine	5•(PosEdgeTrigLatch) + 5•(ResetControl) + 3•(TwoInputNANDGate) + 18•(ThreeInputNANDGate) + 9•(FourInputNANDGate) + 6•(FiveInputNANDGate) + 1•(SevenInputNANDGate) + 2•(NineInputNANDGate)
SequenceDetect	1•(StateMachine) + 1•(StateControl) + 2•(DownCounterWithPreset) + 2•(FiveInputNANDGate)
NetworkSecurity	1•(ClockGen) + 1•(NetworkReceiver) + 1•(NetworkTransmitter) + 1•(ControlModule) + 1•(ShiftRegister) + 2•(EightBitDataGate) + 1•(SequenceDetect) + 1•(NewPathStore) + 6•(FourBitTwoLineSelector) + 6•(FourBitRegister) + 1•(TwelveBitFourLineSelector) + 3•(DynamicRAM) + 3•(PosEdgeTrigLatch) + 1•(ThreeBitDataGate) + 2•(ThreeBySevenDemux) + 7•(WindowControl) + 2•(Inverter) + 7•(TwoInputNANDGate) + 1•(SevenInputNORGate)

For the designer's reference, the entry point into the simulation (the highest level block of integration) for both versions of the Analysis Module is the "NetworkSecurity" Verilog module. Additionally, the "path and volume" version of the Analysis Module simulated implements seven window control modules with each window control module having a granularity of four bits. This means that the "path and volume" Analysis Module described in the Verilog simulation is capable of supporting up to seven valid connections and that the leaky bucket traffic meter on each connection will support a traffic credit system with a maximum of sixteen credits per connection and that the lowest credit generation rate possible will be one credit every sixteen clock cycles.

If, at some future date, it is necessary to extend this simulation to support more simultaneous connections, it is only necessary to add more traffic control modules (and, the appropriate number of dynamic RAMs) to the "NetworkSecurity" module in the Verilog source code. However, the changes necessary to change the granularity of the window control modules will be more extensive since this will involve changes, not only to the counters that manage the credit system within these window control modules but also to the latches that control how often to generate a credit. Not to be excluded from these changes, are the equality testers that check when it is time to generate a new credit and when a connection traffic volume has overflowed. All of the changes necessary to change the granularity of the credit system would be in the "WindowControl" module of the Verilog source code.

CHAPTER VI

PERFORMANCE ISSUES

The digital circuits assembled indicate that this design can correctly handle traffic from all of the ATM forum data rate specifications. These calculations were made using worst case network traffic assumptions with full traffic violation rates. This means that the basic assumption regarding traffic arrival characteristics were that no link bandwidth was being left unutilized and that the arriving traffic could be either completely invalid for all arriving cells or completely valid for all arriving cells.

All of the components mentioned in this design can easily be implemented in the TTL (transistor-transistor logic), ECL (emitter-coupled logic) and HC (high-speed CMOS) logic families as evidenced by the range of products available in any catalog from the major digital applications semiconductor manufacturers. The external microcontrollers and dynamic memories have been available for considerable periods of time and, thus, are considered to be very stable from the point of view of reliability of operating characteristics. Therefore, this design should be feasible utilizing only standard, off the shelf components for the implementation of three of the major components of the design which are not specifically laid out in this document: Receivers, Transmitters and Control Module. The Analysis Module should be implementable through current one-micron and sub-micron production procedures coupled with current VLSI design tools. Again, current product literature allows for the conclusion that two, three and four million transistor count microchip designs are feasible on a scale that allows for mass manufacturing [31, 32].

In order to assess the feasibility of the implementation of the design of this device, it is necessary to establish what the approximate transistor counts for the various versions of the device will be, as well as, the maximum gate delays necessary to make the device feasible. In order to accomplish both of these measurements, the circuits described in the design portion of this document will be used. Approximate transistor counts will be reached by counting the gates necessary to implement these circuits and maximum gate delays necessary will be calculated by finding the longest series chain of gates through which a signal must pass in any one clock cycle and still allow the device to accomplish its function correctly.

The design of the Analysis Module was described to be as logic family independent as possible. While it is not feasible to use the exact same circuit to perform the necessary functions in all of the logic families, the circuit, as described, could be implemented in all of the logic families and be quite close to optimal in component count. In the RTL (archaic), DTL (archaic), TTL and ECL logic families, the circuit description is very close to optimal. In the CMOS and High Speed CMOS logic families, the circuit description could vary somewhat due to the availability of very low part count latches within these logic families. However, these low part count latches, are of the level-triggered variety and would require additional logic in order to ensure device stability. Therefore, an assumption involved in using the part count estimates described here for the CMOS logic family implementations is that the additional logic necessary to account for the level-triggered nature of CMOS latches would balance out the transistors lost by using these lower component count latches.

In an attempt to calculate the necessary component count in a way that is independent of a particular logic family implementation, this component count will be assessed based on gate counts with each gate being assigned a component weight based on the number of inputs. Since all of the logic families share the common characteristic that the transistor count necessary to implement an "n" input logic gate is directly and linearly proportional to "n" (the number of inputs to the gate), we can accurately approximate the component count by summing the weight of each gate used. This sum of input-normalized gate weights would then be multiplied by a constant in order to predict the component count for the device's construction within each logic family. For RTL (the most primitive of the logic families; largely archaic) this multiplication constant would be exactly one, since one transistor is required for every gate input. For the CMOS families, this multiplication constant would be approximately two and the TTL/ECL families would fall somewhere in between [33].

The component count for the "path-only" version of the Analysis Module will remain fixed for all situations since the design, as presented, has sufficient capabilities to support invalid cell suppression for arrivals with any path information. However, the additional circuitry necessary to implement the "path and volume" version of the Analysis Module is significant and has the ability to grow to an untenable component count. In order to keep this version of the design within a reasonable component count, the component weights are calculated based on two variables. The number of "window control modules" and size of the window control module demultiplex selector in this version of the design is in direct relation to the number of valid paths for which the Analysis Module may provide traffic volume verification. Therefore, the first variable in the component count for the design of the "path and volume" Analysis Module will be the number of valid connection paths supported by the Analysis module which will be referred to as "W".

The granularity with which the leaky bucket mechanism in the "path and volume" Analysis Module can verify traffic along each valid connection path is directly related to the number of bits in the internal states of the two counters, the size of the input words of the equality testers and the size of the storage register which compose the window control modules. Therefore, the second variable in the component count for the design of the "path and volume" Analysis Module will be the granularity supported by each leaky bucket mechanism controlling each valid connection which will be referred to as "N".

TABLE XVI
Composition and component weight of the modules in the "path-only" version of the Analysis Module

Module Name	Gate Composition	Component Weight
4-bit multiplexer for loading cell header data (six units)	72•(Two Input) + 6•(One Input)	150
4-bit latch for storing cell header data (six units)	24•(Three Input) + 120•(Two Input)	312
12-bit by 4-line multiplexer for presenting data to memory	12•(Four Input) + 48•(Three Input) + 2•(One Input)	194
12-bit latch for new path data (two units)	24•(Three Input) + 120•(Two Input)	312
1-bit latch for new path data state	1•(Three Input) + 5•(Two Input)	13
latch for memory lookup result	1•(Three Input) + 5•(Two Input)	13
SR-latch for status of new path registers	2•(Two Input)	4
memory lookup module for path verification	External Unit	0
alert latch for result from memory lookup	1•(Three Input) + 5•(Two Input)	13
sequence / detect module for overall control	9•(One Input) + 95•(Two Input) + 43•(Three Input) + 22•(Four Input) + 23•(Five Input) + 8•(Six Input) + 7•(Seven Input) + 2•(Nine Input)	646
12-bit shift register with 9 stages for cell data transit area	720•(Two Input) + 144•(Three Input)	1872
data gate for cell output control	16•(One Input) + 16•(Two Input)	48
TOTAL		3577

As shown in table XVI and table XVII, the part count in the Analysis Module is sufficiently low to lend itself to VLSI implementation only if the "path-only" version is implemented or if the "path with volume" version is implemented with a limited number of window control modules. As the analysis shows, the "path-only" version could be implemented with a component weight of only three to four thousand, which is trivial by modern VLSI standards. However, it is evident that for the "path and volume" version, the component weight depends heavily on the number of window control modules implemented and their associated granularity. In fact, there is a square relationship between the component weight and the granularity of each window control module while there is a linear times log relationship between the component weight and the number of window control modules implemented.

TABLE XVII

Composition and component weight of the sub-modules composing one "window control module" used in the "path and volume" version of the Analysis Module

Module Name	Gate Composition	Component Weight
"N"-bit register	5N•(Two Input) + N•(Three Input)	13N
"N"-bit counter (with reset)	N•(One Input) + ½(N^2+13N+4)•(Two Input) + (N)•(Three Input) + 2•($\Sigma_{3...N}$ Input)	$2N^2 + 16N$
"N"-bit by "N"-bit equality tester	3N•(Two Input) + 1•("N" Input)	7N
"N"-bit counter with control	(N^2+4N+6)•(Two Input) + 5N•(Three Input) + N•(Four Input) + 6•($\Sigma_{3...N}$ Input) + 2N•("N" Input)	$6N^2 + 23N + 4$
"N"-bit zero tester	1•("N" Input)	N
TOTAL		$8N^2 + 60N + 4$

However, the component *weights required to implement* the "path and volume" version of the Analysis Module are not so great as to make them unfeasible at current VLSI densities. The governing relationships necessary to calculate the component weight of the "path and volume" implementation as a function of the number of window control modules added and their associated granularity is are shown in table XVIII. With a component weight of one million, it is feasible to implement 100 window control modules with each module having a granularity of 32 bits. If the component weight is allowed to grow to two million, then it becomes feasible to implement 200 window control modules with 32 bits of granularity apiece. Also, it should be noted that if the window control module granularity is halved, the corresponding number of modules which can be added to keep the component weight at the same level more than doubles. To extend the example, if the window control module granularity is reduced to 16 bits, then 329 window control modules may be placed within an Analysis Module at a component weight of one million with this figure growing to 658 window control modules at a component weight of two million. Table XIX shows the order of magnitude correlation between the number of window control modules constructed within a "path and volume" Analysis Module, their associated granularities and the resulting component weight of that Analysis Module.

Since the component weights of one million and two million components correspond to an actual transistor count of up to two to four million transistors, respectively (depending on the logic family used for its implementation) it is evident that these design goals are not unrealistic.

TABLE XVIII
Composition and component weight of the modules in the "path and volume" version of the Analysis Module

Module Name	Gate Composition	Component Weight
4-bit multiplexer for loading cell header data (six units)	6•(One Input) + 72•(Two Input)	150
4-bit latch for storing cell header data (six units)	120•(Two Input) + 24•(Three Input)	312
12-bit by 4-line multiplexer for presenting data to memory	2•(One Input) + 48•(Three Input) + 12•(Four Input)	194
12-bit latch for new path data (two units)	120•(Two Input) + 24•(Three Input)	312
n-bit latch for new path data state (sufficient bits to load a word describing a unique window control module)	5log$_2$(W)•(Two Input) + log$_2$(W)•(Three Input)	13(log$_2$ W)
latch for memory lookup result (sufficient units to store a word describing a unique window control module)	5log$_2$(W)•(Two Input) + log$_2$(W)•(Three Input)	13(log$_2$ W)
SR-latch for status of new path registers	2•(Two Input)	4
Memory lookup module for path verification	External Unit	0
Demultiplexer to select window control module (sufficient outputs to select one of all window control modules – two units: load new data and react to a memory lookup)	2W•(log$_2$(W) Input)	2W(log$_2$ W)
Alert latch for result from memory lookup (sufficient bits to load a word describing a unique window control module)	5log$_2$(W)•(Two Input) + log$_2$(W)•(Three Input)	13(log$_2$ W)
Window control module with "N" bits of granularity (W units)	N•(One Input) + ½(3N^2 + 37N + 4)•(Two Input) + 7N•(Three Input) + N•(Four Input) + 8•($\Sigma_{3\ldots N}$ Input) + (2N + 2)•("N" Input)	8WN^2 + 60WN + 4W
sequence / detect module for overall control	9•(One Input) + 95•(Two Input) + 43•(Three Input) + 22•(Four Input) + 23•(Five Input) + 8•(Six Input) + 7•(Seven Input) + 2•(Nine Input)	646
12-bit shift register with 9 stages for cell data transit area	720•(Two Input) + 144•(Three Input)	1872
data gate for cell output control	16•(One Input) + 16•(Two Input)	48
TOTAL		W(8N^2 + 60N + 4) + (2W + 39)(log$_2$ W) + 3538

TABLE XIX
Order of magnitude correlation of the component weight of the "path and volume" Analysis Module as the number and granularity of "window control modules" implemented varies

Parameter	Δ (Component Weight)
# of window control modules	O(W log W)
granularity of each window control module (expressed in number of bits)	$O(N^2)$

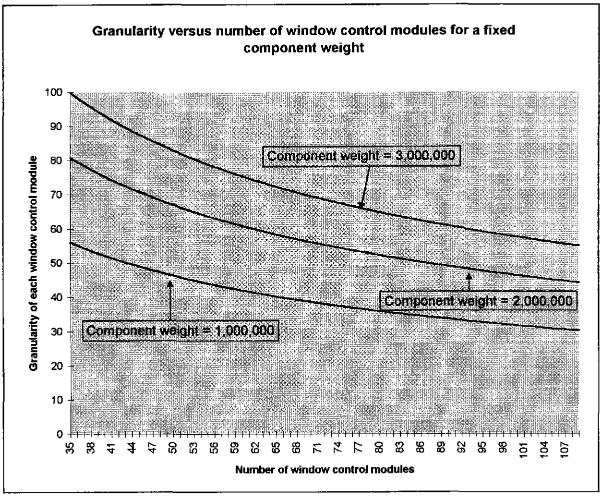


Fig. 27. Granularity versus number of window control modules which may be implemented in one Analysis Module for fixed component weights

Therefore, it has been shown that the component weight necessary to implement the device is significantly impacted by the granularity of the "window control modules" placed on each Analysis Module. Figure 27 shows this in graphical form as the design component weights begin in increase exponentially if the window control module granularity is increased linearly. Figure 28 presents this component weight information as a function of the number of window control modules implemented and allows the conclusion that the component weight is a linear function of the number of window control modules.

All "window control modules", regardless of their granularity, exhibit an upper limit on the traffic volume they will permit to pass of one cell credit per clock cycle, which translates to one credit per cell time (if the Analysis Modules' clock is divided by 26.5 for all window control modules). However, their granularity affects the minimum allowable traffic rate per connection, as well as, the greatest number of traffic credits any one connection is allowed to accumulate when that connection is utilizing less than its declared allowable bandwidth. Therefore, with "window control modules" of greater granularity, the Analysis Module is capable of successfully controlling virtual connections with lower traffic limits and, also, of allowing uninterrupted traffic flow for connections with "bursty" traffic patterns. All of this is possible while still verifying that they do not exceed their allowable "mean" traffic limits. Both of these characteristics are favorable to supporting the wide range of traffic types envisioned for the distributed nature of wide-area backbones [34, 35].

The remaining issue which pertains to the components necessary to implement this design are those of the amount of dynamic RAM memory that will be required off-chip for the Analysis Module. In the case of the "path-only" Analysis Module, it is only necessary to place one 16 megabit RAM in the circuit in order to support complete screening of all possible connection paths. However, in the "path and volume" implementation, the amount of memory which will be required will be a function of the number of valid paths which must be supported by each Analysis Module. To be more precise, sufficient memory will be required in order to generate a data word wide enough to support the selection of one unique window control module for any random address within a 24-bit address space. Therefore, in the case where "W" window control modules have been implemented within an Analysis Module, a data word with a width of $\log_2(W)$ will be required in order to select one of them. By extension, this means that $\log_2(W)$ memories of 16 megabits apiece will be necessary to support an Analysis Module with "W" window control modules. Therefore $2\bullet\log_2(W)$ megabytes of memory would be required by this design (with one byte equaling to eight bits). If we were to place 16 megabytes of memory in one Analysis Module, this would allow for the support of 256 window control modules. Likewise, 8 and 4 megabytes in each Analysis Module would support 128 and 64 window control modules, respectively. These memory ranges are not unreasonable, given the current market availability of these components.

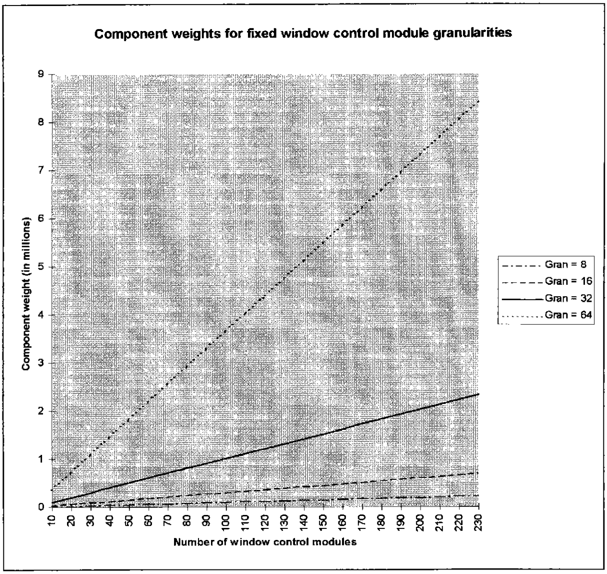


Fig. 28. Component weight versus number of window control modules which may be implemented in one Analysis Module for fixed module granularities

Up to this point, it has been shown that the component count of this security device is significantly impacted by the number of simultaneous network virtual connections the device will support. This impact it so great that for security devices connected to nodes through which a large amount of traffic passes, the number of simultaneous connections could very well exceed the number of components that may feasibly be mounted on one or a few dies. Likewise, it does not make sense to make the investment to develop a high component chip only to install it into a security device that monitors a gateway to the backbone where only a few connections may simultaneously exist. Therefore, an approach should be discussed by which an extensible version of this security device may be implemented. Extensibility of the chip in this design refers to an implementation that uses this same core of design decision in such a fashion that multiple identical devices may be interconnected to operate as one device which can handle a greater number of simultaneous virtual connections than any one chip would normally be able to.

The issue at the heart of creating a series of devices which can behave as one is to divide the set of virtual connections which may exist simultaneously among different units. In this way, every individual unit can test incoming traffic for validity or volume violations and only forward that portion of the traffic found to belong to a valid connection, for which it is responsible, to the network backbone. That traffic for which a particular module is not responsible will be forwarded to the next security unit in the sequence. This extensible approach is described graphically in figure 29.

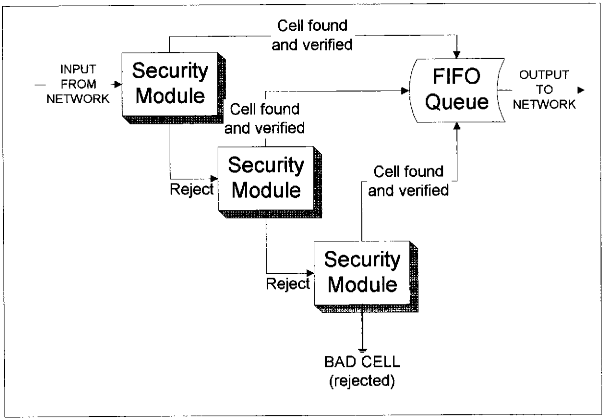


Fig. 29. High level view of the interconnections of Security Modules in a simple extensible implementation

Using this approach, every security module is responsible only for those cells belonging to connections that are found within its own window control modules. The additional hardware necessary to implement such an approach consists only of a first-in first-out (FIFO) queue which would capture those cells that are found to be valid and within volume limits by any of the Security Modules and forward them to the network. Since, in the worst case, cells will be fed into the first Security Module in the chain at the network's peak transmission rate, and the interconnections between the Security Modules will pass these cells to one another at this same rate, at most one cell may exist within one Security Module at any given time. Due to this, the greatest number of cells that may be passed to the FIFO queue is the same as the number of Security Modules to which it is connected. Therefore, the FIFO queue's depth need be no greater than the number of Security Modules to which it is connected.

