Abstract

NIMMELAPELLI, RAJA. FPGA Implementation of a SIP Message Processor. (Under the direction of Dr. Yannis Viniotis.)

Session Initiation Protocol (SIP) is fast emerging as the next generation signaling protocol. It operates independently of the underlying network transport protocol, establishing sessions between multiple users irrespective if the media is voice, data or video. It is projected to eventually replace the existing multiple voice and video signaling protocols as a single protocol which achieves all. SIP implements a non-trivial grammar. Parsing this grammar to extract the protocol fields proves to be a high overhead for the CPU. This paper presents hardware offload architecture; the SIP Offload Engine (SOE) which essentially extracts the SIP fields and stores them is a proprietary data structure, for easy access by the CPU. An analysis has been done which shows a reduction in the CPU overhead by as much as 88%.

FPGA Implementation of a SIP Message Processor

by

Raja Nimmelapelli

A thesis submitted to the Graduate Faculty of
North Carolina State University
in partial fulfillment of the
requirements for the Degree of
Master of Science

Computer Engineering

Raleigh, North Carolina

2006

Approved By:

Dr. Gregory Byrd	Dr. Adolfo Rodriguez
Dr. Yannis Viniotis Chair of Advisory Committee	Dr. William Rhett Davis

Biography

Raja Nimmelapelli was born in Chennai, India in June 1977. He graduated from Pune Institute of Computer Technology, India, with a Bachelors degree in Electronics and Telecommunications Engineering in June 2000. After undergraduate studies, he worked with Paxonet Communications, Pune, for four years. In August 2004, he joined North Carolina State University as a graduate student in the Computer Engineering program. In Summer and Fall of 2005, he interned at ADC Telecommunications, Raleigh in the ASIC/FPGA Group. While working towards the Masters degree, he worked on his thesis under the guidance of Dr. Yannis Viniotis.

Acknowledgement

I sincerely thank my advisor, Dr.Yannis Viniotis for his invaluable guidance and support through my graduate studies. I would like to sincerely thank him for providing me with the opportunity and encouraging me in pursuing my interests in the area of protocol hardware offload. I thank him for his constant support and encouragement.

I am very grateful to Dr. W. Rhett Davis and Dr. Gregory Byrd for agreeing to be on my thesis committee and for the valuable feedback regarding the thesis document.

I would like to thank Dr. Adolfo Rodriguez and Dr. Curtis Hrischuk, who spared time from their busy schedule at IBM, Raleigh to add value at the thesis defense.

I'd like to mention my family, who were always encouraging and supportive at all times. I am indebted to my sister and brother-in-law for providing me with this opportunity to pursue graduate studies.

I would also like to mention Megha and all my friends, without whose support I would never have been able to succeed in my endeavors.

Contents

List of Figures

List of Tables

Chapter 1

Introduction and Literature Review

This chapter serves as an introduction to the main concepts behind the thesis: the Session Initiation Protocol (SIP) and hardware offloading.

1.1 Importance of SIP

Amongst a host of other compelling reasons, SIP gains importance for two reasons discussed in following paragraphs.

Sophisticated functions like multimedia and videoconferencing have enjoyed global appeal. Owing to the potentially large number of users, a robust distributed call management protocol needed to be developed, which would take care of user location, call setup, capability negotiation and call termination. To allow for easier scaling of the core network, the intelligence had to be kept away from the core and embedded in the end-points. SIP is a signaling protocol which provides this. It also allows run-time modification of the session parameters.

The other reason for the importance is the rich feature-set it provides for the user. Although implemented over the IP, it provides familiar Public Switched Telephone Network (PSTN) like operations: dialing a number, causing a phone to ring, hearing the ringback tone or a busy tone. It allows a user to leave a voice message, activate his

answering machine via phone or Internet. It enables personal mobility and provides the user with a greater degree of freedom. For example, a user could register from multiple locations and have the phone at all these locations ringing at the same time. The user could pick up the phone at whichever location he is and start the conversation, without having to go through the trouble of updating his location whenever he moves.

1.2 Current Implementations of SIP

This section and its subsections elaborate on the different components that make up a SIP system.