The impact on the overall performance of the device in terms of cell delay time are significant. In the best case, the cell will be found to be valid and to be within volume limits within the first Security Module to which it is transmitted. In this case the cell will experience one-half cell time delay within that module and negligible delay within the FIFO queue (assuming it is empty). Therefore, in the best case, cell delay experienced within this extensible configuration will be the same as that of the non-extensible device. In the worst case, a cell will not be found to be valid until it reaches the last Security Module in the chain. Also, when that cell is finally transmitted to the FIFO queue, it may experience additional delays due to cells that may already be in that queue. Since every Security Module delays a cell by one-half of a cell time and every cell already in the FIFO queue will delay that cell by an additional cell time, the worst-case delay a valid cell may experience while travelling though this device will be $\left(\frac{N}{2}+N-1\right)$ cell times for N Security Modules. Therefore, if there were four Security Modules chained together outputting their valid data to a FIFO queue with a depth of four cells, the worst-case cell delay would be five cell times.

The Control Module in this configuration would have to oversee the operation of a number of Security Modules for every data path, instead of just one, as in the non-extensible configuration. In order to determine whether or not a cell truly belongs to an invalid path, it would have to correlate the invalid cell alarms from all of the Security Modules along one data path together. However, as in the non-extensible configuration, a connection volume violation alarm from any one of the Security Modules will suffice in order to detect a traffic volume violation. Finally, in order to create new valid connections along any one data path, the Control Module will not only have to determine into which memory slot to place the connection, but it will also have to select one of the Security Modules along that data path first. This decision will further be complicated by the fact that the cells belonging to connections which are tracked in the first Security Module in the chain will experience a smaller delay than those being tracked in the last Security Module in the chain.

Therefore, before adopting this extensible configuration, the fact that cell delay in these security devices will be a linear function of the number of Security Modules used in each data path should be duly noted.

CHAPTER VII

CONCLUSION

The primary goals of this thesis work were to create a design by which basic covert traffic minimization mechanisms could be implemented in hardware with the scope of providing a mechanism for uniform security enforcement across a wide-area ATM/SONET technology network backbone.

A module level description of the device has been presented and shown to be implementable with currently available off-the-shelf components and custom application specific integrated circuitry (ASIC) available at current levels of integration technology. The performance of the device has been evaluated under worst-case conditions for network traffic. Is has been shown that the delay experienced by network traffic in existing virtual connections in the network is trivial when compared to its expected transit time within the network and that the management functions of creating and destroying virtual connections are not a function of the creation / destruction rate of these connections. Through the description of its operation, it is evident that, while utilizing such a framework of traffic security enforcement, the full bandwidth of the network is available to all users for authorized utilization and that through traffic delays network cells will experience are constant even under sustained peak traffic conditions.

The possibility of implementing fixed-window leaky bucket traffic control mechanisms, whether for actual security enforcement purposes or others, was actually shown to be feasible. While actual performance measurements on the correlation between the "burstiness" of connection traffic and size of the leaky bucket mechanism window have not been taken, this information is amply documented in [36] and [37]. Even though no guidelines have been given with regard to the window size of the leaky bucket mechanism that should be implemented, there is sufficient research to allow for an educated decision with regard to the tradeoff between the component count of the ASIC that would need to be implemented and the "burstiness" of the connection traffic that should be allowed to be admitted through the network.

The device was tested by simulation for proper operation with the Verilog hardware description language and, according to its specifications, and was found to meet its design goals for any design whose gate delays are less than two nanoseconds. While integrated circuit gate delays are highly logic family dependent, this requirement should not be a significant hindrance to the implementation of the ASIC since large microcontroller designs have already been shown to have the capacity to operate at clocking speeds in excess of thirty-eight MHz (the intended clocking speed for this device).

VII.A Future Work

The details for the components of the security framework presented here have concentrated primarily on the mechanisms by which actual enforcement should occur and how to limit the impact which it has on overall network performance. Many portions of the larger issues of this method of security enforcement have been glossed over. Foremost among these issues is the topology and physical architecture which should be used to implement the network by which supervisory control data is transferred between the modules that actually provide the enforcement and the workstations which keep the operators of the security body appraised of the state of the network. Toward this end, a significant amount of work lies ahead in order to assess which topologies and implementation technologies would be optimal for this overlying network. An integral component of this decision will be an assessment of exactly what criteria to use in order to derive the level of enforcement that the modules designed in this document will be required to perform. Based on this, assessments may be made with regard to what the overall bandwidth and worst-case delays of the overlying network must be in order to provide an interface to the individual enforcement modules that is deemed to be acceptable from the network management perspective.

Another issue of paramount importance which needs to be addressed are the mechanisms that will be used to protect the overlying "security traffic only" network which allow the enforcement modules to communicate with the operator workstations. While trying to avoid a "who guards the guardian" paradox, it will be necessary to produce a methodology by which "sufficient" impenetrability for this network may be assessed.

A final area for future work is an examination of how many connections are typically supported in tandem on any given port of a switch in a wide-area ATM network. Such an assessment will be necessary in order to decide at which level to implement the integration of the security enforcement ASIC in order to provide the level of required traffic support at a minimal cost. Without such surveys, it is possible to construct devices that are prohibitively expensive yet provide support for many more connections than actually exist or, alternatively, to construct devices whose connection support is so limited as to severely handicap the capability of the network to provide the level of service for which it was designed. As a possible alternate approach to the solution of this problem, it may be possible to modify the design presented here in such a fashion that it becomes scaleable with respect to the number of connections a security module may support. This modification would allow for the addition of inexpensive, readily available option modules in those areas where connection support is found to be insufficient.

REFERENCES

[1] Uyless Black, *ATM: Foundation for Broadband Networks,* Prentice Hall PTR, Englewood Cliffs, New Jersey, 1995

[2] V. L. Voydock and S. T. Kent, "Security Mechanisms in High-Level Network Protocols," *ACM Computing Surveys,* vol. 15, no. 2, June 1983, pp. 135-171

[3] R. H. Deng, L. Gong and A. A. Lazar, "Securing Data Transfer in Asynchronous Transfer Mode Networks," *Proceedings of Global Telecommunications Conference '95,* Singapore, November 13-17, 1995, vol. 2, pp. 1198-1202

[4] J. McHugh and L. Young, "A Taxonomy of Covert Channels in ATM Networks, with Examples," Computer Science Dept., Portland State University, Portland, Oregon, Technical Report 94-3, July 1994

[5] G. E. Liepins, H. S. Vaccaro, "Detection of Anomalous Computer Session Activity," in *Proceedings of IEEE Computer Society Symposium on Security and Privacy '89,* Oakland, California, May 1-3, 1989, pp. 280-289

[6] W. J. Page, J. R. Winkler, "Intrusion and Anomaly Detection in Trusted Systems," *Proceedings of Fifth Annual Computer Security Applications Conference,* Tucson, Arizona, December 4-8, 1989, pp. 39-45

[7] M. Becker, H. Debar, D. Siboni, "A Neural Network Component for an Intrusion Detection System," in *Proceedings of IEEE Computer Society Symposium on Research in Security and Privacy '92,* Oakland, California, May 4-6, 1992, pp. 240-250

[8] L. Heberlein, K. Levitt, B. Mukherjee, "Network Intrusion Detection," *IEEE Network,* Vol. 8, Issue 3, May 1994, pp. 26-41

[9] W. Page, J. Winkler, "Intrusion and Anomaly Detection in Trusted Systems," in *Proceedings of Fifth Annual Computer Security Applications Conference,* Tucson, Arizona, December, 1989, pp. 39-45

[10] J. Brentano, G. V. Dias, T. L. Goan, T. Grance, L. T. Heberlein, C. L. Ho, K. N. Levitt, D. L. Mansur, B. Mukherjee, K. L. Pon, S. E. Smaha, S. R. Snapp, "A System for Distributed Intrusion Detection," in *Proceedings of IEEE COMPCON Spring '91,* San Francisco, California, February, 1991, pp. 170-176

[11] S. E. Smaha, "Haystack: An Intrusion Detection System," in *Proceedings of IEEE Fourth Aerospace Computer Security Applications Conference,* Orlando, Florida, December, 1988, pp. 37-44

[12] R. Jagannathan, R. Lee, S. Listgarten, T. F. Lunt, A. Whitehurst, "Knowledge-Based Intrusion Detection," in *Proceedings of the Annual AI Systems in Government Conference '89,* Washington, DC, March 27-31, 1989, pp. 102-107

[13] T. F. Lunt, "Real-Time Intrusion Detection," in *Proceedings of Thirty-Fourth IEEE Computer Society International Conference: Intellectual Leverage,* San Francisco, California, February 27 – March 3, 1989, pp. 348-353

[14] H. S. Javitz, A. Valdes, "The SRI IDES Statistical Anomaly Detector," in *Proceedings of IEEE Computer Society Symposium on Research in Security and Privacy,* Oakland, California, 20-22 May, 1991, pp. 316-326

[15] K. Tan, "The Application of Neural Networks to UNIX Computer Security," in *Proceedings of International Conference on Neural Networks '95*, Perth, Wales, Australia, 1995, pp. 476-481

[16] M. H. Rahman, J. H. Weigelt, "Securing Asynchronous Transfer Mode Based Networks Through the Use of Encryption," in *Proceedings of Global Telecommunications Conference '95*, Montreal, Quebec, pp. 1198-1202

[17] G. C. Girling, "Covert Channels in LANs," *IEEE Transactions on Software Engineering*, vol. SE-13, no. 2, February, 1987, pp. 292-296

[18] J. K. Millen, "Covert Channel Capacity," in *Proceedings of the IEEE Computer Society Symposium on Research in Security and Privacy '87*, Oakland, California, April 27-29, 1987, pp. 60-66

[19] R. Browne, "Mode Security: An Infrastructure for Covert Channel Suppression," in *Proceedings of the IEEE Computer Society Symposium on Research in Security and Privacy '94*, Oakland, California, May 16-18, 1994, pp. 39-55

[20] T. Aoki, "Future Switching System Requirements," *IEEE Communications Magazine*, January 1993, pp. 34-38

[21] R. Barker, "Broadband Networking in a National Security and Emergency Preparedness (NS/EP) Environment," in *Proceedings of Global Telecommunications Conference '93*, Boston, Massachusetts, Oct. 11-14, 1993, pp. 140-144

[22] J. R. Cleveland, N. K. Cranfill, "Emerging Technologies for the Control of the Defense Red Switch Network," in *Proceedings of Military Communications Conference '94*, Fort Monmouth, NJ, pp. 664-668

[23] T. J. Robe, K. A. Walsh, "A SONET STS-3c User Network Interface IC," in *Proceedings of IEEE Custom Integrated Circuitry Conference '91*, San Diego, California, May 1991

[24] H. J. Chao, C. A. Johnston, "The ATM Layer Chip: An ASIC for B-ISDN Applications," *IEEE Journal on Selected Areas in Communications*, Vol. 9, Issue 5, June 1991, pp. 741-750

[25] ATM Forum, "ATM User-Network Interface Specification (v3.1)," ATM Forum, September, 1994
Available through the World Wide Web at:
ftp://ftp.atmforum.com/pub/approved-specs/af-uni-0010.002.pdf.tar.Z

[26] ATM Forum, "DS1 Physical Layer Specification," ATM Forum, September, 1994
Available through the World Wide Web at:
ftp://ftp.atmforum.com/pub/approved-specs/af-phy-0016.000.ps

[27] ATM Forum, "Physical Interface Specification for 25.6 Mb/s Over Twisted Pair Cable," ATM Forum, November, 1995
Available through the World Wide Web at:
ftp://ftp.atmforum.com/pub/approved-specs/af-phy-0040.000.ps

[28] ATM Forum, "DS3 Physical Layer Specification," ATM Forum, January, 1996
Available through the World Wide Web at:
ftp://ftp.atmforum.com/pub/approved-specs/af-phy-0054.000.ps

[29] ATM Forum, "ATM Physical Medium Dependent Interface Specification for 155 Mb/s Over Twisted Pair Cable," ATM Forum, September, 1994
Available through the World Wide Web at:
ftp://ftp.atmforum.com/pub/approved-specs/af-phy-0047.000.ps

[30] ATM Forum, "622.08 Mbps Physical Layer Specification," ATM Forum, January, 1996
Available through the World Wide Web at:
ftp://ftp.atmforum.com/pub/approved-specs/af-phy-0046.000.ps

[31] Motorola Microprocessor and Memory Technologies Group, "MC68302 Integrated Multiprotocol Processor User's Manual," Motorola Corp., 1995
Available through the World Wide Web at:
http://www.mot.com/netcomm/aesop/683XX/302/302UM.pdf

[32] Motorola Microprocessor and Memory Technologies Group, "MC68PM302 Integrated Multiprotocol Processor with PCMCIA Reference Manual," Motorola Corp., 1995
Available through the World Wide Web at:
http://www.mot.com/netcomm/aesop/683XX/302/PM302UM.pdf

[33] Morris Mano, *Digital Design, 2nd edition*, Prentice-Hall Publishers, Englewood Cliffs, NJ 1991

[34] H. Ahmadi, R. Guérin, K. Sohraby, "Analysis of Leaky Bucket Access Control Mechanism with Batch Arrival Process," *Proceedings of Global Telecommunications Conference '90*, San Diego, California, Dec. 2-5, 1990, pp. 344-349

[35] J. Chao, "Design of Leaky Bucket Access Control Schemes in ATM Networks," in *Proceedings of International Conference on Communications '91*, Denver, Colorado, June 23-26, 1991, pp. 180-187

[36] Department of Defense, "Department of Defense Trusted Computer Systems Evaluation Criteria, Report DOD 5200.28-STD," Department of Defense, Washington D.C., December 1985

[37] National Computer Security Center, "A Guide to Understanding Security Modeling in Trusted Systems, Report NCSC-TG-010 Version-1," National Computer Security Center, Ft. George G. Meade, Maryland, October 1992

[38] National Computer Security Center, "Trusted Network Interpretation. Report NCSC-TG-005 Version-1," National Computer Security Center, Ft. George G. Meade, Maryland, July 1987

[39] L. S. Rutledge and L. J. Hoffman, "A Survey of Issues in Computer Network Security," *Computers and Security*, vol. 5, 1986, pp. 296-308

[40] Randy H. Katz, *Contemporary Logic Design*, The Benjamin/Cummings Publishing Company, Redwood City, CA, 1993

APPENDIX A

"PATH-ONLY"ANALYSIS MODULE SIMULATION CODE

This appendix contains the Verilog hardware description language code necessary to implement a gate-level simulation of the "path-only" version of the Analysis Module. The Receivers, Transmitters and memories involved in the design of the network security device were simulated at the procedural level and the Analysis Module was simulated at the gate level.

All sub-module inputs and outputs are fully commented.