1.2.1 Architecture of a SIP system

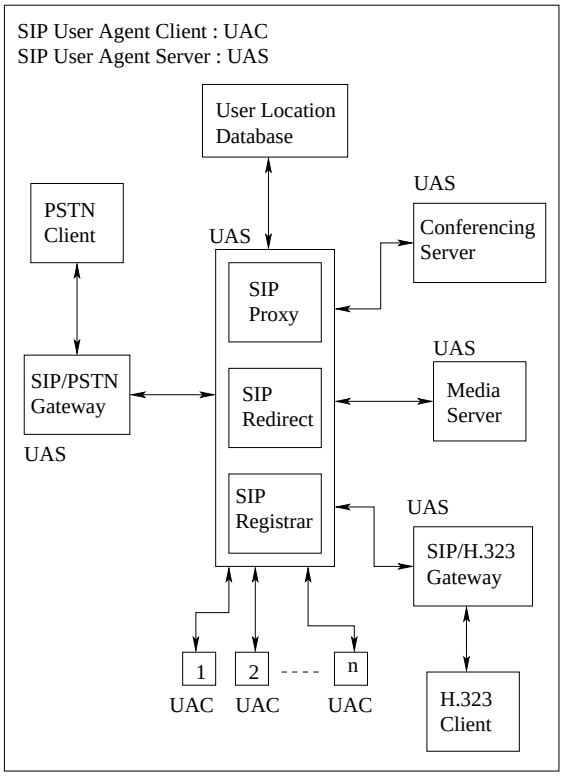

Figure 1.1: Functional Architecture Block Diagram

Figure 1.1 depicts the architecture of a typical SIP system [1]. Components of a SIP system are called User Agents (UA's). A UA is either a SIP User Agent Client (UAC) or a SIP User Agent Server. A SIP UAC is an entity which is a receiver of servi ces provided by the SIP system. A SIP UAS is an entity which in some way enables these services. The SIP UAC and UAS are discussed in more detail in sections 1.2.2 and 1.2.3. The User Location Database (ULD) stores the location information of the UAC's. This database is queried when a particular UAC is to be located. There are non-SIP users (PSTN, H.323 clients) which enter the SIP system via gateways.

The components of a SIP system converse using dialogs. The standards track for SIP, RFC 3261 [2], defines a dialog as "a peer-to-peer SIP relationship between two UAs that persists for some time". It represents a context that facilitates the

sequencing of messages between the UA's and proper routing of requests between both of them. A dialog goes through three phases: the creation of the dialog, the processing of requests within the dialog and the termination of the dialog.

With respect to a dialog, communication between UA's can occur:

1. *Outside* a dialog. This communication occurs when two UA's are in the process of setting up a dialog.

2. *Inside* a dialog. This communication occurs when two UA's have established a dialog.

1.2.2 SIP User Agent Client (UAC)

A UAC represents an end system which generates requests. These requests are usually the result of human actions like clicking the mouse button or other similar external stimulus. The requests are processed by the UAS and a reply is sent back to the UAC.

The request generation described above starts with the UAC initiating the establishment of a dialog using the INVITE method. Upon receipt of a success code from the intended recipient, the dialog is said to be established. The session parameters can then be negotiated. Data transfer starts once the communicating UAC's decide on the multimedia to be used in the session. To tear down the dialog, the BYE method is used by any one of the UAC's.

Existing implementations of UAC's can be classified as *hardphones* or *softphones*. Hardphones are UAC's implemented in hardware as IP appliances. They resemble the traditional PSTN telephone handset. A commercial example of this is the Cisco 7900 series unified IP phones [3]. Softphones are UAC's that are implemented in software. They typically run on a PC and use the PC hardware for audio/video input/output. A commercial implementation of a softphone is the Microsoft Messenger 4.7 [4].

1.2.3 SIP User Agent Server (UAS)

A UAS represents an end system which processes requests sent by the UAC. Based on the type of request received, the UAS executes procedures to generate a reply. This reply is then routed back to the UAC.

The UAS receives the requests from the UAC to establish a dialog. For this out-of-dialog request, there is a pre-defined set of procedures to be executed before the UAS can process this request and send a reply to establish a dialog. The UAS, in this order, needs to:

1. Authenticate the request.
2. Inspect the method in the request to see if supported by the UAS.
3. Inspect the header in the request, ignoring any malformed fields.
4. Inspect the content of the request, potentially rejecting the request for various reasons.
5. If all above inspections have passed, run method-specific procedures to generate a reply.

Once the success-reply generated by the UAS reaches the UAC, the dialog is said to be established. The UAS then processes any within-dialog requests like capability negotiations. Upon receving the BYE request, the dialog is torn down.

There are many forms a UAS can take in a SIP system, depending on the task it is supposed to perform. The various implementations are listed in section 1.2.4. Note that this is not a comprehensive list of all UAS implementations, only the major ones.

1.2.4 Examples of a UAS

In this section, for each implementation of a UAS, the inputs received and the outputs generated are discussed with respect to their usage in the SIP architecture discussed in Section 1.2.1.