A.A Verilog Simulation

```
module Inverter (In, Out);

  input     In;
  output    Out;

  reg       Out;

  always
    #1 Out = ~In;

endmodule


module TwoInputNANDGate (InOne, InTwo, Out);

  input     InOne, InTwo;
  output    Out;

  reg       Out;

  always
```

```
    #1 Out = ~(InOne & InTwo);

endmodule

module ThreeInputNANDGate (InOne, InTwo, InThree, Out);

  input      InOne, InTwo, InThree;
  output     Out;

  reg        Out;

  always
    #1 Out = ~(InOne & InTwo & InThree);

endmodule


module FourInputNANDGate (InOne, InTwo, InThree, InFour, Out);

  input      InOne, InTwo, InThree, InFour;
  output     Out;

  reg        Out;

  always
    #1 Out = ~(InOne & InTwo & InThree & InFour);

endmodule



module FiveInputNANDGate (InOne, InTwo, InThree, InFour, InFive, Out);

  input      InOne, InTwo, InThree, InFour, InFive;
  output     Out;
  reg        Out;

  always
    #1 Out = ~(InOne & InTwo & InThree & InFour & InFive);

endmodule




module SixInputNANDGate (InOne, InTwo, InThree, InFour, InFive, InSix, Out);

  input      InOne, InTwo, InThree, InFour, InFive, InSix;
  output     Out;

  reg        Out;

  always
    #1 Out = ~(InOne & InTwo & InThree & InFour & InFive & InSix);

endmodule



module SevenInputNANDGate (InOne, InTwo, InThree, InFour, InFive, InSix,
                           InSeven, Out                     );

  input      InOne, InTwo, InThree, InFour, InFive, InSix, InSeven;
  output     Out;

  reg        Out;

  always
    #1 Out = ~(InOne & InTwo & InThree & InFour & InFive & InSix & InSeven);

endmodule
```

```
module NineInputNANDGate (InOne, InTwo, InThree, InFour, InFive, InSix,
                    InSeven, InEight, InNine, Out             );

  input      InOne, InTwo, InThree, InFour, InFive, InSix, InSeven,
           InEight, InNine;
  output     Out;

  reg        Out;

  always
    #1 Out = ~(InOne & InTwo & InThree & InFour & InFive & InSix &
           InSeven & InEight & InNine                   );

endmodule


module TwoInputNORGate (InOne, InTwo, Out);

  input      InOne, InTwo;
  output     Out;

  reg        Out;

  always
    #1 Out = ~(InOne | InTwo);

endmodule

module ThreeInputNORGate (InOne, InTwo, InThree, Out);

  input      InOne, InTwo, InThree;
  output     Out;
  reg        Out;

  always
    #1 Out = ~(InOne | InTwo | InThree);

endmodule


module FourInputNORGate (InOne, InTwo, InThree, InFour, Out);

  input      InOne, InTwo, InThree, InFour;
  output     Out;

  reg        Out;

  always
    #1 Out = ~(InOne | InTwo | InThree | InFour);

endmodule


module FiveInputNORGate (InOne, InTwo, InThree, InFour, InFive, Out);

  input      InOne, InTwo, InThree, InFour, InFive;
  output     Out;

  reg        Out;

  always
    #1 Out = ~(InOne | InTwo | InThree | InFour | InFive);

endmodule


module SixInputNORGate (InOne, InTwo, InThree, InFour, InFive, InSix, Out);
```

```
  input       InOne, InTwo, InThree, InFour, InFive, InSix;
  output      Out;

  reg         Out;

  always
    #1 Out = ~(InOne | InTwo | InThree | InFour | InFive | InSix);

endmodule


module SevenInputNORGate (InOne, InTwo, InThree, InFour,
                          InFive, InSix, InSeven, Out   );

  input       InOne, InTwo, InThree, InFour, InFive, InSix, InSeven;
  output      Out;

  reg         Out;

  always
    #1 Out = ~(InOne | InTwo | InThree | InFour | InFive | InSix | InSeven);

endmodule


module SRLatch (Set, Reset, Out, InvertOut);

  input       Set, Reset;
  output      Out, InvertOut;

  TwoInputNANDGate   GateOne    (Set, InvertOut, Out);
  TwoInputNANDGate   GateTwo    (Reset, Out, InvertOut);

endmodule
```

```
module PosEdgeTrigLatch (Clock, Data, Out, InvertOut);

  input      Clock, Data;
  output     Out, InvertOut;

  wire       w1, w2, w3, w4;

  TwoInputNANDGate    GateOne    (w4, w2, w1);
  TwoInputNANDGate    GateTwo    (w1, Clock, w2);
  ThreeInputNANDGate  GateThree  (w2, Clock, w4, w3);
  TwoInputNANDGate    GateFour   (w3, Data, w4);
  TwoInputNANDGate    GateFive   (w2, InvertOut, Out);
  TwoInputNANDGate    GateSix    (Out, w3, InvertOut);

endmodule


// Name:    FourBitRegister
//
// Inputs:  Data    [3:0] - The data to be latched by the register on the
//                          next rising clock edge.
//          Clock         - The clocking signal which controls data latching.
//
// Outputs: Out     [3:0] - The data latched on the last rising clock edge.

module FourBitRegister (Clock, Data, Out);

  input  [3:0]  Data;
  input         Clock;
  output [3:0]  Out;

  wire   [3:0]  OutInv;

  PosEdgeTrigLatch   BitZero   (Clock, Data[0], Out[0], OutInv[0]);
  PosEdgeTrigLatch   BitOne    (Clock, Data[1], Out[1], OutInv[1]);
  PosEdgeTrigLatch   BitTwo    (Clock, Data[2], Out[2], OutInv[2]);
  PosEdgeTrigLatch    BitThree   (Clock, Data[3], Out[3], OutInv[3]);

endmodule


// Name:    EightBitRegister
//
// Inputs:  Data     [7:0] - The data to be latched by the register on the
//                          next rising clock edge.
//          Clock          - The clocking signal which controls data latching.
//
// Outputs: Out      [7:0] - The data latched on the last rising clock edge.

module EightBitRegister (Clock, Data, Out);

  input  [7:0]  Data;
  input         Clock;
  output [7:0]  Out;

  FourBitRegister    LowNibble  (Clock, Data[3:0], Out[3:0]);
  FourBitRegister    HighNibble (Clock, Data[7:4], Out[7:4]);

endmodule


// Name:    SixteenBitRegister
//
// Inputs:  Data     [15:0] - The data to be latched by the register on the
//                           next rising clock edge.
//          Clock           - The clocking signal which controls data latching.
//
// Outputs: Out      [15:0] - The data latched on the last rising clock edge.

module SixteenBitRegister (Clock, Data, Out);

  input  [15:0] Data;
  input        Clock;
  output [15:0] Out;

  EightBitRegister  LowByte  (Clock, Data[7:0],  Out[7:0] );
  EightBitRegister  HighByte (Clock, Data[15:8], Out[15:8]);

endmodule


// Name:    ShiftRegister
//
// Inputs:  Data    [15:0] - The data to be latched by the shift register on
//                   the next rising clock edge.
//          Clock        - The clocking signal which controls data latching.
//
// Outputs: Out     [15:0] - The data latched on the rising clock edge twenty
//                   seven clock cycles ago
//          OutInv  [15:0] - The negation of the data latched on the rising
//                   clock edge nine clock cycles ago.

module ShiftRegister (Clock, Data, Out);

  input  [15:0] Data;
  input        Clock;
  output [15:0] Out;

  wire   [15:0]  L1, L2, L3, L4, L5, L6, L7, L8;

  SixteenBitRegister Stage0  (Clock, Data, L1);
  SixteenBitRegister Stage1  (Clock, L1,   L2);
  SixteenBitRegister Stage2  (Clock, L2,   L3);
  SixteenBitRegister Stage3  (Clock, L3,   L4);
  SixteenBitRegister Stage4  (Clock, L4,   L5);
  SixteenBitRegister Stage5  (Clock, L5,   L6);
  SixteenBitRegister Stage6  (Clock, L6,   L7);
  SixteenBitRegister Stage7  (Clock, L7,   L8);
  SixteenBitRegister Stage8  (Clock, L8,   Out);

endmodule



// Name:    DataGate
//
// Inputs:  In             - Data input.
//          Select         - If asserted low, the bit value at "In" will be
//                           reflected at "Out". Otherwise, "Out" will
//                           reflect zero.
//
// Outputs: Out            - Reflect "In" if Select is low, otherwise low
//                           regardless of the state of "In".

module DataGate (In, Select, Out);

  input      In, Select;
  output     Out;

  wire       OutInvert;

  TwoInputNANDGate   Gate     (In, Select, OutInvert);
  Inverter           Invert   (OutInvert, Out);

endmodule



// Name:    FourBitDataGate
//
// Inputs:  In       [3:0] - Data input.
//          Select         - If asserted high, the bit values at "In" will be
//                           reflected at "Out". Otherwise, "Out" will
//                           reflect all zeroes.
//
// Outputs: Out      [3:0] - Reflect "In" if Select is high, otherwise just
//                  set all bits to low.

module FourBitDataGate (In, Select, Out);

  input  [3:0]  In;
  input         Select;
  output [3:0]  Out;

  DataGate  Bit0 (In[0], Select, Out[0]);
  DataGate  Bit1 (In[1], Select, Out[1]);
  DataGate  Bit2 (In[2], Select, Out[2]);
  DataGate  Bit3 (In[3], Select, Out[3]);

endmodule



// Name:    EightBitDataGate
//
// Inputs:  In[7:0]       - Data input.
//          Select        - If asserted high, the bit values at "In" will be
//                          reflected at "Out". Otherwise, "Out" will
//                          reflect all zeroes.
//
// Outputs: Out[7:0]      - Reflect "In" if Select is high, otherwise just
//                          go low on all bits.

module EightBitDataGate (In, Select, Out);

  input  [7:0] In;
  input        Select;
  output [7:0] Out;

  FourBitDataGate  LowNibble  (In[3:0], Select, Out[3:0]);
  FourBitDataGate  HighNibble (In[7:4], Select, Out[7:4]);

endmodule
```

```
// Name:    TwoLineSelector
//
// Inputs:  In[1:0]       - Two bits of data input
//          Select        - Input that must be asserted in order to control
//                          which of the two bits of input will be reflected
//                          at the output.
//          SelectInv     - Input which is the inverse of "Select"
//
// Outputs: Out           - Reflect the value at "In[1]" if "Select" is
//                          high and "SelectInv" is low. Reflect the value
//                          at "In[0]" if "Select" is low and "SelectInv" is
//                          high. Behavior is unpredictable otherwise.

module TwoLineSelector (In, Select, SelectInv, Out);

  input  [1:0]  In;
  input         Select, SelectInv;
  output        Out;

  wire   [1:0]  Con;

  TwoInputNANDGate   GateZero  (In[0],  SelectInv, Con[0]);
  TwoInputNANDGate   GateOne   (In[1],  Select,    Con[1]);
  TwoInputNANDGate   GateTwo   (Con[0], Con[1],    Out);

endmodule


// Name:    FourBitTwoLineSelector
//
// Inputs:  InZero  [3:0] - The first input line
//          InOne   [3:0] - The second input line
//          Select        - Input that must be asserted in order to control
//                          which of the two nibbles of input will be
//                  reflected at the output nibble.
//
// Outputs: Out     [3:0] - Reflect the nibble at "InOne" if "Select" is
//                  high. Otherwise, reflect the nibble at "InTwo".

module FourBitTwoLineSelector (InZero, InOne, Select, Out);

  input  [3:0]  InZero, InOne;
  input         Select;
  output [3:0]  Out;

  wire          SelectInv;
  wire   [7:0]  Input;

  assign     Input[0]  = InZero[0], Input[1]  = InOne[0],
             Input[2]  = InZero[1], Input[3]  = InOne[1],
             Input[4]  = InZero[2], Input[5]  = InOne[2],
             Input[6]  = InZero[3], Input[7]  = InOne[3];


  Inverter      Invert  (Select, SelectInv);
  TwoLineSelector Select0 (Input[1:0], Select, SelectInv, Out[0]);
  TwoLineSelector Select1 (Input[3:2], Select, SelectInv, Out[1]);
  TwoLineSelector Select2 (Input[5:4], Select, SelectInv, Out[2]);
  TwoLineSelector Select3 (Input[7:6], Select, SelectInv, Out[3]);

endmodule



// Name:   FourLineSelector
//
// Inputs:  In[3:0]       - Four bits of data input
//          Select[1:0]   - Inputs that must be asserted in order to control
//                          which of the four bits of input will be
//                          reflected at the output.
//        SelectInv[1:0]  - Input which is the inverse of "Select[1:0]" on
//                          all bits.
//
// Outputs: Out           - Depending on the state of the "Select" inputs,
//                          this signal will reflect the state of one of the
//                          bits at the "In" input, according to the table
//                          below. Behavior is unpredictable for conditions
//                          not covered in the table.
//
//                          Sel[0]  SelInv[0]  Sel[1]  SelInv[1]  |  Out
//                          --------------------------------------+-------
//                           Low     High       Low     High      | In[0]
//                           Low     High       High    Low       | In[1]
//                           High    Low        Low     High      | In[2]
//                           High    Low        High    Low       | In[3]

module FourLineSelector (In, Select, SelectInv, Out);

  input  [3:0]  In;
  input  [1:0]  Select, SelectInv;
  output        Out;

  wire   [3:0]  Con;

  ThreeInputNANDGate  GateZero  (In[0], SelectInv[0], SelectInv[1], Con[0]);
  ThreeInputNANDGate  GateOne   (In[1], SelectInv[0], Select[1]   , Con[1]);
  ThreeInputNANDGate  GateTwo   (In[2], Select[0],    SelectInv[1], Con[2]);
  ThreeInputNANDGate  GateThree (In[3], Select[0],    Select[1],    Con[3]);
  FourInputNANDGate   GateFour  (Con[0], Con[1], Con[2], Con[3], Out);

endmodule


// Name:   TwelveBitFourLineSelector
//
// Inputs:  InZero  [11:0] - The first input line
//      InOne   [11:0] - The second input line
//      InTwo   [11:0] - The third input line
//      InThree [11:0] - The fourth input line
//      Select  [1:0]  - Controls whose state govern which of the four
//                       inputs will be reflected at the output.
//                       reflected at the output.
//
// Outputs: Out   [11:0] - Depending on the state of the "Select" inputs,
//                       this signal will reflect the state of the twelve
//                       bits at one of the four inputs. Behavior is
//                       unpredictable for conditions not covered in the
//                       table.
//                            Select0  Select1  | Out
//                            ------------------+----------
//                             Low      Low     | InZero
//                             Low      High    | InOne
//                             High     Low     | InTwo
//                             High     High    | InThree

module TwelveBitFourLineSelector (InZero, InOne, InTwo, InThree, Select, Out);

  input  [11:0] InZero, InOne, InTwo, InThree;
  input  [1:0]  Select;
  output [11:0] Out;

  wire   [1:0]  SelectInv;
  wire   [47:0] Input;

  assign      Input[0]  = InZero[0],   Input[1]  = InOne[0],
            Input[2]  = InTwo[0],    Input[3]  = InThree[0],
            Input[4]  = InZero[1],   Input[5]  = InOne[1],
            Input[6]  = InTwo[1],    Input[7]  = InThree[1],
            Input[8]  = InZero[2],   Input[9]  = InOne[2],
            Input[10] = InTwo[2],    Input[11] = InThree[2],
            Input[12] = InZero[3],   Input[13] = InOne[3],
            Input[14] = InTwo[3],    Input[15] = InThree[3],
        Input[16] = InZero[4],  Input[17] = InOne[4],
        Input[18] = InTwo[4],   Input[19] = InThree[4],
        Input[20] = InZero[5],  Input[21] = InOne[5],
        Input[22] = InTwo[5],   Input[23] = InThree[5],
        Input[24] = InZero[6],  Input[25] = InOne[6],
        Input[26] = InTwo[6],   Input[27] = InThree[6],
        Input[28] = InZero[7],  Input[29] = InOne[7],
        Input[30] = InTwo[7],   Input[31] = InThree[7],
        Input[32] = InZero[8],  Input[33] = InOne[8],
        Input[34] = InTwo[8],   Input[35] = InThree[8],
        Input[36] = InZero[9],  Input[37] = InOne[9],
        Input[38] = InTwo[9],   Input[39] = InThree[9],
        Input[40] = InZero[10], Input[41] = InOne[10],
        Input[42] = InTwo[10],  Input[43] = InThree[10],
        Input[44] = InZero[11], Input[45] = InOne[11],
        Input[46] = InTwo[11],  Input[47] = InThree[11];


  Inverter      Invert0 (Select[0], SelectInv[0]);
  Inverter      Invert1 (Select[1], SelectInv[1]);
  FourLineSelector Select0  (Input[3:0],   Select, SelectInv, Out[0] );
  FourLineSelector Select1  (Input[7:4],   Select, SelectInv, Out[1] );
  FourLineSelector Select2  (Input[11:8],  Select, SelectInv, Out[2] );
  FourLineSelector Select3  (Input[15:12], Select, SelectInv, Out[3] );
  FourLineSelector Select4  (Input[19:16], Select, SelectInv, Out[4] );
  FourLineSelector Select5  (Input[23:20], Select, SelectInv, Out[5] );
  FourLineSelector Select6  (Input[27:24], Select, SelectInv, Out[6] );
  FourLineSelector Select7  (Input[31:28], Select, SelectInv, Out[7] );
  FourLineSelector Select8  (Input[35:32], Select, SelectInv, Out[8] );
  FourLineSelector Select9  (Input[39:36], Select, SelectInv, Out[9] );
  FourLineSelector Select10 (Input[43:40], Select, SelectInv, Out[10]);
  FourLineSelector Select11 (Input[47:44], Select, SelectInv, Out[11]);

endmodule
```

```
// Name:    ClockGen
//
// Inputs:  None.
//
// Outputs: Clock        - Square wave that cycles up and down every
//                       13 nanoseconds thereby producing a signal with a
//                       period of 26 nanoseconds.

module ClockGen (Clock);

  output     Clock;

  reg        Clock;

  initial
    Clock = 1;

  always
    begin
      #13 Clock = 0;
      #13 Clock = 1;
    end

endmodule


// Name:    NewPathStore    - Simulates the storage elements that accept and
//                        hold data about a new path to be loaded into
//                        the memory lookup module by the sequence /
//                        detect module at the appropriate time
//
// Inputs:  Load         - The Set input on the SR latch indicating
//                        whether the unit still contains new data
//          UnLoad       - The Reset input on the SR latch indicating
//                        whether the unit still contains new data
//      DataIn          - Input indicating whether the new path is to be
//                      validated or invalidated
//      AddressIn [23:0]- The input for the new path which is to be
//                      validated or invalidated
//
// Outputs: Full        - The Q output on the SR latch which, if high,
//                      indicates the unit contains new data.
//      Empty           - The Q' output on the SR latch which, if high,
//                      indicates the unit does not contain new data.
//      DataOut         - Output indicating whether the new path
//                      currently stored is to be validated or
//                      invalidated
//      AddressOut[23:0]- The output of the new path which is to be
//                      validated or invalidated

module NewPathStore (Load, UnLoad, AddressIn,  DataIn,
            Full, Empty,  AddressOut, DataOut);

  input       Load, UnLoad, DataIn;
  input  [23:0] AddressIn;
  output      Full, Empty, DataOut;
  output [23:0] AddressOut;

  wire        DataOutInv;

  EightBitRegister Low    (Load, AddressIn[7:0],   AddressOut[7:0]  );
  EightBitRegister Middle (Load, AddressIn[15:8],  AddressOut[15:8] );
  EightBitRegister High   (Load, AddressIn[23:16], AddressOut[23:16]);
  PosEdgeTrigLatch Data   (Load, DataIn, DataOut, DataOutInv);
  SRLatch        Status (Load, UnLoad, Full, Empty);

endmodule


// Name:   DynamicRAM      - Simulates a Texas Instruments SMJ416100-70
//                      dynamic random access memory
```

```
//
// Inputs:  Address [11:0]  - DRAM address lines
//          RAS             - Row address select
//          CAS             - Column address select
//          W              - Read/Write select
//          D             - Data input on memory writes
//
// Outputs: Q               - Data output on memory reads

module DynamicRAM (Address, RAS, CAS, W, D, Q);

  input  [11:0] Address;
  input         RAS, CAS, W, D;
  output        Q;

  reg   [11:0] Row, Column;
  reg          Q, DataIn;

  initial
    Q = 1'bz;

  always
    begin
      wait (!RAS)
      Row = Address;
      wait (!CAS)
      Column = Address;
      if (W == 0)
        begin
          // we are performing a write cycle
          DataIn = D;
          wait (CAS)
          Q = 1'bz;
        end
      else
        begin
```

```
            // we are performing a read cycle
            // for this simulation just present the low bit of the address
            #18 Q = Address[0];
            wait (CAS)
            Q = 1'bz;
          end
      end

endmodule



// Name:    NetworkReceiver
//
// Inputs:  Clock               - Clock on whose negative edge to present data
//
// Outputs: Out         [15:0]  - Present data produced by the receiver.
//          NewCellEven         - Asserted when the starting byte of the cell
//                                currently being transmitted was presented on
//                                the high-order byte of the output.
//          NewCellOdd          - Asserted when the starting byte of the cell
//                                currently being transmitted was presented on
//                                the low-order byte of the output.

module NetworkReceiver (Clock, NewCellLow, NewCellHigh, Out);

  input        Clock;
  output [15:0] Out;
  output       NewCellLow, NewCellHigh;

  reg   [15:0] Out, Temp;
  reg          NewCellLow, NewCellHigh;

  initial
    begin
      @ (negedge Clock) Out[15:8]  = 8'b00000000;
                        Out[7:0]   = 8'b00000001;
```

```
                NewCellLow  = 0;
                NewCellHigh = 0;
   end

  always
   begin
     @ (negedge Clock) Temp[15:8] = Out[15:8] + 2;
                Temp[7:0]  = Out[7:0] + 2;

                if (Temp[15:8] > 52)
                  begin
                    Temp[15:8] = Temp[15:8] - 53;
                    if (Temp[15:8] == 0) NewCellLow = 1;
                  end
                else
                  NewCellLow = 0;

                if (Temp[7:0]  > 52)
                  begin
                    Temp[7:0]  = Temp[7:0]  - 53;
                    if (Temp[7:0] == 0) NewCellHigh = 1;
                  end
                else
                  NewCellHigh = 0;

                Out[15:0] = Temp[15:0];
   end

endmodule


// Name:    NetworkTransmitter
//
// Inputs:  Data       [15:0] - The data to be transmitted out onto the
//                              network.
//          NewCellEven       - Asserted when the starting byte of the cell
//                      currently being transmitted was presented on
//                      the high-order byte of the input.
//      NewCellOdd      - Asserted when the starting byte of the cell
//                      currently being transmitted was presented on
//                      the low-order byte of the input.

module NetworkTransmitter (Clock, NewCellEven,  NewCellOdd,  Data);

  input      Clock, NewCellEven, NewCellOdd;
  input  [15:0] Data;

endmodule


module ResetControl (Clock, Input, Output);
  input      Clock, Input;
  output     Output;

  reg        Output;

  initial
    begin
      Output = 0;
      #26 Output = Input;
      // @(negedge Clock) Output = Input;
    end

  always
    Output = Input;

endmodule


// Name:   ControlModule
//
// Inputs:  LatchSet        - If high, indicates that the new path storage
//                      module still contains new data.
//      LatchReset      - If high, indicates that the new path storage
//                      module has been cleared of new data.
//
// Output:  SetLatch    - If high, indicates that new data has been
//                      presented and should be latched.
//      Data            - If high, indicates that the new path
//                      being modified is to be a valid path.
//                      Otherwise, the new path is to be an invalid
//                      one.
//      Address         - Indicates the VPI/VCI pair of the path whose
//                      status is to be modified.

module ControlModule (Address, Data, SetLatch, LatchSet, LatchReset);

  input      LatchSet, LatchReset;
  output     SetLatch, Data;