1. Registrar

   - Functional Diagram

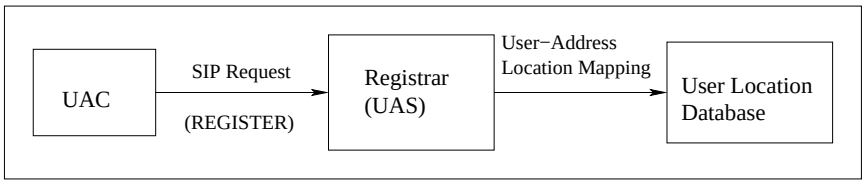

Figure 1.2: Functional location of a Registrar

   - Inputs received

     The Registrar accepts REGISTER inputs from UAC's.

   - Outputs generated

     The Registrar processes the REGISTER request and updates the User Location Database with the binding information for that UAC.

2. Proxy

   - Functional Diagram

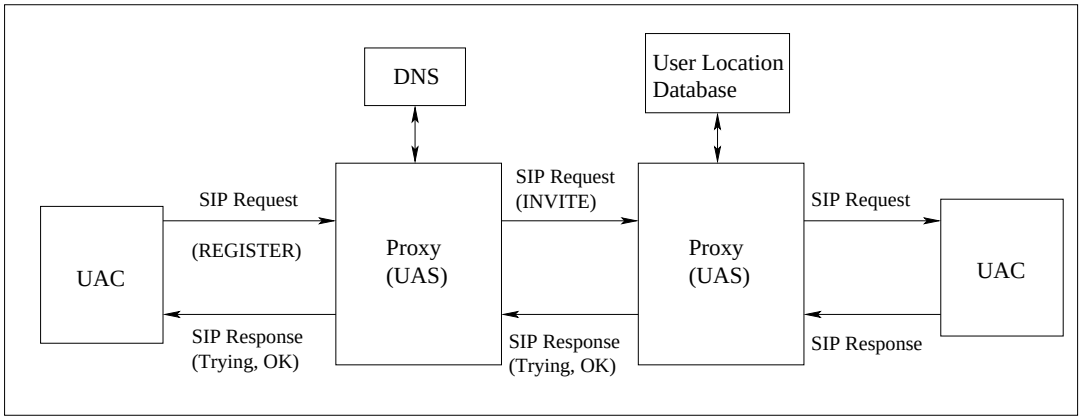

Figure 1.3: Functional location of a Proxy

- Inputs received

  A Proxy receives SIP requests like INVITE which are to be routed to the destination UAC. It could also receive SIP responses from the destination which are to be relayed to the originating UAC.

- Outputs generated

  The Proxy receives a SIP request (INVITE) at its input from the source UAC. It replicates this SIP request at the output and sends it to the target domain Proxy. It also receives SIP response messages(Trying, OK) from the destination UAC and replicates these at its output to the source UAC.

3. Gateways

   Gateways exist to serve a strong business purpose. They act as a bridge between the new SIP components and existing legacy PSTN and H.323 equipment. With the help of gateways, end-to-end service can be provided seamlessly over these different protocols.

   - PSTN gateway [5]
     - Functional Diagram

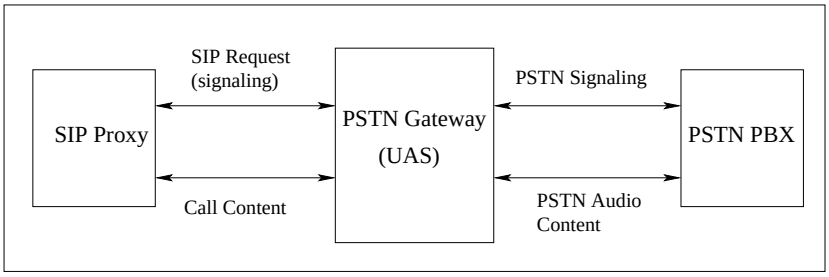

Figure 1.4: Functional location of a PSTN gateway

     - Inputs received

       A PSTN gateway receives both signaling and content information. It receives PSTN signaling/data from the PSTN PBX which needs to be translated into the SIP domain. Conversely, it also receives SIP signaling/content which needs to be translated into the PSTN domain.

  - Outputs generated

    The gateway outputs SIP signaling/content when it receives PSTN signaling/content at its input. It outputs PSTN signaling/content when it receives SIP signaling/content at its input.