  output [23:0] Address;

  reg        SetLatch, Data;
  reg   [23:0] Address;

  initial
    begin
      SetLatch = 0; Data = 0; Address = 0;
    end

  always
    begin
      #1 if (LatchSet == 0)
          begin
            Address = Address + 1;
            if (Data == 0) Data = 1;
            if (Data == 1) Data = 0;
            #1 SetLatch = 1;
            #1 SetLatch = 0;
```

```
         end
     end

endmodule


// Name:    DownCounterWithPreset
//
// Inputs:  Clock          - Signal on whose positive edge, the counter
//                         must change state
//       Set26             - If high on a rising edge of "Clock", then
//                         it forces the next state of the counter to
//                         be 26 transitions away from zero.
//       Set27             - If high on a rising edge of "Clock", then
//                         it forces the next state of the counter to
//                         be 27 transitions away from zero.
//
// Output:  Bit0...4        - Individual lines of the output of the five
//                         latches that store the current state of the
//                         counter. Bit0 refers to the lowest order
//                         bit and Bit4 to the highest order bit.

module DownCounterWithPreset (Clock, Set26, Set27,
                    Bit0, Bit1, Bit2, Bit3, Bit4);

  input      Clock, Set26, Set27;
  output     Bit0, Bit1, Bit2, Bit3, Bit4;

  wire       Bit0Input, Bit1Input, Bit2Input, Bit3Input, Bit4Input;
  wire       Set26Inv, Set27Inv;
  wire  [22:0] Line;

  // Memory elements to store the current state
  PosEdgeTrigLatch   BitZero  (Clock, Bit0Input, Bit0, Bit0Inv);
  PosEdgeTrigLatch   BitOne   (Clock, Bit1Input, Bit1, Bit1Inv);
  PosEdgeTrigLatch   BitTwo   (Clock, Bit2Input, Bit2, Bit2Inv);
PosEdgeTrigLatch   BitThree (Clock, Bit3Input, Bit3, Bit3Inv);
PosEdgeTrigLatch   BitFour  (Clock, Bit4Input, Bit4, Bit4Inv);

// Prepare inputs
Inverter        Gate0   (Set26, Set26Inv);
Inverter        Gate1   (Set27, Set27Inv);

// Decode logic for bit 0
TwoInputNORGate    Gate2   (Bit0Inv, Set27, Line[2]);
TwoInputNORGate    Gate3   (Set26Inv, Set27, Line[3]);
SixInputNORGate    Gate4   (Bit0, Bit1, Bit2, Bit3, Bit4, Set27,
                    Line[4]                     );
ThreeInputNORGate  Gate5   (Line[2], Line[3], Line[4], Bit0Input);

// Decode logic for bit 1
FourInputNORGate   Gate6   (Bit0, Bit1Inv, Set26, Set27, Line[6]);
FourInputNORGate   Gate7   (Bit0Inv, Bit1, Set26, Set27, Line[7]);
SevenInputNORGate  Gate8   (Bit0, Bit1, Bit2, Bit3, Bit4, Set26,
                    Set27, Line[8]              );
ThreeInputNORGate  Gate9   (Line[6], Line[7], Line[8], Bit1Input);

// Decode logic for bit 2
TwoInputNORGate    Gate10  (Bit1Inv, Bit2, Line[10]);
TwoInputNORGate    Gate11  (Bit0Inv, Bit2, Line[11]);
ThreeInputNORGate  Gate12  (Bit0, Bit1, Bit2Inv, Line[12]);
FiveInputNORGate   Gate13  (Bit0, Bit1, Bit2, Bit3, Bit4, Line[13]);
SixInputNORGate    Gate14  (Line[10], Line[11], Line[12], Line[13],
                    Set26, Set27, Bit2Input     );

// Decode logic for bit 3
FourInputNORGate   Gate15  (Bit2Inv, Bit3, Set26, Set27, Line[15]);
FourInputNORGate   Gate16  (Bit1Inv, Bit3, Set26, Set27, Line[16]);
FourInputNORGate   Gate17  (Bit0Inv, Bit3, Set26, Set27, Line[17]);
SixInputNORGate    Gate18  (Bit0, Bit1, Bit2, Bit3Inv, Set26, Set27,
                    Line[18]                    );
SevenInputNORGate  Gate19  (Bit0, Bit1, Bit2, Bit3, Bit4, Set26, Set27,
                    Line[19]                         );
  FiveInputNORGate    Gate20   (Line[15], Line[16], Line[17], Line[18],
                    Line[19], Bit3Input              );

  // Decode logic for bit 4
  ThreeInputNORGate   Gate21   (Bit4, Set26, Set27, Line[21]);
  SixInputNORGate     Gate22   (Bit0, Bit1, Bit2, Bit3, Set26, Set27,
                    Line[22]                         );
  TwoInputNORGate     Gate23   (Line[21], Line[22], Bit4Input);

endmodule


// Name:    StateControl
//
// Inputs:  NewCellLow        - When high, indicates a new cell is coming
//                              in with the first byte starting on the low
//                              order bits of the input.
//          NewCellHigh       - When high, indicates a new cell is coming
//                              in with the first byte starting on the high
//                              order bits of the input.
//          LookupRes         - Path validity result of the memory lookup
//                              for the transitting cell
//          Bit         [4:0] - The state of the five bits which define the
//                              current state of the state machine for which
//                              the control lines must be decoded.
//          BitInv      [4:0] - The negated state of the five bits specified
//                              by the "Bit" input.
//
// Output:  PVRL              - Latch the results of the read from the
//                              memory lookup module.
//          RSRL              - Clear the new path information in the new
//                              path registers (by setting the SR-Latch
//                              indicating the validity of the data as
//                              being false)
//          LLODG26           - Start the low-byte counter at 26
//      LLODG27          - Start the low-byte counter at 27
//      LHODG27          - Start the high-byte counter at 27
//      VVRL           - Not applicable to "path-only" Analysis Mod.
//      RAS           - Row address select line on the memory
//                    lookup module
//      CAS           - Column address select line on the memory
//                    lookup module
//      W             - Read/Write control line on the memory
//                    lookup module
//      FourBDS     [5:0] - Control lines to the four bit multiplexer
//                    that shunt different portions of the
//                    incoming data words from the Receiver
//      FourBDL     [5:0] - Latch control lines on the latches that store
//                    the path information of the currently
//                    transiting cell
//      TwelveBDS   [1:0] - Control lines to the twelve bit by four line
//                    multiplexer that presents data from
//                    various latch groups to the memory lookup
//                    module

module StateControl (Bit, BitInv, NewCellLow, NewCellHigh, LookupRes,
            FourBDS, FourBDL, TwelveBDS, PVRL, RSRL,
            LLODG26, LLODG27, LHODG27, VVRL, RAS, CAS, W);

  input      NewCellLow, NewCellHigh, LookupRes;
  input  [4:0] Bit, BitInv;
  output     PVRL, RSRL, LLODG26, LLODG27, LHODG27, VVRL, RAS, CAS, W;
  output [5:0] FourBDS, FourBDL;
  output [1:0] TwelveBDS;

  wire       LowStart, HighStart;
  wire   [4:0] Stage;
  wire  [28:0] Line;

  assign     FourBDL[0]  = Stage[1],  FourBDL[1]  = Stage[2],
          FourBDL[2]  = Stage[2],  FourBDL[3]  = Stage[3],
        FourBDL[4]  = Stage[3],  FourBDL[5]  = Stage[4],
        TwelveBDS[0] = BitInv[2], TwelveBDS[1] = Bit[4],
        LLODG26    = LowStart,  LHODG27    = LowStart,
        LLODG27    = HighStart;
// Logic for PVRL
FiveInputNANDGate  Gate0  (BitInv[4], BitInv[3], Bit[2], BitInv[1],
                   Bit[0], Line[0]                 );
Inverter        Gate1   (Line[0], PVRL);

// Logic for RSRL
FiveInputNANDGate  Gate2  (Bit[4], Bit[3], Bit[2], BitInv[1], Bit[0],
                   Line[2]                        );
Inverter        Gate3   (Line[2], RSRL);

// Logic for LxODG2x
SevenInputNANDGate  Gate4   (BitInv[4], Bit[3], Bit[2], BitInv[1],
                   BitInv[0], NewCellLow, LookupRes, Line[4]);
SevenInputNANDGate  Gate5   (BitInv[4], Bit[3], Bit[2], BitInv[1],
                   BitInv[0], NewCellHigh, LookupRes, Line[5]);
Inverter        Gate6   (Line[4], LowStart);
Inverter        Gate7   (Line[5], HighStart);

// Logic for VVRL
FiveInputNANDGate  Gate8   (BitInv[4], BitInv[3], Bit[2], BitInv[1],
                   BitInv[0], Line[8]              );
Inverter        Gate9   (Line[8], VVRL);

// Logic for RAS
TwoInputNANDGate   Gate10  (Bit[4], BitInv[3], Line[10]);
TwoInputNANDGate   Gate11  (BitInv[2], Bit[0], Line[11]);
ThreeInputNANDGate Gate12  (Bit[4], Bit[2], BitInv[1], Line[12]);
ThreeInputNANDGate Gate13  (Bit[3], BitInv[1], Bit[0], Line[13]);
ThreeInputNANDGate Gate14  (BitInv[4], BitInv[1], BitInv[0], Line[14]);
FiveInputNANDGate  Gate15  (Line[10], Line[11], Line[12], Line[13],
                   Line[14], RAS                  );
```

```
    // Logic for CAS
    TwoInputNANDGate    Gate16  (BitInv[3], BitInv[2], Line[16]);
    ThreeInputNANDGate  Gate17  (BitInv[3], Bit[1], BitInv[0], Line[17]);
    FourInputNANDGate   Gate18  (Bit[4], Bit[3], Bit[1], BitInv[0], Line[18]);
    FourInputNANDGate   Gate19  (Bit[4], Bit[3], BitInv[1], Bit[0], Line[19]);
    FourInputNANDGate   Gate20  (Bit[4], Bit[3], BitInv[2], Bit[1], Line[20]);
    FiveInputNANDGate   Gate21  (Line[16], Line[17], Line[18], Line[19],
                      Line[20], CAS                  );

    // Logic for W
    FiveInputNANDGate   Gate22  (Bit[4], Bit[3], Bit[2], Bit[1], Bit[0], W);

    // Logic for 4BDL
    FiveInputNANDGate   Gate23  (BitInv[4], BitInv[3], BitInv[2],
                      BitInv[1], Bit[0], Line[23]   );
    FiveInputNANDGate   Gate24  (Bit[4], BitInv[3], Bit[2], BitInv[1],
                      BitInv[0], Line[24]          );
    FiveInputNANDGate   Gate25  (Bit[4], BitInv[3], Bit[2], BitInv[1],
                      Bit[0], Line[25]             );
    FiveInputNANDGate   Gate26  (BitInv[4], BitInv[3], BitInv[2], Bit[1],
                      Bit[0], Line[26]             );
    FiveInputNANDGate   Gate27  (Bit[4], BitInv[3], Bit[2], Bit[1], Bit[0],
                      Line[27]                     );
    TwoInputNANDGate    Gate28  (Line[23], Line[24], Stage[1]);
    TwoInputNANDGate    Gate29  (Line[23], Line[25], Stage[2]);
    TwoInputNANDGate    Gate30  (Line[26], Line[25], Stage[3]);
    TwoInputNANDGate    Gate31  (Line[26], Line[27], Stage[4]);

    // No logic block necessary for 12BDS[B,S]

endmodule


// Name:   StateMachine
//
// Inputs:  Clock          - Signal on whose rising edge the state
//                      machine must make a state change
//      NewCellLow      - When high, indicates a new cell is coming
//                      in with the first byte starting on the low
//                      order bits of the input.
//      NewCellHigh     - When high, indicates a new cell is coming
//                      in with the first byte starting on the high
//                      order bits of the input.
//      LatchSet        - Output of the SR-Latch which, if high,
//                      indicates there is new path data to be loaded
//                      into the memory lookup module.
//      LatchReset      - The negated state of the LatchSet input.
//
// Output: Bit      [4:0] - The state of the five bits which define the
//                      current state of the state machine.
//      BitInv    [4:0] - The negated state of the five bits specified
//                      by the "Bit" input.

module StateMachine (Clock, NewCellLow, NewCellHigh, LatchSet, LatchReset,
            Bit, BitInv                         );

  input     Clock, NewCellLow, NewCellHigh, LatchSet, LatchReset;
  output  [4:0] Bit, BitInv;

  wire   [4:0] BitDecode, BitInput;
  wire   [39:0] Line;


  // Memory elements to store the current state
  PosEdgeTrigLatch   BitZero  (Clock, BitInput[0], Bit[0], BitInv[0]);
  PosEdgeTrigLatch   BitOne   (Clock, BitInput[1], Bit[1], BitInv[1]);
  PosEdgeTrigLatch   BitTwo   (Clock, BitInput[2], Bit[2], BitInv[2]);
  PosEdgeTrigLatch   BitThree (Clock, BitInput[3], Bit[3], BitInv[3]);
  PosEdgeTrigLatch   BitFour  (Clock, BitInput[4], Bit[4], BitInv[4]);

  // Decode logic for bit 0
  ThreeInputNANDGate  Gate0   (Bit[3], Bit[2], BitInv[1], Line[0]);
ThreeInputNANDGate  Gate1   (Bit[4], Bit[3], Bit[2], Line[1]);
ThreeInputNANDGate  Gate2   (Bit[4], BitInv[1], Bit[0], Line[2]);
ThreeInputNANDGate  Gate3   (Bit[4], Bit[3], BitInv[0], Line[3]);
FourInputNANDGate   Gate4   (BitInv[4], Bit[3], BitInv[2], Bit[1],
                    Line[4]                    );
FourInputNANDGate   Gate5   (BitInv[4], BitInv[3], Bit[2], Bit[1],
                    Line[5]                    );
FourInputNANDGate   Gate6   (BitInv[3], BitInv[2], BitInv[1], Bit[0],
                    Line[6]                      );
FiveInputNANDGate   Gate7   (BitInv[3], BitInv[2], BitInv[1],
                    BitInv[0], NewCellLow, Line[7]);
FiveInputNANDGate   Gate8   (Bit[4], BitInv[3], BitInv[1], BitInv[0],
                    NewCellHigh, Line[8]          );
NineInputNANDGate   Gate9   (Line[0], Line[1], Line[2], Line[3],
                    Line[4], Line[9], Line[6], Line[7],
                    Line[8], BitDecode[0]         );
ResetControl     Reset0  (Clock, BitDecode[0], BitInput[0]);

// Decode logic for bit 1
TwoInputNANDGate    Gate10  (Bit[1], BitInv[0], Line[10]);
ThreeInputNANDGate  Gate11  (BitInv[3], BitInv[2], Bit[0], Line[11]);
ThreeInputNANDGate  Gate12  (Bit[4], BitInv[3], Bit[0], Line[12]);
FourInputNANDGate   Gate13  (BitInv[4], Bit[3], Bit[2], Bit[0], Line[13]);
FourInputNANDGate   Gate14  (Bit[4], BitInv[2], BitInv[1], Bit[0],
                    Line[14]                   );
FiveInputNANDGate   Gate15  (Bit[4], BitInv[2], Bit[1], Bit[0],
                    LatchSet, Line[15]           );
SixInputNANDGate    Gate16  (Line[10], Line[11], Line[12], Line[13],
                    Line[14], Line[15], BitDecode[1]     );
ResetControl     Reset1  (Clock, BitDecode[1], BitInput[1]);

// Decode logic for bit 2
ThreeInputNANDGate  Gate17  (BitInv[4], BitInv[3], Bit[2], Line[17]);
ThreeInputNANDGate  Gate18  (BitInv[4], Bit[2], BitInv[1], Line[18]);
ThreeInputNANDGate  Gate19  (BitInv[4], Bit[2], Bit[0], Line[19]);
ThreeInputNANDGate  Gate20  (BitInv[3], Bit[1], BitInv[0], Line[20]);
ThreeInputNANDGate  Gate21  (Bit[4], Bit[1], BitInv[0], Line[21]);
ThreeInputNANDGate  Gate22  (Bit[4], Bit[3], Bit[2], Line[22]);
ThreeInputNANDGate  Gate23  (Bit[2], BitInv[1], Bit[0], Line[23]);
FiveInputNANDGate   Gate24  (Bit[4], BitInv[3], BitInv[1], BitInv[0],
                    NewCellHigh, Line[24]              );
FiveInputNANDGate   Gate25  (Bit[4], BitInv[2], Bit[1], Bit[0],
                    LatchReset, Line[25]          );
NineInputNANDGate   Gate26  (Line[17], Line[18], Line[19], Line[20],
                    Line[21], Line[22], Line[23], Line[24],
                    Line[25], BitDecode[2]             );
ResetControl        Reset2  (Clock, BitDecode[2], BitInput[2]);

// Decode logic for bit 3
TwoInputNANDGate    Gate27  (BitInv[4], Bit[3], Line[27]);
ThreeInputNANDGate  Gate28  (Bit[3], Bit[2], Bit[1], Line[28]);
ThreeInputNANDGate  Gate29  (Bit[3], Bit[1], BitInv[0], Line[29]);
ThreeInputNANDGate  Gate30  (Bit[3], BitInv[1], Bit[0], Line[30]);
ThreeInputNANDGate  Gate31  (Bit[3], BitInv[2], BitInv[1], Line[31]);
FourInputNANDGate   Gate32  (BitInv[4], Bit[2], BitInv[1], BitInv[0],
                    Line[32]                      );
FiveInputNANDGate   Gate33  (Bit[4], BitInv[2], Bit[1], Bit[0], LatchSet,
                    Line[33]                          );
SevenInputNANDGate  Gate34  (Line[27], Line[28], Line[29], Line[30],
                    Line[31], Line[32], Line[33], BitDecode[3]);
ResetControl        Reset3  (Clock, BitDecode[3], BitInput[3]);

// Decode logic for bit 4
TwoInputNANDGate    Gate35  (Bit[4], Bit[3], Line[35]);
ThreeInputNANDGate  Gate36  (Bit[4], BitInv[1], Bit[0], Line[36]);
FourInputNANDGate   Gate37  (Bit[3], BitInv[2], BitInv[1], BitInv[0],
                    Line[37]                      );
FourInputNANDGate   Gate38  (Bit[4], BitInv[3], BitInv[0], NewCellHigh,
                    Line[38]                        );
FourInputNANDGate   Gate39  (Line[35], Line[36], Line[37], Line[38],
                    BitDecode[4]                    );
ResetControl        Reset4  (Clock, BitDecode[4], BitInput[4]);
```

```
endmodule


// Name:    SequenceDetect
//
// Inputs:  Clock          - Signal on whose rising edge the state
//                         machine must make a state change.
//          NewCellLow     - When high, indicates a new cell is coming
//                         in with the first byte starting on the low
//                         order bits of the input.
//          NewCellHigh    - When high, indicates a new cell is coming
//                         in with the first byte starting on the high
//                         order bits of the input.
//          LatchSet       - Output of the SR-Latch which, if high,
//                         indicates there is new path data to be loaded
//                         into the memory lookup module.
//          LatchReset     - The negated state of the LatchSet input.
//
// Output:  PVRL           - Latch the results of the read from the
//                         memory lookup module.
//          RSRL           - Clear the new path information in the new
//                         path registers (by setting the SR-Latch
//                         indicating the validity of the data as
//                         being false)
//          LLODG26        - Start the low-byte counter at 26
//          LLODG27        - Start the low-byte counter at 27
//          LHODG27        - Start the high-byte counter at 27
//          VVRL           - Not applicable to "path-only" Analysis Mod.
//          RAS            - Row address select line on the memory
//                         lookup module
//          CAS            - Column address select line on the memory
//                         lookup module
//          W              - Read/Write control line on the memory
//                         lookup module
//          LowChoke       - Control line to the data gate that informs
//                     it whether to transmit the low byte of the
//                     data words exiting from the shift register
//      HighChoke         - Control line to the data gate that informs
//                     it whether to transmit the high byte of the
//                     data words exiting from the shift register
//      FourBDS     [5:0] - Control lines to the four bit multiplexer
//                     that shunt different portions of the
//                     incoming data words from the Receiver
//      FourBDL     [5:0] - Latch control lines on the latches that store
//                     the path information of the currently
//                     transiting cell
//      TwelveBDS    [1:0] - Control lines to the twelve bit by four line
//                     multiplexer that presents data from
//                     various latch groups to the memory lookup
//                     module

module SequenceDetect (Clock, NewCellLow, NewCellHigh, LatchSet, LatchReset,
            LookupRes, FourBDS, FourBDL, TwelveBDS, PVRL, RSRL,
            VVRL, RAS, CAS, W, LowChoke, HighChoke          );

  input      Clock, NewCellLow, NewCellHigh, LatchSet, LatchReset,
           LookupRes;
  output     PVRL, RSRL, VVRL, RAS, CAS, W;
  output     LowChoke, HighChoke;
  output  [1:0] TwelveBDS;
  output  [5:0] FourBDS, FourBDL;

  wire       LLODG26, LLODG27, LHODG27, Ground;
  wire    [4:0] Bit, BitInv, LowByte, HighByte;

  assign     Ground = 0;

  StateMachine  Core   (Clock, NewCellLow, NewCellHigh, LatchSet,
              LatchReset, Bit, BitInv               );
  StateControl  Signal (Bit, BitInv, NewCellLow, NewCellHigh, LookupRes,
              FourBDS, FourBDL, TwelveBDS, PVRL, RSRL,
              LLODG26, LLODG27, LHODG27, VVRL, RAS, CAS, W);

  DownCounterWithPreset LowByteCounter
              (Clock, LLODG26, LLODG27, LowByte[0], LowByte[1],
               LowByte[2], LowByte[3], LowByte[4]          );
  DownCounterWithPreset HighByteCounter
              (Clock, Ground, LHODG27, HighByte[0], HighByte[1],
               HighByte[2], HighByte[3], HighByte[4]          );
  FiveInputNANDGate    LowByteChoke
              (LowByte[0], LowByte[1], LowByte[2], LowByte[3],
               LowByte[4], LowChoke                  );
  FiveInputNANDGate    HighByteChoke
              (HighByte[0], HighByte[1], HighByte[2], HighByte[3],
               HighByte[4], HighChoke                  );

endmodule


// Let's bring the whole thing together

module NetworkSecurity;

  wire        Clock, NewCellLow, NewCellHigh, LatchSet, LatchReset,
            PVRL, RSRL, VVRL, RAS, CAS, W, LowChoke, HighChoke,
            NewPathDataIn, NewPathDataOut, LoadNewData, UnLoadNewData,
            PathState;
  wire   [1:0] TwelveBDS;
  wire   [4:0] StateBit, StateBitInv;
  wire   [5:0] FourBDS, FourBDL;
  wire  [11:0] RAMAddress;
  wire  [15:0] DataIn, ShiftOut, GateOut;
  wire  [23:0] NewPathAddressIn, NewPathAddressOut, LatchIn, LatchOut;

  initial
    begin
      // generate our report
```

```
   // $shm_open;
   // $shm_probe("AC");
   // #5000  $shm_close;
   #5000 $finish;
   // $monitor ($time,,
   //        "S0=%b  S1=%b  #1=%d  #2=%d  #3=%d  #4=%d  O=%d",
   //        Sel0, Sel1, One, Two, Three, Four, Out);
 end

ClockGen         Timer    (Clock);

NetworkReceiver    Receive  (Clock, NewCellLow, NewCellHigh, DataIn);
NetworkTransmitter Transmit (Clock, NewCellLow, NewCellHigh, GateOut);
ControlModule      PathGen  (NewPathAddressIn, NewPathDataIn, LoadNewData,
                   LatchSet, LatchReset                    );
ShiftRegister    Shifter  (Clock, DataIn, ShiftOut);
EightBitDataGate  LowGate  (ShiftOut[7:0], LowChoke, GateOut[7:0]);
EightBitDataGate  HighGate (ShiftOut[15:8], HighChoke, GateOut[15:8]);

SequenceDetect     Control  (Clock, NewCellLow, NewCellHigh, LatchSet,
                   LatchReset, PathState, FourBDS, FourBDL,
                   TwelveBDS, PVRL, RSRL, VVRL, RAS, CAS, W,
                   LowChoke, HighChoke                  );

NewPathStore       NewPath  (LoadNewData, UnLoadNewData, NewPathAddressIn,
                   NewPathDataIn, LatchSet, LatchReset,
                   NewPathAddressOut, NewPathDataOut          );

FourBitTwoLineSelector  S1  (DataIn[11:8],  DataIn[3:0],   FourBDS[0],
                   LatchIn[23:20]                  );
FourBitTwoLineSelector  S2  (DataIn[7:4],   DataIn[15:12], FourBDS[1],
                   LatchIn[19:16]                  );
FourBitTwoLineSelector  S3  (DataIn[3:0],   DataIn[11:8],  FourBDS[2],
                   LatchIn[15:12]                  );
FourBitTwoLineSelector  S4  (DataIn[15:12], DataIn[7:4],   FourBDS[3],
                   LatchIn[11:8]                   );
```

```
FourBitTwoLineSelector  S5  (DataIn[11:8],  DataIn[3:0],   FourBDS[4],
                    LatchIn[7:4]                    );
FourBitTwoLineSelector  S6  (DataIn[7:4],   DataIn[15:12], FourBDS[5],
                    LatchIn[3:0]                    );
FourBitRegister       L1  (FourBDL[0], LatchIn[23:20], LatchOut[23:20]);
FourBitRegister       L2  (FourBDL[1], LatchIn[19:16], LatchOut[19:16]);
FourBitRegister       L3  (FourBDL[2], LatchIn[15:12], LatchOut[15:12]);
FourBitRegister       L4  (FourBDL[3], LatchIn[11:8],  LatchOut[11:8] );
FourBitRegister       L5  (FourBDL[4], LatchIn[7:4],   LatchOut[7:4]  );
FourBitRegister       L6  (FourBDL[5], LatchIn[3:0],   LatchOut[3:0]  );

TwelveBitFourLineSelector SM(LatchOut[23:12],
                    LatchOut[11:00],
                    NewPathAddressOut[23:12],
                    NewPathAddressOut[11:0],
                    TwelveBDS, RAMAddress    );

DynamicRAM          Lookup (RAMAddress, RAS, CAS, W, NewPathDataOut,
                    PathState                     );


endmodule
```

APPENDIX B

"PATH AND VOLUME" ANALYSIS MODULE SIMULATION CODE

This appendix contains the Verilog hardware description language code necessary to implement a gate-level simulation of the "path and volume" version of the Analysis Module. The Receivers, Transmitters and memories involved in the design of the network security device were simulated at the procedural level and the Analysis Module was simulated at the gate level.

All sub-module inputs and outputs are fully commented.

B.A Verilog simulation

```
module Inverter (In, Out);

  input     In;
  output     Out;

  reg     Out;

  always
    #1 Out = ~In;

endmodule


module TwoInputNANDGate (InOne, InTwo, Out);

  input     InOne, InTwo;
  output     Out;

  reg     Out;

  always
```

```
    #1 Out = ~(InOne & InTwo);

endmodule

module ThreeInputNANDGate (InOne, InTwo, InThree, Out);

  input       InOne, InTwo, InThree;
  output      Out;

  reg         Out;

  always
    #1 Out = ~(InOne & InTwo & InThree);

endmodule


module FourInputNANDGate (InOne, InTwo, InThree, InFour, Out);

  input       InOne, InTwo, InThree, InFour;
  output      Out;

  reg         Out;

  always
    #1 Out = ~(InOne & InTwo & InThree & InFour);

endmodule



module FiveInputNANDGate (InOne, InTwo, InThree, InFour, InFive, Out);

  input       InOne, InTwo, InThree, InFour, InFive;
  output      Out;
  reg          Out;

  always
    #1 Out = ~(InOne & InTwo & InThree & InFour & InFive);

endmodule




module SixInputNANDGate (InOne, InTwo, InThree, InFour, InFive, InSix, Out);

  input        InOne, InTwo, InThree, InFour, InFive, InSix;
  output       Out;

  reg          Out;

  always
    #1 Out = ~(InOne & InTwo & InThree & InFour & InFive & InSix);

endmodule




module SevenInputNANDGate (InOne, InTwo, InThree, InFour, InFive, InSix,
                     InSeven, Out                    );

  input        InOne, InTwo, InThree, InFour, InFive, InSix, InSeven;
  output       Out;

  reg          Out;

  always
    #1 Out = ~(InOne & InTwo & InThree & InFour & InFive & InSix & InSeven);

endmodule
```

```
module NineInputNANDGate (InOne, InTwo, InThree, InFour, InFive, InSix,
                 InSeven, InEight, InNine, Out           );

  input     InOne, InTwo, InThree, InFour, InFive, InSix, InSeven,
          InEight, InNine;
  output    Out;

  reg       Out;

  always
    #1 Out = ~(InOne & InTwo & InThree & InFour & InFive & InSix & InSeven &
          InEight & InNine                        );

endmodule


module TwoInputNORGate (InOne, InTwo, Out);

  input     InOne, InTwo;
  output    Out;

  reg       Out;

  always
    #1 Out = ~(InOne | InTwo);

endmodule

module ThreeInputNORGate (InOne, InTwo, InThree, Out);

  input     InOne, InTwo, InThree;
  output    Out;
  reg        Out;

  always
    #1 Out = ~(InOne | InTwo | InThree);

endmodule


module FourInputNORGate (InOne, InTwo, InThree, InFour, Out);

  input      InOne, InTwo, InThree, InFour;
  output     Out;

  reg        Out;

  always
    #1 Out = ~(InOne | InTwo | InThree | InFour);

endmodule


module FiveInputNORGate (InOne, InTwo, InThree, InFour, InFive, Out);

  input      InOne, InTwo, InThree, InFour, InFive;
  output     Out;

  reg        Out;

  always
    #1 Out = ~(InOne | InTwo | InThree | InFour | InFive);

endmodule


module SixInputNORGate (InOne, InTwo, InThree, InFour, InFive, InSix, Out);
```

```
  input      InOne, InTwo, InThree, InFour, InFive, InSix;
  output     Out;

  reg        Out;

  always
    #1 Out = ~(InOne | InTwo | InThree | InFour | InFive | InSix);

endmodule



module SevenInputNORGate (InOne, InTwo, InThree, InFour,
                   InFive, InSix, InSeven, Out   );

  input      InOne, InTwo, InThree, InFour, InFive, InSix, InSeven;
  output     Out;

  reg        Out;

  always
    #1 Out = ~(InOne | InTwo | InThree | InFour | InFive | InSix | InSeven);

endmodule



module SRLatch (Set, Reset, Out, InvertOut);

  input      Set, Reset;
  output     Out, InvertOut;

  TwoInputNANDGate    GateOne    (Set, InvertOut, Out);
  TwoInputNANDGate    GateTwo    (Reset, Out, InvertOut);

endmodule
```

```
module PosEdgeTrigLatch (Clock, Data, Out, InvertOut);

  input      Clock, Data;
  output     Out, InvertOut;

  wire       w1, w2, w3, w4;

  TwoInputNANDGate    GateOne    (w4, w2, w1);
  TwoInputNANDGate    GateTwo    (w1, Clock, w2);
  ThreeInputNANDGate  GateThree  (w2, Clock, w4, w3);
  TwoInputNANDGate    GateFour   (w3, Data, w4);
  TwoInputNANDGate    GateFive   (w2, InvertOut, Out);
  TwoInputNANDGate    GateSix    (Out, w3, InvertOut);

endmodule



// Name:    FourBitRegister
//
// Inputs:  Data    [3:0] - The data to be latched by the register on the
//                          next rising clock edge.
//          Clock         - The clocking signal which controls data latching.
//
// Outputs: Out     [3:0] - The data latched on the last rising clock edge.

module FourBitRegister (Clock, Data, Out);

  input  [3:0] Data;
  input        Clock;
  output [3:0] Out;

  wire   [3:0] OutInv;

  PosEdgeTrigLatch   BitZero   (Clock, Data[0], Out[0], OutInv[0]);
  PosEdgeTrigLatch   BitOne    (Clock, Data[1], Out[1], OutInv[1]);
  PosEdgeTrigLatch   BitTwo    (Clock, Data[2], Out[2], OutInv[2]);
  PosEdgeTrigLatch    BitThree   (Clock, Data[3], Out[3], OutInv[3]);

endmodule



// Name:    EightBitRegister
//
// Inputs:  Data     [7:0] - The data to be latched by the register on the
//                     next rising clock edge.
//          Clock        - The clocking signal which controls data latching.
//
// Outputs: Out      [7:0] - The data latched on the last rising clock edge.

module EightBitRegister (Clock, Data, Out);

  input  [7:0]  Data;
  input         Clock;
  output [7:0]  Out;

  FourBitRegister    LowNibble  (Clock, Data[3:0], Out[3:0]);
  FourBitRegister    HighNibble (Clock, Data[7:4], Out[7:4]);

endmodule



// Name:    SixteenBitRegister
//
// Inputs:  Data    [15:0] - The data to be latched by the register on the
//                     next rising clock edge.
//          Clock        - The clocking signal which controls data latching.
//
// Outputs: Out     [15:0] - The data latched on the last rising clock edge.

module SixteenBitRegister (Clock, Data, Out);

  input  [15:0] Data;
  input       Clock;
  output [15:0] Out;

  EightBitRegister  LowByte  (Clock, Data[7:0],  Out[7:0] );
  EightBitRegister  HighByte (Clock, Data[15:8], Out[15:8]);

endmodule



// Name:    ShiftRegister
//
// Inputs:  Data    [15:0] - The data to be latched by the shift register on
//                   the next rising clock edge.
//        Clock       - The clocking signal which controls data latching.
//
// Outputs: Out    [15:0] - The data latched on the rising clock edge twenty
//                   seven clock cycles ago
//        OutInv  [15:0] - The negation of the data latched on the rising
//                   clock edge nine clock cycles ago.

module ShiftRegister (Clock, Data, Out);

  input  [15:0] Data;
  input       Clock;
  output [15:0] Out;

  wire   [15:0]  L1, L2, L3, L4, L5, L6, L7, L8;

  SixteenBitRegister Stage0  (Clock, Data, L1);
  SixteenBitRegister Stage1  (Clock, L1,   L2);
  SixteenBitRegister Stage2  (Clock, L2,   L3);
  SixteenBitRegister Stage3  (Clock, L3,   L4);
  SixteenBitRegister Stage4  (Clock, L4,   L5);
  SixteenBitRegister Stage5  (Clock, L5,   L6);
  SixteenBitRegister Stage6  (Clock, L6,   L7);
  SixteenBitRegister Stage7  (Clock, L7,   L8);
  SixteenBitRegister Stage8  (Clock, L8,   Out);

endmodule



// Name:   DataGate
//
// Inputs:  In           - Data input.
//          Select       - If high, the bit value at "In" will be reflected
//                         at "Out". Otherwise, "Out" will reflect zero.
//
// Outputs: Out          - Reflect "In" if Select is low, otherwise low
//                         regardless of the state of "In".

module DataGate (In, Select, Out);

  input      In, Select;
  output     Out;

  wire       OutInvert;

  TwoInputNANDGate   Gate      (In, Select, OutInvert);
  Inverter           Invert    (OutInvert, Out);

endmodule



// Name:   CounterGate
//
// Inputs:  InUp         - Data input to count up
//          InDown       - Data input to count down
//          InSame       - Data input to remain in same state
//          Incr         - If high, we must count up on next transition
//          Decr         - If high, we must count down on next transition
//
// Outputs: Out          - Reflect the value to be loaded for the next
//                    transition

module CounterGate (InUp, InDown, InSame, Incr, Decr, Out);

  input      InUp, InDown, InSame, Incr, Decr;
  output     Out;

  wire       IncrInv, DecrInv;
  wire   [3:0] Line;

  // Provide the inverted logic controls
  Inverter         Gate0 (Incr, IncrInv);
  Inverter         Gate1 (Decr, DecrInv);

  // Implcment the data gatc
  ThreeInputNANDGate  Gate2 (IncrInv, DecrInv, InSame, Line[0]);
  ThreeInputNANDGate  Gate3 (Incr,    Decr,    InSame, Line[1]);
  ThreeInputNANDGate  Gate4 (IncrInv, Decr,    InDown, Line[2]);
  ThreeInputNANDGate  Gate5 (Incr,    DecrInv, InUp,   Line[3]);
  FourInputNANDGate   Gatc6 (Line[0], Line[1], Line[2], Line[3], Out);

endmodule



// Name:    BitEqualTest
//
// Inputs:  InZero        - First data input
//          InZeroInv     - Negation of first data input
//          InOne         - Second data input
//          InOneInv      - Negation of second data input
//
// Outputs: Result        - Result of equality test

module BitEqualTest (InZero, InZeroInv, InOne, InOneInv, Result);

  input      InZero, InZeroInv, InOne, InOneInv;
  output      Result;

  wire   [1:0] Line;

  TwoInputNANDGate   Gate0  (InZeroInv, InOneInv, Line[0]);
  TwoInputNANDGate   Gate1  (InZero, InOne, Line[1]);
  TwoInputNANDGate   Gate2  (Line[0], Line[1], Result);

endmodule


// Name:    ThreeBitDataGate
//
// Inputs:  In        [2:0] - Data input.
//          Select          - If asserted high, the bit values at "In" will be
//                            reflected at "Out". Otherwise, "Out" will
//                            reflect all zeroes.
//
// Outputs: Out       [2:0] - Reflect "In" if Select is high, otherwise just
//                            set all bits to low.

module ThreeBitDataGate (In, Select, Out);

  input  [2:0]  In;
  input         Select;
  output [2:0]  Out;

  DataGate   Bit0 (In[0], Select, Out[0]);
  DataGate   Bit1 (In[1], Select, Out[1]);
  DataGate   Bit2 (In[2], Select, Out[2]);

endmodule


// Name:    FourBitDataGate
//
```

```
// Inputs:  In       [3:0] - Data input.
//          Selcct         - If asserted high, the bit values at "In" will bc
//                         reflected at "Out". Otherwise, "Out" will
//                         reflect all zeroes.
//
// Outputs: Out      [3:0] - Reflect "In" if Select is high, otherwise just
//                         set all bits to low.

module FourBitDataGate (In, Select, Out);

  input  [3:0]  In;
  input         Select;
  output [3:0]  Out;

  DataGate  Bit0 (In[0], Select, Out[0]);
  DataGate  Bit1 (In[1], Select, Out[1]);
  DataGate  Bit2 (In[2], Select, Out[2]);
  DataGate  Bit3 (In[3], Select, Out[3]);

endmodule


// Name:    EightBitDataGate
//
// Inputs:  In[7:0]        - Data input.
//          Selcct         - If asserted high, the bit values at "In" will bc
//                         reflected at "Out". Otherwise, "Out" will
//                         reflect all zeroes.
//
// Outputs: Out[7:0]       - Reflect "In" if Select is high, otherwise just
//                         go low on all bits.

module EightBitDataGate (In, Select, Out);

  input  [7:0] In;
  input        Select;
  output [7:0] Out;

  FourBitDataGate  LowNibble  (In[3:0], Select, Out[3:0]);
  FourBitDataGate  HighNibble (In[7:4], Select, Out[7:4]);

endmodule


// Name:    TwoLineSelector
//
// Inputs:  In[1:0]      - Two bits of data input
//          Select       - Input that must be asserted in order to control
//                       which of the two bits of input will be reflected
//                       at the output.
//          SelectInv    - Input which is the inverse of "Select"
//
// Outputs: Out          - Reflect the value at "In[1]" if "Select" is
//                       high and "SelectInv" is low. Reflect the value
//                       at "In[0]" if "Select" is low and "SelectInv" is
//                       high. Behavior is unpredictable otherwise.

module TwoLineSelector (In, Select, SelectInv, Out);

  input  [1:0]  In;
  input         Select, SelectInv;
  output        Out;

  wire  [1:0]  Con;

  TwoInputNANDGate   GateZero  (In[0],  SelectInv, Con[0]);
  TwoInputNANDGate   GateOne   (In[1],  Select,    Con[1]);
  TwoInputNANDGate   GateTwo   (Con[0], Con[1],    Out);

endmodule
```

```
// Name:   FourBitTwoLineSelector
//
// Inputs:  InZero  [3:0] - The first input line
//          InOne   [3:0] - The second input line
//          Select        - Input that must be asserted in order to control
//                          which of the two nibbles of input will be
//                          reflected at the output nibble.
//
// Outputs: Out     [3:0] - Reflect the nibble at "InOne" if "Select" is
//                          high. Otherwise, reflect the nibble at "InTwo".

module FourBitTwoLineSelector (InZero, InOne, Select, Out);


  input  [3:0]  InZero, InOne;
  input         Select;
  output [3:0]  Out;


  wire          SelectInv;
  wire   [7:0]  Input;


  assign    Input[0]  = InZero[0], Input[1]  = InOne[0],
            Input[2]  = InZero[1], Input[3]  = InOne[1],
            Input[4]  = InZero[2], Input[5]  = InOne[2],
            Input[6]  = InZero[3], Input[7]  = InOne[3];



  Inverter      Invert  (Select, SelectInv);
  TwoLineSelector Select0 (Input[1:0], Select, SelectInv, Out[0]);
  TwoLineSelector Select1 (Input[3:2], Select, SelectInv, Out[1]);
  TwoLineSelector Select2 (Input[5:4], Select, SelectInv, Out[2]);
  TwoLineSelector Select3 (Input[7:6], Select, SelectInv, Out[3]);

endmodule
```

```
// Name:    FourLineSelector
//
// Inputs:  In[3:0]         - Four bits of data input
//          Select[1:0]     - Inputs that must be asserted in order to control
//                            which of the four bits of input will be
//                            reflected at the output.
//          SelectInv[1:0]  - Input which is the inverse of "Select[1:0]" on
//                            all bits.
//
// Outputs: Out             - Depending on the state of the "Select" inputs,
//                            this signal will reflect the state of one of the
//                            bits at the "In" input, according to the table
//                            below. Behavior is unpredictable for conditions
//                            not covered in the table.
//
//                            Sel[0]  SelInv[0]  Sel[1]  SelInv[1]  | Out
//                            --------------------------------------+-------
//                             Low     High       Low     High      | In[0]
//                             Low     High       High    Low       | In[1]
//                             High    Low        Low     High      | In[2]
//                             High    Low        High    Low       | In[3]

module FourLineSelector (In, Select, SelectInv, Out);

  input  [3:0]  In;
  input  [1:0]  Select, SelectInv;
  output        Out;

  wire   [3:0]  Con;

  ThreeInputNANDGate  GateZero  (In[0], SelectInv[0], SelectInv[1], Con[0]);
  ThreeInputNANDGate  GateOne   (In[1], SelectInv[0], Select[1]   , Con[1]);
  ThreeInputNANDGate  GateTwo   (In[2], Select[0],    SelectInv[1], Con[2]);
  ThreeInputNANDGate  GateThree (In[3], Select[0],    Select[1],    Con[3]);
  FourInputNANDGate   GateFour  (Con[0], Con[1], Con[2], Con[3], Out);
```

```
endmodule


// Name:    TwelveBitFourLineSelector
//
// Inputs:  InZero  [11:0] - The first input line
//          InOne   [11:0] - The second input line
//          InTwo   [11:0] - The third input line
//          InThree [11:0] - The fourth input line
//          Select  [1:0]  - Controls whose state govern which of the four
//                           inputs will be reflected at the output.
//                           reflected at the output.
//
// Outputs: Out     [11:0] - Depending on the state of the "Select" inputs,
//                           this signal will reflect the state of the twelve
//                           bits at one of the four inputs. Behavior is
//                           unpredictable for conditions not covered in the
//                           table.
//                                Select0  Select1 | Out
//                                -----------------+----------
//                                 Low     Low     | InZero
//                                 Low     High    | InOne
//                                 High    Low     | InTwo
//                                 High    High    | InThree

module TwelveBitFourLineSelector (InZero, InOne, InTwo, InThree, Select, Out);

  input  [11:0] InZero, InOne, InTwo, InThree;
  input  [1:0]  Select;
  output [11:0] Out;

  wire  [1:0]  SelectInv;
  wire  [47:0] Input;

  assign     Input[0]  = InZero[0],   Input[1]  = InOne[0],
           Input[2]  = InTwo[0],    Input[3]  = InThree[0],
        Input[4]  = InZero[1],   Input[5]  = InOne[1],
        Input[6]  = InTwo[1],    Input[7]  = InThree[1],
        Input[8]  = InZero[2],   Input[9]  = InOne[2],
        Input[10] = InTwo[2],    Input[11] = InThree[2],
        Input[12] = InZero[3],   Input[13] = InOne[3],
        Input[14] = InTwo[3],    Input[15] = InThree[3],
        Input[16] = InZero[4],   Input[17] = InOne[4],
        Input[18] = InTwo[4],    Input[19] = InThree[4],
        Input[20] = InZero[5],   Input[21] = InOne[5],
        Input[22] = InTwo[5],    Input[23] = InThree[5],
        Input[24] = InZero[6],   Input[25] = InOne[6],
        Input[26] = InTwo[6],    Input[27] = InThree[6],
        Input[28] = InZero[7],   Input[29] = InOne[7],
        Input[30] = InTwo[7],    Input[31] = InThree[7],
        Input[32] = InZero[8],   Input[33] = InOne[8],
        Input[34] = InTwo[8],    Input[35] = InThree[8],
        Input[36] = InZero[9],   Input[37] = InOne[9],
        Input[38] = InTwo[9],    Input[39] = InThree[9],
        Input[40] = InZero[10],  Input[41] = InOne[10],
        Input[42] = InTwo[10],   Input[43] = InThree[10],
        Input[44] = InZero[11],  Input[45] = InOne[11],
        Input[46] = InTwo[11],   Input[47] = InThree[11];



Inverter     Invert0  (Select[0], SelectInv[0]);
Inverter     Invert1  (Select[1], SelectInv[1]);
FourLineSelector Select0  (Input[3:0],   Select, SelectInv, Out[0] );
FourLineSelector Select1  (Input[7:4],   Select, SelectInv, Out[1] );
FourLineSelector Select2  (Input[11:8],  Select, SelectInv, Out[2] );
FourLineSelector Select3  (Input[15:12], Select, SelectInv, Out[3] );
FourLineSelector Select4  (Input[19:16], Select, SelectInv, Out[4] );
FourLineSelector Select5  (Input[23:20], Select, SelectInv, Out[5] );
FourLineSelector Select6  (Input[27:24], Select, SelectInv, Out[6] );
FourLineSelector Select7  (Input[31:28], Select, SelectInv, Out[7] );
FourLineSelector Select8  (Input[35:32], Select, SelectInv, Out[8] );
```

```
  FourLineSelector Select9  (Input[39:36], Select, SelectInv, Out[9] );
  FourLineSelector Select10 (Input[43:40], Select, SelectInv, Out[10]);
  FourLineSelector Select11 (Input[47:44], Select, SelectInv, Out[11]);

endmodule



// Name:    ThreeBySevenDemux
//
// Inputs:  In      [2:0]  - Input lines to be demultiplexed (all zero bits
//                           state is not decoded)
//
// Outputs: Out     [6:0]  - Output lines to be asserted based on the state
//                           of the input lines

module ThreeBySevenDemux (In, Out);

  input   [2:0] In;
  output  [6:0] Out;

  wire    [2:0] InInv;
  wire    [6:0] OutInv;

  Inverter         Gate0  (In[0], InInv[0]);
  Inverter         Gate1  (In[1], InInv[1]);
  Inverter         Gate2  (In[2], InInv[2]);
  ThreeInputNANDGate Gate3  (InInv[0], InInv[1], In[2],    OutInv[0]);
  ThreeInputNANDGate Gate4  (InInv[0], In[1],    InInv[2], OutInv[1]);
  ThreeInputNANDGate Gate5  (InInv[0], In[1],    In[2],    OutInv[2]);
  ThreeInputNANDGate Gate6  (In[0],    InInv[1], InInv[2], OutInv[3]);
  ThreeInputNANDGate Gate7  (In[0],    InInv[1], In[2],    OutInv[4]);
  ThreeInputNANDGate Gate8  (In[0],    In[1],    InInv[2], OutInv[5]);
  ThreeInputNANDGate Gate9  (In[0],    In[1],    In[2],    OutInv[6]);
  Inverter         Gate10 (OutInv[0], Out[0]);
  Inverter         Gate11 (OutInv[1], Out[1]);
  Inverter         Gate12 (OutInv[2], Out[2]);
  Inverter      Gate13 (OutInv[3], Out[3]);
  Inverter      Gate14 (OutInv[4], Out[4]);
  Inverter      Gate15 (OutInv[5], Out[5]);
  Inverter      Gate16 (OutInv[6], Out[6]);

endmodule


// Name:    ClockGen
//
// Inputs:  None.
//
// Outputs: Clock        - Square wave that cycles up and down every 13 nsec
//                         thereby producing a signal with a period of
//                         26 nsec.

module ClockGen (Clock);

  output      Clock;

  reg         Clock;

  initial
    Clock = 1;

  always
    begin
      #13 Clock = 0;
      #13 Clock = 1;
    end

endmodule


// Name:    NewPathStore    - Simulates the storage elements that accept and
//                            hold data about a new path to be loaded into
//                  the memory lookup module by the sequence /
//                  detect module at the appropriate time
//
// Inputs: Load         - The Set input on the SR latch indicating
//                  whether the unit still contains new data
//      UnLoad          - The Reset input on the SR latch indicating
//                  whether the unit still contains new data
//      DataIn          - Input indicating how the new path is to be
//                  validated or invalidated
//      AddressIn [23:0]- The input for the new path which is to be
//                  validated or invalidated
//
// Outputs: Full        - The Q output on the SR latch which, if high,
//                  indicates the unit contains new data.
//      Empty           - The Q' output on the SR latch which, if high,
//                  indicates the unit does not contain new data.
//      DataOut         - Output indicating whether the new path
//                  currently stored is to be validated or
//                  invalidated
//      AddressOut[23:0]- The output of the new path which is to be
//                  validated or invalidated

module NewPathStore (Load, UnLoad, AddressIn,  DataIn,
          Full, Empty,  AddressOut, DataOut);

   input      Load, UnLoad;
   input  [6:0] DataIn;
   input  [23:0] AddressIn;
   output      Full, Empty;
   output  [6:0] DataOut;
   output [23:0] AddressOut;

   wire   [6:0] DataOutInv;

   EightBitRegister Low    (Load, AddressIn[7:0],   AddressOut[7:0]  );
   EightBitRegister Middle (Load, AddressIn[15:8],  AddressOut[15:8] );
```

```
  EightBitRegister High   (Load, AddressIn[23:16], AddressOut[23:16]);
  PosEdgeTrigLatch Dat0   (Load, DataIn[0], DataOut[0], DataOutInv[0]);
  PosEdgeTrigLatch Dat1   (Load, DataIn[1], DataOut[1], DataOutInv[1]);
  PosEdgeTrigLatch Dat2   (Load, DataIn[2], DataOut[2], DataOutInv[2]);
  PosEdgeTrigLatch Dat3   (Load, DataIn[3], DataOut[3], DataOutInv[3]);
  PosEdgeTrigLatch Dat4   (Load, DataIn[4], DataOut[4], DataOutInv[4]);
  PosEdgeTrigLatch Dat5   (Load, DataIn[5], DataOut[5], DataOutInv[5]);
  PosEdgeTrigLatch Dat6   (Load, DataIn[6], DataOut[6], DataOutInv[6]);
  SRLatch         Status (Load, UnLoad, Full, Empty);

endmodule



// Name:   DynamicRAM     - Simulates a Texas Instruments SMJ416100-70
//                    dynamic random access memory
//
// Inputs:  Address [11:0]  - DRAM address lines
//         RAS           - Row address select
//         CAS           - Column address select
//         W             - Read/Write select
//         D             - Data input on memory writes
//
// Outputs: Q            - Data output on memory reads

module DynamicRAM (Address, RAS, CAS, W, D, Q);

  input  [11:0] Address;
  input        RAS, CAS, W, D;
  output       Q;

  reg   [11:0] Row, Column;
  reg          Q, DataIn;

  initial
    Q = 1'bz;
```

```
 always
   begin
     wait (!RAS)
     Row = Address;
     wait (!CAS)
     Column = Address;
     if (W == 0)
       begin
         // we are performing a write cycle
         DataIn = D;
         wait (CAS)
         Q = 1'bz;
       end
     else
       begin
         // we are performing a read cycle
         // for this simulation just present the low bit of the address
         #18 Q = Address[0];
         wait (CAS)
         Q = 1'bz;
       end
   end

endmodule


// Name:   NetworkReceiver
//
// Inputs:  Clock             - Clock on whose negative edge to present
//                          data.
//
// Outputs: Out       [15:0]  - Present data produced by the receiver.
//       NewCellEven         - Asserted when the starting byte of the cell
//                          currently being transmitted was presented on
//                          the high-order byte of the output.
//       NewCellOdd          - Asserted when the starting byte of the cell
//                          currently being transmitted was presented on
//                          the low-order byte of the output.

module NetworkReceiver (Clock, NewCellLow, NewCellHigh, Out);

  input       Clock;
  output [15:0] Out;
  output      NewCellLow, NewCellHigh;

  reg   [15:0] Out, Temp;
  reg          NewCellLow, NewCellHigh;

  initial
    begin
      @ (negedge Clock) Out[15:8]   = 8'b00000000;
                        Out[7:0]    = 8'b00000001;
                        NewCellLow  = 0;
                        NewCellHigh = 0;
    end

  always
    begin
      @ (negedge Clock) Temp[15:8] = Out[15:8] + 2;
                        Temp[7:0]  = Out[7:0] + 2;

                        if (Temp[15:8] > 52)
                          begin
                            Temp[15:8] = Temp[15:8] - 53;
                            if (Temp[15:8] == 0) NewCellLow = 1;
                          end
                        else
                          NewCellLow = 0;

                        if (Temp[7:0]  > 52)
                          begin
                            Temp[7:0]  = Temp[7:0]  - 53;
```

```
                    if (Temp[7:0] == 0) NewCellHigh = 1;
                  end
                else
                  NewCellHigh = 0;

                Out[15:0] = Temp[15:0];
    end

endmodule



// Name:    NetworkTransmitter
//
// Inputs:  Data       [15:0] - The data to be transmitted out onto the
//                       network.
//       NewCellEven       - Asserted when the starting byte of the cell
//                       currently being transmitted was presented on
//                       the high-order byte of the input.
//       NewCellOdd        - Asserted when the starting byte of the cell
//                       currently being transmitted was presented on
//                       the low-order byte of the input.

module NetworkTransmitter (Clock, NewCellEven,  NewCellOdd,  Data);

  input        Clock, NewCellEven, NewCellOdd;
  input  [15:0] Data;

endmodule



module ResetControl (Clock, Input, Output);
  input        Clock, Input;
  output       Output;

  reg          Output;
  initial
    begin
      Output = 0;
      #26 Output = Input;
      // @(negedge Clock) Output = Input;
    end

  always
    Output = Input;

endmodule



// Name:    ControlModule
//
// Inputs:  LatchSet        - If high, indicates that the new path storage
//                          module still contains new data.
//          LatchReset      - If high, indicates that the new path storage
//                          module has been cleared of new data.
//
// Output:  SetLatch        - If high, indicates that new data has been
//                          presented and should be latched.
//          Data            - If high, indicates that the new path
//                          being modified is to be a valid path.
//                          Otherwise, the new path is to be an invalid
//                          one.
//          Address         - Indicates the VPI/VCI pair of the path whose
//                          status is to be modified.

module ControlModule (Address, Data, SetLatch, LatchSet, LatchReset);

  input       LatchSet, LatchReset;
  output      SetLatch;
  output  [6:0] Data;
  output [23:0] Address;
  reg         SetLatch;
  reg    [6:0] Data;
  reg   [23:0] Address;

  initial
    begin
      SetLatch = 0; Data = 0; Address = 0;
    end

  always
    begin
      #1 if (LatchSet == 0)
          begin
            Address = Address + 1;
// For the purposes of this simulation, assign the lower
            // three bits of the address to point to the window
            // control module and let the next four higher order bits
            // be the value the gets loaded into the window control
            // module's trigger register
            Data[0] = Address[0];
            Data[1] = Address[1];
            Data[2] = Address[2];
            Data[3] = Address[3];
            Data[4] = Address[4];
            Data[5] = Address[5];
            Data[6] = Address[6];
            #1 SetLatch = 1;
            #1 SetLatch = 0;
          end
    end

 endmodule


// Name:   DownCounterWithPreset (this counter does not roll over)
//
```

```
// Inputs:  Clock          - Signal on whose positive edge, the counter
//                          must change state
//        Set26            - If high on a rising edge of "Clock", then
//                          it forces the next state of the counter to
//                          be 26 transitions away from zero.
//        Set27            - If high on a rising edge of "Clock", then
//                          it forces the next state of the counter to
//                          be 27 transitions away from zero.
//
// Output:  Bit0...4       - Individual lines of the output of the five
//                          latches that store the current state of the
//                          counter. Bit0 refers to the lowest order
//                          bit and Bit4 to the highest order bit.

module DownCounterWithPreset (Clock, Set26, Set27,
                    Bit0, Bit1, Bit2, Bit3, Bit4);

  input     Clock, Set26, Set27;
  output    Bit0, Bit1, Bit2, Bit3, Bit4;

  wire      Bit0Input, Bit1Input, Bit2Input, Bit3Input, Bit4Input;
  wire      Set26Inv, Set27Inv;
  wire  [22:0] Line;

  // Memory elements to store the current state
  PosEdgeTrigLatch   BitZero  (Clock, Bit0Input, Bit0, Bit0Inv);
  PosEdgeTrigLatch   BitOne   (Clock, Bit1Input, Bit1, Bit1Inv);
  PosEdgeTrigLatch   BitTwo   (Clock, Bit2Input, Bit2, Bit2Inv);
  PosEdgeTrigLatch   BitThree (Clock, Bit3Input, Bit3, Bit3Inv);
  PosEdgeTrigLatch   BitFour  (Clock, Bit4Input, Bit4, Bit4Inv);

  // Prepare inputs
  Inverter       Gate0   (Set26, Set26Inv);
  Inverter       Gate1   (Set27, Set27Inv);

  // Decode logic for bit 0
```

```
TwoInputNORGate    Gate2   (Bit0Inv, Set27, Line[2]);
TwoInputNORGate    Gate3   (Set26Inv, Set27, Line[3]);
SixInputNORGate    Gate4   (Bit0, Bit1, Bit2, Bit3, Bit4, Set27,
                   Line[4]                  );
ThreeInputNORGate  Gate5   (Line[2], Line[3], Line[4], Bit0Input);

// Decode logic for bit 1
FourInputNORGate   Gate6   (Bit0, Bit1Inv, Set26, Set27, Line[6]);
FourInputNORGate   Gate7   (Bit0Inv, Bit1, Set26, Set27, Line[7]);
SevenInputNORGate  Gate8   (Bit0, Bit1, Bit2, Bit3, Bit4, Set26, Set27,
                   Line[8]                        );
ThreeInputNORGate  Gate9   (Line[6], Line[7], Line[8], Bit1Input);

// Decode logic for bit 2
TwoInputNORGate    Gate10  (Bit1Inv, Bit2, Line[10]);
TwoInputNORGate    Gate11  (Bit0Inv, Bit2, Line[11]);
ThreeInputNORGate  Gate12  (Bit0, Bit1, Bit2Inv, Line[12]);
FiveInputNORGate   Gate13  (Bit0, Bit1, Bit2, Bit3, Bit4, Line[13]);
SixInputNORGate    Gate14  (Line[10], Line[11], Line[12], Line[13],
                   Set26, Set27, Bit2Input          );

// Decode logic for bit 3
FourInputNORGate   Gate15  (Bit2Inv, Bit3, Set26, Set27, Line[15]);
FourInputNORGate   Gate16  (Bit1Inv, Bit3, Set26, Set27, Line[16]);
FourInputNORGate   Gate17  (Bit0Inv, Bit3, Set26, Set27, Line[17]);
SixInputNORGate    Gate18  (Bit0, Bit1, Bit2, Bit3Inv, Set26, Set27,
                   Line[18]                     );
SevenInputNORGate  Gate19  (Bit0, Bit1, Bit2, Bit3, Bit4, Set26, Set27,
                   Line[19]                        );
FiveInputNORGate   Gate20  (Line[15], Line[16], Line[17], Line[18],
                   Line[19], Bit3Input               );

// Decode logic for bit 4
ThreeInputNORGate  Gate21  (Bit4, Set26, Set27, Line[21]);
SixInputNORGate    Gate22  (Bit0, Bit1, Bit2, Bit3, Set26, Set27,
                   Line[22]                     );
```

```
  TwoInputNORGate    Gate23   (Line[21], Line[22], Bit4Input);

endmodule


// Name:    StateControl
//
// Inputs:  NewCellLow        - When high, indicates a new cell is coming
//                            in with the first byte starting on the low
//                            order bits of the input.
//       NewCellHigh       - When high, indicates a new cell is coming
//                            in with the first byte starting on the high
//                            order bits of the input.
//       LookupRes         - Result of the path lookup.
//       Bit       [4:0] - The state of the five bits which define the
//                            current state of the state machine for which
//                            the control lines must be decoded.
//       BitInv     [4:0] - The negated state of the five bits specified
//                            by the "Bit" input.
//
// Output:  PVRL             - Latch the results of the read from the
//                            memory lookup module.
//       RSRL              - Clear the new path information in the new
//                            path registers (by setting the SR-Latch
//                            indicating the validity of the data as
//                            being false)
//       LLODG26           - Start the low-byte counter at 26
//       LLODG27           - Start the low-byte counter at 27
//       LHODG27           - Start the high-byte counter at 27
//       VVRL              - Not applicable to "path-only" Analysis Mod.
//       RAS               - Row address select line on the memory
//                            lookup module
//       CAS               - Column address select line on the memory
//                            lookup module
//       W                 - Read/Write control line on the memory
//                            lookup module
//      FourBDS     [5:0] - Control lines to the four bit multiplexer
//                  that shunt different portions of the
//                  incoming data words from the Receiver
//      FourBDL     [5:0] - Latch control lines on the latches that store
//                  the path information of the currently
//                  transiting cell
//      TwelveBDS    [1:0] - Control lines to the twelve bit by four line
//                  multiplexer that presents data from
//                  various latch groups to the memory lookup
//                  module

module StateControl (Bit, BitInv, NewCellLow, NewCellHigh, LookupRes,
              FourBDS, FourBDL, TwelveBDS, PVRL, RSRL,
              LLODG26, LLODG27, LHODG27, VVRL, RAS, CAS, W);

  input       NewCellLow, NewCellHigh, LookupRes;
  input  [4:0] Bit, BitInv;
  output      PVRL, RSRL, LLODG26, LLODG27, LHODG27, VVRL, RAS, CAS, W;
  output  [5:0] FourBDS, FourBDL;
  output  [1:0] TwelveBDS;

  wire        LowStart, HighStart;
  wire   [4:0] Stage;
  wire   [28:0] Line;

  assign      FourBDL[0]   = Stage[1],  FourBDL[1]   = Stage[2],
           FourBDL[2]   = Stage[2],  FourBDL[3]   = Stage[3],
           FourBDL[4]   = Stage[3],  FourBDL[5]   = Stage[4],
           TwelveBDS[0] = BitInv[2],  TwelveBDS[1] = Bit[4],
           LLODG26      = LowStart,  LHODG27      = LowStart,
           LLODG27      = HighStart;
  // Logic for PVRL
  FiveInputNANDGate  Gate0   (BitInv[4], BitInv[3], Bit[2], BitInv[1],
                        Bit[0], Line[0]                    );
  Inverter         Gate1   (Line[0], PVRL);
```

```
// Logic for RSRL
FiveInputNANDGate   Gate2    (Bit[4], Bit[3], Bit[2], BitInv[1], Bit[0],
                    Line[2]                        );
Inverter        Gate3    (Line[2], RSRL);

// Logic for LxODG2x
SevenInputNANDGate  Gate4    (BitInv[4], Bit[3], Bit[2], BitInv[1],
                    BitInv[0], LookupRes, NewCellLow, Line[4]);
SevenInputNANDGate  Gate5    (BitInv[4], Bit[3], Bit[2], BitInv[1],
                    BitInv[0], LookupRes, NewCellHigh, Line[5]);
Inverter        Gate6    (Line[4], LowStart);
Inverter        Gate7    (Line[5], HighStart);

// Logic for VVRL
FiveInputNANDGate   Gate8    (BitInv[4], BitInv[3], Bit[2], BitInv[1],
                    BitInv[0], Line[8]                 );
Inverter        Gate9    (Line[8], VVRL);

// Logic for RAS
TwoInputNANDGate    Gate10   (Bit[4], BitInv[3], Line[10]);
TwoInputNANDGate    Gate11   (BitInv[2], Bit[0], Line[11]);
ThreeInputNANDGate  Gate12   (Bit[4], Bit[2], BitInv[1], Line[12]);
ThreeInputNANDGate  Gate13   (Bit[3], BitInv[1], Bit[0], Line[13]);
ThreeInputNANDGate  Gate14   (BitInv[4], BitInv[1], BitInv[0], Line[14]);
FiveInputNANDGate   Gate15   (Line[10], Line[11], Line[12], Line[13],
                    Line[14], RAS                  );

// Logic for CAS
TwoInputNANDGate    Gate16   (BitInv[3], BitInv[2], Line[16]);
ThreeInputNANDGate  Gate17   (BitInv[3], Bit[1], BitInv[0], Line[17]);
FourInputNANDGate   Gate18   (Bit[4], Bit[3], Bit[1], BitInv[0], Line[18]);
FourInputNANDGate   Gate19   (Bit[4], Bit[3], BitInv[1], Bit[0], Line[19]);
FourInputNANDGate   Gate20   (Bit[4], Bit[3], BitInv[2], Bit[1], Line[20]);
FiveInputNANDGate   Gate21   (Line[16], Line[17], Line[18], Line[19],
                    Line[20], CAS                  );
```

```
  // Logic for W
  FiveInputNANDGate   Gate22   (Bit[4], Bit[3], Bit[2], Bit[1], Bit[0], W);

  // Logic for 4BDL
  FiveInputNANDGate   Gate23   (BitInv[4], BitInv[3], BitInv[2], BitInv[1],
                               Bit[0], Line[23]                    );
  FiveInputNANDGate   Gate24   (Bit[4], BitInv[3], Bit[2], BitInv[1],
                               BitInv[0], Line[24]          );
  FiveInputNANDGate   Gate25   (Bit[4], BitInv[3], Bit[2], BitInv[1],
                               Bit[0], Line[25]             );
  FiveInputNANDGate   Gate26   (BitInv[4], BitInv[3], BitInv[2], Bit[1],
                               Bit[0], Line[26]                );
  FiveInputNANDGate   Gate27   (Bit[4], BitInv[3], Bit[2], Bit[1], Bit[0],
                               Line[27]                        );
  TwoInputNANDGate    Gate28   (Line[23], Line[24], Stage[1]);
  TwoInputNANDGate    Gate29   (Line[23], Line[25], Stage[2]);
  TwoInputNANDGate    Gate30   (Line[26], Line[25], Stage[3]);
  TwoInputNANDGate    Gate31   (Line[26], Line[27], Stage[4]);

  // No logic block necessary for 12BDS[B,S]

endmodule


// Name:    StateMachine
//
// Inputs:  Clock           - Signal on whose rising edge the state
//                          machine must make a state change
//          NewCellLow      - When high, indicates a new cell is coming
//                          in with the first byte starting on the low
//                          order bits of the input.
//          NewCellHigh     - When high, indicates a new cell is coming
//                          in with the first byte starting on the high
//                          order bits of the input.
//          LatchSet        - Output of the SR-Latch which, if high,
//                          indicates there is new path data to be loaded
//                          into the memory lookup module.
//        LatchReset        - The negated state of the LatchSet input.
//
// Output:  Bit          [4:0] - The state of the five bits which define the
//                          current state of the state machine.
//        BitInv      [4:0] - The negated state of the five bits specified
//                          by the "Bit" input.

module StateMachine (Clock, NewCellLow, NewCellHigh, LatchSet, LatchReset,
              Bit, BitInv                              );

  input       Clock, NewCellLow, NewCellHigh, LatchSet, LatchReset;
  output  [4:0] Bit, BitInv;

  wire   [4:0] BitDecode, BitInput;
  wire   [39:0] Line;


  // Memory elements to store the current state
  PosEdgeTrigLatch   BitZero  (Clock, BitInput[0], Bit[0], BitInv[0]);
  PosEdgeTrigLatch   BitOne   (Clock, BitInput[1], Bit[1], BitInv[1]);
  PosEdgeTrigLatch   BitTwo   (Clock, BitInput[2], Bit[2], BitInv[2]);
  PosEdgeTrigLatch   BitThree (Clock, BitInput[3], Bit[3], BitInv[3]);
  PosEdgeTrigLatch   BitFour  (Clock, BitInput[4], Bit[4], BitInv[4]);

  // Decode logic for bit 0
  ThreeInputNANDGate  Gate0   (Bit[3], Bit[2], BitInv[1], Line[0]);
  ThreeInputNANDGate  Gate1   (Bit[4], Bit[3], Bit[2], Line[1]);
  ThreeInputNANDGate  Gate2   (Bit[4], BitInv[1], Bit[0], Line[2]);
  ThreeInputNANDGate  Gate3   (Bit[4], Bit[3], BitInv[0], Line[3]);
  FourInputNANDGate   Gate4   (BitInv[4], Bit[3], BitInv[2], Bit[1],
                      Line[4]                    );
  FourInputNANDGate   Gate5   (BitInv[4], BitInv[3], Bit[2], Bit[1],
                      Line[5]                    );
  FourInputNANDGate   Gate6   (BitInv[3], BitInv[2], BitInv[1], Bit[0],
                      Line[6]                    );
```

```
FiveInputNANDGate   Gate7   (BitInv[3], BitInv[2], BitInv[1], BitInv[0],
                    NewCellLow, Line[7]                       );
FiveInputNANDGate   Gate8   (Bit[4], BitInv[3], BitInv[1], BitInv[0],
                    NewCellHigh, Line[8]                  );
NineInputNANDGate   Gate9   (Line[0], Line[1], Line[2], Line[3],
                    Line[4], Line[9], Line[6], Line[7],
                    Line[8], BitDecode[0]           );
ResetControl        Reset0  (Clock, BitDecode[0], BitInput[0]);

// Decode logic for bit 1
TwoInputNANDGate    Gate10  (Bit[1], BitInv[0], Line[10]);
ThreeInputNANDGate  Gate11  (BitInv[3], BitInv[2], Bit[0], Line[11]);
ThreeInputNANDGate  Gate12  (Bit[4], BitInv[3], Bit[0], Line[12]);
FourInputNANDGate   Gate13  (BitInv[4], Bit[3], Bit[2], Bit[0], Line[13]);
FourInputNANDGate   Gate14  (Bit[4], BitInv[2], BitInv[1], Bit[0],
                    Line[14]                    );
FiveInputNANDGate   Gate15  (Bit[4], BitInv[2], Bit[1], Bit[0],
                    LatchSet, Line[15]           );
SixInputNANDGate    Gate16  (Line[10], Line[11], Line[12], Line[13],
                    Line[14], Line[15], BitDecode[1]      );
ResetControl        Reset1  (Clock, BitDecode[1], BitInput[1]);

// Decode logic for bit 2
ThreeInputNANDGate  Gate17  (BitInv[4], BitInv[3], Bit[2], Line[17]);
ThreeInputNANDGate  Gate18  (BitInv[4], Bit[2], BitInv[1], Line[18]);
ThreeInputNANDGate  Gate19  (BitInv[4], Bit[2], Bit[0], Line[19]);
ThreeInputNANDGate  Gate20  (BitInv[3], Bit[1], BitInv[0], Line[20]);
ThreeInputNANDGate  Gate21  (Bit[4], Bit[1], BitInv[0], Line[21]);
ThreeInputNANDGate  Gate22  (Bit[4], Bit[3], Bit[2], Line[22]);
ThreeInputNANDGate  Gate23  (Bit[2], BitInv[1], Bit[0], Line[23]);
FiveInputNANDGate   Gate24  (Bit[4], BitInv[3], BitInv[1], BitInv[0],
                    NewCellHigh, Line[24]                 );
FiveInputNANDGate   Gate25  (Bit[4], BitInv[2], Bit[1], Bit[0],
                    LatchReset, Line[25]           );
NineInputNANDGate   Gate26  (Line[17], Line[18], Line[19], Line[20],
                    Line[21], Line[22], Line[23], Line[24],
                Line[25], BitDecode[2]          );
ResetControl        Reset2  (Clock, BitDecode[2], BitInput[2]);

// Decode logic for bit 3
TwoInputNANDGate    Gate27  (BitInv[4], Bit[3], Line[27]);
ThreeInputNANDGate  Gate28  (Bit[3], Bit[2], Bit[1], Line[28]);
ThreeInputNANDGate  Gate29  (Bit[3], Bit[1], BitInv[0], Line[29]);
ThreeInputNANDGate  Gate30  (Bit[3], BitInv[1], Bit[0], Line[30]);
ThreeInputNANDGate  Gate31  (Bit[3], BitInv[2], BitInv[1], Line[31]);
FourInputNANDGate   Gate32  (BitInv[4], Bit[2], BitInv[1], BitInv[0],
                Line[32]                );
FiveInputNANDGate   Gate33  (Bit[4], BitInv[2], Bit[1], Bit[0], LatchSet,
                Line[33]                   );
SevenInputNANDGate  Gate34  (Line[27], Line[28], Line[29], Line[30],
                Line[31], Line[32], Line[33], BitDecode[3]);
ResetControl        Reset3  (Clock, BitDecode[3], BitInput[3]);

// Decode logic for bit 4
TwoInputNANDGate    Gate35  (Bit[4], Bit[3], Line[35]);
ThreeInputNANDGate  Gate36  (Bit[4], BitInv[1], Bit[0], Line[36]);
FourInputNANDGate   Gate37  (Bit[3], BitInv[2], BitInv[1], BitInv[0],
                Line[37]                );
FourInputNANDGate   Gate38  (Bit[4], BitInv[3], BitInv[0], NewCellHigh,
                Line[38]                  );
FourInputNANDGate   Gate39  (Line[35], Line[36], Line[37], Line[38],
                BitDecode[4]              );
ResetControl        Reset4  (Clock, BitDecode[4], BitInput[4]);

endmodule


// Name:   CounterWithZeroTest (this counter does not roll-over)
//
// Inputs:  Clock         - Signal on whose positive edge, the counter
//                          must change state
//          CountUp       - If high on a rising edge of "Clock", then
//                    it forces the counter to increment in the
//                    next state
//       CountDown       - If high on a rising edge of "Clock", then
//                    it forces the counter to decrement in the
//                    next state
//
// Output:  ZeroTest        - High only when the internal state of the
//                    counter is zero (zero value on all bits)

module CounterWithZeroTest (Clock, CountUp, CountDown, ZeroTest);

  input      Clock, CountUp, CountDown;
  output     ZeroTest;

  wire   [3:0] BitInUp, BitInDown, BitIn, Bit, BitInv;
  wire   [20:0] Line;

  // Internal state storage elements
  PosEdgeTrigLatch   Bit0   (Clock, BitIn[0], Bit[0], BitInv[0]);
  PosEdgeTrigLatch   Bit1   (Clock, BitIn[1], Bit[1], BitInv[1]);
  PosEdgeTrigLatch   Bit2   (Clock, BitIn[2], Bit[2], BitInv[2]);
  PosEdgeTrigLatch   Bit3   (Clock, BitIn[3], Bit[3], BitInv[3]);

  // Decode input for bit 0 when counting up
  FourInputNANDGate   Gate0  (Bit[0], Bit[1], Bit[2], Bit[3], Line[0]);
  TwoInputNANDGate    Gate1  (Bit[0], Line[0], BitInUp[0]);

  // Decode input for bit 0 when counting down
  FourInputNORGate    Gate2  (Bit[0], Bit[1], Bit[2], Bit[3], Line[2]);
  TwoInputNORGate     Gate3  (Bit[0], Line[0], BitInDown[0]);

  // Decode input for bit 1 when counting up (reuse from bit 0)
  TwoInputNANDGate    Gate4  (BitInv[0], Bit[1], Line[4]);
  TwoInputNANDGate    Gate5  (Bit[0], BitInv[1], Line[5]);
  ThreeInputNANDGate  Gate6  (Line[0], Line[4], Line[5], BitInUp[1]);
```

```
// Decode input for bit 1 when counting down (reuse from bit 0)
TwoInputNORGate     Gate7  (BitInv[0], Bit[1], Line[7]);
TwoInputNORGate     Gate8  (Bit[0], BitInv[1], Line[8]);
ThreeInputNORGate   Gate9  (Line[2], Line[7], Line[8], BitInDown[1]);

// Decode input for bit 2 when counting up (reuse from bit 0)
TwoInputNANDGate    Gate10 (BitInv[0], Bit[2], Line[10]);
TwoInputNANDGate    Gate11 (BitInv[1], Bit[2], Line[11]);
ThreeInputNANDGate  Gate12 (Bit[0], Bit[1], BitInv[2], Line[12]);
FourInputNANDGate   Gate13 (Line[0], Line[10], Line[11], Line[12],
                    BitInUp[2]                     );

// Decode input for bit 2 when counting down (reuse from bit 0)
TwoInputNORGate     Gate14 (BitInv[0], Bit[2], Line[14]);
TwoInputNORGate     Gate15 (BitInv[1], Bit[2], Line[15]);
ThreeInputNORGate   Gate16 (Bit[0], Bit[1], BitInv[2], Line[16]);
FourInputNORGate    Gate17 (Line[2], Line[14], Line[15], Line[16],
                    BitInDown[2]                   );

// Decode input for bit 3 when counting up (terminal bit)
ThreeInputNANDGate  Gate18 (Bit[0], Bit[1], Bit[2], Line[18]);
TwoInputNANDGate    Gate19 (BitInv[3], Line[18], BitInUp[3]);

// Decode input for bit 3 when counting down (terminal bit)
ThreeInputNORGate   Gate20 (Bit[0], Bit[1], Bit[2], Line[20]);
TwoInputNORGate     Gate21 (BitInv[3], Line[20], BitInDown[3]);

// Select which direction to count on the next transition
CounterGate     Cont0  (BitInUp[0], BitInDown[0], Bit[0], CountUp,
                CountDown, BitIn[0]               );
CounterGate     Cont1  (BitInUp[1], BitInDown[1], Bit[1], CountUp,
                CountDown, BitIn[1]               );
CounterGate     Cont2  (BitInUp[2], BitInDown[2], Bit[2], CountUp,
                CountDown, BitIn[2]               );
CounterGate     Cont3  (BitInUp[3], BitInDown[3], Bit[3], CountUp,
                CountDown, BitIn[3]               );
```

```
  // Test if we are currently in state zero
  FourInputNORGate    Gate28 (Bit[0], Bit[1], Bit[2], Bit[3], ZeroTest);

endmodule



// Name:   CounterWithReset (this counter rolls-over)
//
// Inputs:  Clock          - Signal on whose positive edge, the counter
//                         must count up
//         Reset           - If high on a rising edge of "Clock", then
//                         it forces the counter to a zero state
//
// Output:  Bit            - Four bit output reflecting the internal
//                         state of the counter
//         BitInv          - Four bit output reflecting the negation
//                         of the internal state of the counter

module CounterWithReset (Clock, Reset, Bit, BitInv);

  input       Clock, Reset;
  output  [3:0] Bit, BitInv;

  wire        ResetInv;
  wire    [3:0] BitInUp, BitIn;
  wire    [8:0] Line;

  // Internal state storage elements
  PosEdgeTrigLatch   Bit0   (Clock, BitIn[0], Bit[0], BitInv[0]);
  PosEdgeTrigLatch   Bit1   (Clock, BitIn[1], Bit[1], BitInv[1]);
  PosEdgeTrigLatch   Bit2   (Clock, BitIn[2], Bit[2], BitInv[2]);
  PosEdgeTrigLatch   Bit3   (Clock, BitIn[3], Bit[3], BitInv[3]);

  // Decode input for bit 0
  Inverter         Gate0  (Bit[0], BitInUp[0]);
  // Decode input for bit 1
  TwoInputNANDGate   Gate1  (BitInv[0], Bit[1], Line[1]);
  TwoInputNANDGate   Gate2  (Bit[0], BitInv[1], Line[2]);
  TwoInputNANDGate   Gate3  (Line[1], Line[2], BitInUp[1]);

  // Decode input for bit 2
  TwoInputNANDGate   Gate4  (BitInv[0], Bit[2], Line[4]);
  TwoInputNANDGate   Gate5  (BitInv[1], Bit[2], Line[5]);
  ThreeInputNANDGate Gate6  (Bit[0], Bit[1], BitInv[2], Line[6]);
  ThreeInputNANDGate Gate7  (Line[4], Line[5], Line[6], BitInUp[2]);

  // Decode input for bit 3 (terminal bit)
  ThreeInputNANDGate Gate8  (Bit[0], Bit[1], Bit[2], Line[8]);
  Inverter        Gate9  (Line[8], BitInUp[3]);

  // Select if we will count up or reset the counter
  Inverter        Gate10 (Reset, ResetInv);
  DataGate          Cont0  (BitInUp[0], ResetInv, BitIn[0]);
  DataGate          Cont1  (BitInUp[1], ResetInv, BitIn[1]);
  DataGate          Cont2  (BitInUp[2], ResetInv, BitIn[2]);
  DataGate          Cont3  (BitInUp[3], ResetInv, BitIn[3]);

endmodule


// Name:    WindowCounter
//
// Inputs:  Clock            - Signal on whose positive edge, the counter
//                          must count up
//          DataIn [3:0]     - Data to be loaded into the trigger register
//                          that controls at which counter value the
//                          counter will be reset
//          Load             - Signal on whose positive edge, new data from
//                          the "DataIn" input will be latched into the
//                          trigger register
```

```
//
// Output:  Increment        - Asserted for one clock cycle when the
//                    counter has reached the value stored in the
//                    trigger register (the counter will be reset
//                    on the following cycle and begin counting
//                    again)

module WindowCounter (Clock, DataIn, Load, Increment);

  input       Clock, Load;
  input   [3:0] DataIn;

  output      Increment;

  wire        Equal;
  wire   [3:0] RegOut, RegOutInv, CntOut, CntOutInv, Test;

  // Four bit trigger register
  PosEdgeTrigLatch  Lat0   (Load, DataIn[0], RegOut[0], RegOutInv[0]);
  PosEdgeTrigLatch  Lat1   (Load, DataIn[1], RegOut[1], RegOutInv[1]);
  PosEdgeTrigLatch  Lat2   (Load, DataIn[2], RegOut[2], RegOutInv[2]);
  PosEdgeTrigLatch  Lat3   (Load, DataIn[3], RegOut[3], RegOutInv[3]);

  // Resettable counter
  CounterWithReset  Count  (Clock, Equal, CntOut, CntOutInv);

  // Equality tester
  BitEqualTest     Test0  (RegOut[0], RegOutInv[0], CntOut[0],
                   CntOutInv[0], Test[0]          );
  BitEqualTest     Test1  (RegOut[1], RegOutInv[1], CntOut[1],
                   CntOutInv[1], Test[1]          );
  BitEqualTest     Test2  (RegOut[2], RegOutInv[2], CntOut[2],
                   CntOutInv[2], Test[2]          );
  BitEqualTest     Test3  (RegOut[3], RegOutInv[3], CntOut[3],
                   CntOutInv[3], Test[3]          );
  FourInputNANDGate Result (Test[0], Test[1], Test[2], Test[3], Equal);
  Inverter        Inv0  (Equal, Increment);

endmodule


// Name:   WindowControl
//
// Inputs:  Clock           - Signal on whose positive edge, arrivals must
//                          be marked
//        Arrival          - If high, indicates that a cell has arrived
//                          on the path assigned to this window control
//        DataIn [3:0]      - Data to be loaded into the trigger register
//                          that controls every how many clock cycles
//                          the window counter will be decremented
//        Load             - Signal on whose negative edge, new data from
//                          the "DataIn" input will be latched into the
//                          register that indicates how often to
//                          decrement the window counter.
//
// Output:  Alarm           - If high, indicates that the window counter
//                          has reached zero and, therefore, to many
//                          cells have passed through the path assigned
//                          to this counter.

module WindowControl (Clock, Arrival, DataIn, Load, Alarm);

  input      Clock, Arrival, Load;
  input  [3:0] DataIn;

  output     Alarm;

  wire       Increment, LoadInv;

  Inverter        Gate0  (Load, LoadInv);
  WindowCounter    Control (Clock, DataIn, LoadInv, Increment);
  CounterWithZeroTest Check  (Clock, Increment, Arrival, Alarm);
```

```
endmodule


// Name:    SequenceDetect
//
// Inputs:  Clock          - Signal on whose rising edge the state
//                         machine must make a state change.
//          NewCellLow     - When high, indicates a new cell is coming
//                         in with the first byte starting on the low
//                         order bits of the input.
//          NewCellHigh    - When high, indicates a new cell is coming
//                         in with the first byte starting on the high
//                         order bits of the input.
//          LookupRes      - Result of the memory lookup to see if new
//                         cell is valid.
//          LatchSet       - Output of the SR-Latch which, if high,
//                         indicates there is new path data to be loaded
//                         into the memory lookup module.
//          LatchReset     - The negated state of the LatchSet input.
//
// Output:  PVRL           - Latch the results of the read from the
//                         memory lookup module.
//          RSRL           - Clear the new path information in the new
//                         path registers (by setting the SR-Latch
//                         indicating the validity of the data as
//                         being false)
//          LLODG26        - Start the low-byte counter at 26
//          LLODG27        - Start the low-byte counter at 27
//          LHODG27        - Start the high-byte counter at 27
//          VVRL           - Turn on the data gate to make the demux
//                         assert one of its outputs, to trigger one of
//                         the window control modules
//          RAS            - Row address select line on the memory
//                         lookup module
//          CAS            - Column address select line on the memory
//                      lookup module
//      W               - Read/Write control line on the memory
//                      lookup module
//      LowChoke           - Control line to the data gate that informs
//                      it whether to transmit the low byte of the
//                      data words exiting from the shift register
//      HighChoke          - Control line to the data gate that informs
//                      it whether to transmit the high byte of the
//                      data words exiting from the shift register
//      FourBDS     [5:0] - Control lines to the four bit multiplexer
//                      that shunt different portions of the
//                      incoming data words from the Receiver
//      FourBDL     [5:0] - Latch control lines on the latches that store
//                      the path information of the currently
//                      transiting cell
//      TwelveBDS    [1:0] - Control lines to the twelve bit by four line
//                      multiplexer that presents data from
//                      various latch groups to the memory lookup
//                      module

module SequenceDetect (Clock, NewCellLow, NewCellHigh, LatchSet, LatchReset,
            LookupRes, FourBDS, FourBDL, TwelveBDS, PVRL, RSRL,
            VVRL, RAS, CAS, W, LowChoke, HighChoke            );

  input      Clock, NewCellLow, NewCellHigh, LatchSet, LatchReset,
            LookupRes;
  output     PVRL, RSRL, VVRL, RAS, CAS, W;
  output     LowChoke, HighChoke;
  output  [1:0] TwelveBDS;
  output  [5:0] FourBDS, FourBDL;

  wire       LLODG26, LLODG27, LHODG27, Ground;
  wire    [4:0] Bit, BitInv, LowByte, HighByte;

  assign     Ground = 0;
  StateMachine  Core   (Clock, NewCellLow, NewCellHigh, LatchSet,
                        LatchReset, Bit, BitInv                    );
  StateControl  Signal (Bit, BitInv, NewCellLow, NewCellHigh, LookupRes,
                        FourBDS, FourBDL, TwelveBDS, PVRL, RSRL,
                        LLODG26, LLODG27, LHODG27, VVRL, RAS, CAS, W);

  DownCounterWithPreset LowByteCounter
                       (Clock, LLODG26, LLODG27, LowByte[0], LowByte[1],
                        LowByte[2], LowByte[3], LowByte[4]          );
  DownCounterWithPreset HighByteCounter
                       (Clock, Ground, LHODG27, HighByte[0], HighByte[1],
                        HighByte[2], HighByte[3], HighByte[4]        );
  FiveInputNANDGate     LowByteChoke
                       (LowByte[0], LowByte[1], LowByte[2], LowByte[3],
                        LowByte[4], LowChoke                   );
  FiveInputNANDGate     HighByteChoke
                       (HighByte[0], HighByte[1], HighByte[2], HighByte[3],
                        HighByte[4], HighChoke                   );

endmodule


// Let's bring the whole thing together

module NetworkSecurity;

  wire        Clock, NewCellLow, NewCellHigh, LatchSet, LatchReset,
            SDDG, PVRL, RSRL, VVRL, RAS, CAS, W, LowChoke, HighChoke,
            LoadNewData, UnLoadNewData, LookupRes, LookupResInv;
  wire   [1:0] TwelveBDS;
  wire   [2:0] PathState, PathStInv, MuxTrigIn, MuxLoadIn, MemRes, PathData;
  wire   [3:0] WindowData;
  wire   [4:0] StateBit, StateBitInv;
  wire   [5:0] FourBDS, FourBDL;
  wire   [6:0] WindowTrig, LoadWindow, Alarm, NewPathDataIn, NewPathDataOut,
            LoadTrig;
```

```
wire  [11:0] RAMAddress;
wire  [15:0] DataIn, ShiftOut, GateOut;
wire  [23:0] NewPathAddressIn, NewPathAddressOut, LatchIn, LatchOut,
          Address;

assign
  PathData[0]  = NewPathDataOut[0],  PathData[1]  = NewPathDataOut[1],
  PathData[2]  = NewPathDataOut[2],
  WindowData[0] = NewPathDataOut[3],  WindowData[1] = NewPathDataOut[4],
  WindowData[2] = NewPathDataOut[5],  WindowData[3] = NewPathDataOut[6];

initial
  begin
    // generate our report
    // $shm_open;
    // $shm_probe("AC");
    // #5000  $shm_close;
    #5000 $finish;
    // $monitor ($time,,
    //        "S0=%b  S1=%b  #1=%d  #2=%d  #3=%d  #4=%d  O=%d",
    //        Sel0, Sel1, One, Two, Three, Four, Out);
  end

ClockGen          Timer   (Clock);

NetworkReceiver    Receive  (Clock, NewCellLow, NewCellHigh, DataIn);
NetworkTransmitter Transmit (Clock, NewCellLow, NewCellHigh, GateOut);
ControlModule      PathGen  (NewPathAddressIn, NewPathDataIn, LoadNewData,
                    LatchSet, LatchReset                    );
ShiftRegister      Shifter  (Clock, DataIn, ShiftOut);
EightBitDataGate   LowGate  (ShiftOut[7:0], LowChoke, GateOut[7:0]);
EightBitDataGate   HighGate (ShiftOut[15:8], HighChoke, GateOut[15:8]);

SequenceDetect     Control  (Clock, NewCellLow, NewCellHigh, LatchSet,
                    LatchReset, LookupRes, FourBDS, FourBDL,
                    TwelveBDS, PVRL, RSRL, VVRL, RAS, CAS, W,
                  LowChoke, HighChoke);

NewPathStore      NewPath  (LoadNewData, UnLoadNewData, NewPathAddressIn,
                  NewPathDataIn, LatchSet, LatchReset,
                  NewPathAddressOut, NewPathDataOut          );

FourBitTwoLineSelector  S1  (DataIn[11:8],  DataIn[3:0],   FourBDS[0],
                  LatchIn[23:20]                );
FourBitTwoLineSelector  S2  (DataIn[7:4],   DataIn[15:12], FourBDS[1],
                  LatchIn[19:16]                );
FourBitTwoLineSelector  S3  (DataIn[3:0],   DataIn[11:8],  FourBDS[2],
                  LatchIn[15:12]                );
FourBitTwoLineSelector  S4  (DataIn[15:12], DataIn[7:4],   FourBDS[3],
                  LatchIn[11:8]                 );
FourBitTwoLineSelector  S5  (DataIn[11:8],  DataIn[3:0],   FourBDS[4],
                  LatchIn[7:4]                  );
FourBitTwoLineSelector  S6  (DataIn[7:4],   DataIn[15:12], FourBDS[5],
                  LatchIn[3:0]                  );
FourBitRegister      L1  (FourBDL[0], LatchIn[23:20], LatchOut[23:20]);
FourBitRegister      L2  (FourBDL[1], LatchIn[19:16], LatchOut[19:16]);
FourBitRegister      L3  (FourBDL[2], LatchIn[15:12], LatchOut[15:12]);
FourBitRegister      L4  (FourBDL[3], LatchIn[11:8],  LatchOut[11:8] );
FourBitRegister      L5  (FourBDL[4], LatchIn[7:4],   LatchOut[7:4]  );
FourBitRegister      L6  (FourBDL[5], LatchIn[3:0],   LatchOut[3:0]  );

TwelveBitFourLineSelector SM(LatchOut[23:12],
                  LatchOut[11:00],
                  NewPathAddressOut[23:12],
                  NewPathAddressOut[11:0],
                  TwelveBDS, RAMAddress   );

DynamicRAM        Lookup0(RAMAddress, RAS, CAS, W, PathData[0],
                  MemRes[0]                 );
DynamicRAM        Lookup1(RAMAddress, RAS, CAS, W, PathData[1],
                  MemRes[1]                 );
DynamicRAM        Lookup2(RAMAddress, RAS, CAS, W, PathData[2],
                MemRes[2]                );

// Path and volume version specific hardware
PosEdgeTrigLatch    Latch0 (VVRL, MemRes[0], PathState[0], PathStInv[0]);
PosEdgeTrigLatch    Latch1 (VVRL, MemRes[1], PathState[1], PathStInv[1]);
PosEdgeTrigLatch    Latch2 (VVRL, MemRes[2], PathState[2], PathStInv[2]);

ThreeBitDataGate    Gate0  (PathState, VVRL, MuxTrigIn);
ThreeBySevenDemux   Mux0   (MuxTrigIn, WindowTrig);
ThreeBySevenDemux   Mux1   (PathData, LoadTrig);
Inverter        Gate1  (W, WInv);
TwoInputNANDGate    Gate2  (LoadTrig[0], WInv, LoadWindow[0]);
TwoInputNANDGate    Gate3  (LoadTrig[1], WInv, LoadWindow[1]);
TwoInputNANDGate    Gate4  (LoadTrig[2], WInv, LoadWindow[2]);
TwoInputNANDGate    Gate5  (LoadTrig[3], WInv, LoadWindow[3]);
TwoInputNANDGate    Gate6  (LoadTrig[4], WInv, LoadWindow[4]);
TwoInputNANDGate    Gate7  (LoadTrig[5], WInv, LoadWindow[5]);
TwoInputNANDGate    Gate8  (LoadTrig[6], WInv, LoadWindow[6]);
WindowControl       Cont0  (Clock, WindowTrig[0], WindowData,
                LoadWindow[0], Alarm[0]        );
WindowControl       Cont1  (Clock, WindowTrig[1], WindowData,
                LoadWindow[1], Alarm[1]        );
WindowControl       Cont2  (Clock, WindowTrig[2], WindowData,
                LoadWindow[2], Alarm[2]        );
WindowControl       Cont3  (Clock, WindowTrig[3], WindowData,
                LoadWindow[3], Alarm[3]        );
WindowControl       Cont4  (Clock, WindowTrig[4], WindowData,
                LoadWindow[4], Alarm[4]        );
WindowControl       Cont5  (Clock, WindowTrig[5], WindowData,
                LoadWindow[5], Alarm[5]        );
WindowControl       Cont6  (Clock, WindowTrig[6], WindowData,
                LoadWindow[6], Alarm[6]        );
SevenInputNORGate   Gate9  (Alarm[0], Alarm[1], Alarm[2], Alarm[3],
                Alarm[4], Alarm[5], Alarm[6], LookupResInv);
Inverter        Gate10 (LookupResInv, LookupRes);
```

VITA

Name:	Dan Cristian Teodor
Date of Birth:	December 9, 1970
Place of Birth:	Bucharest, Romania
Parents:	Liviu and Gabriela Teodor
Education:	Master of Science, 1997 Texas A&M University College Station, TX Major: Computer Science
	Bachelor of Science, 1993 State University of New York at Buffalo Buffalo, NY Major: Electrical Engineering
Professional Experience:	Systems Engineer, Electrospace Systems Inc. Dallas, TX
	Design Engineer, Eshed Robotec Princeton, NJ
Permanent Address:	66-48 Thornton Place Rego Park, NY 11374