- H.323 gateway [6]

  - Functional Diagram

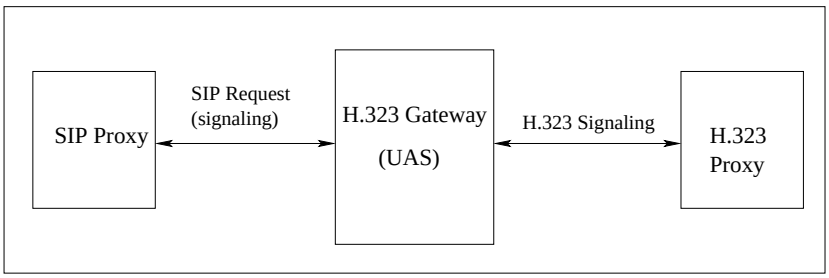

Figure 1.5: Functional location of an H.323 gateway

  - Inputs received

    An H.323 gateway receives both signaling and content information. It receives H.323 signaling/data from the H.323 proxy which needs to be translated into the SIP domain. Conversely, it also receives SIP signaling/content which needs to be translated into the H.323 domain.

  - Outputs generated

    The gateway outputs SIP signaling/content when it receives H.323 signaling/content at its input. It outputs H.323 signaling/content when it receives SIP signaling/content at its input.

4. Redirect server

   - Functional Diagram

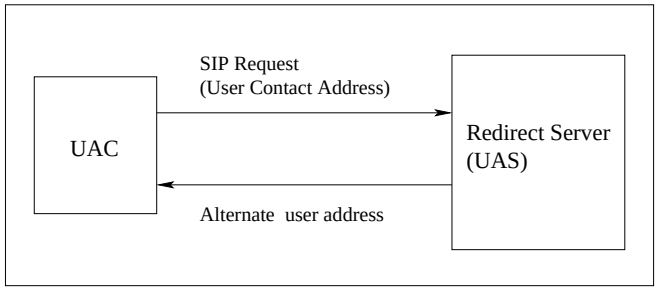

Figure 1.6: Functional location of a Redirect Server

- Inputs received

  The Redirect server receives a SIP request at its input with a user contact address.

- Outputs generated

  The Redirect server outputs a list of alternate addresses that user could potentially be available at.

5. Media server

   - Functional Diagram

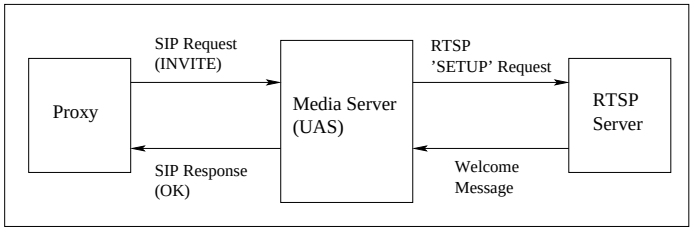

Figure 1.7: Functional location of a Media Server

   - Inputs received

     The Media server receives a SIP INVITE request at its input.

   - Outputs generated

The Media server outputs a request to the Real Time Streaming Protocol (RTSP) server to obtain the welcome message and recording of voicemail. It sends an OK response to the originating UAC. The UAC then directly exchanges packets with the RTSP server.

6. Conferencing server [7]

   - Functional Diagram

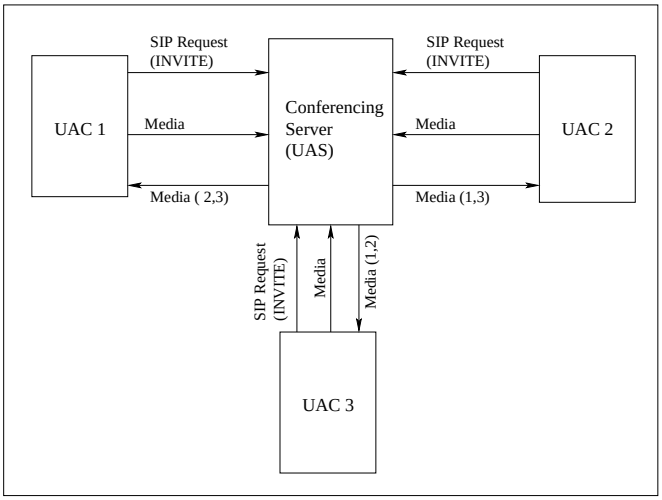

Figure 1.8: Functional location of a Conferencing Server

   - Inputs received

     The Conferencing server receives SIP INVITE requests from multiple UAC's wishing to hold a conference. It also receives the conference media content from all UAC's part of that conference.

   - Outputs generated

     The Conference server outputs OK messages to the various UAC's requesting to be a part of the conference. It also mixes the content input by indivudual UAC's and sends them to the other UAC's, as shown in the Figure 1.8

1.3 Maximizing performance via hardware offload

1.3.1 Issues with software network processing

Networking software on a general workstation has to cater to a wide range of client applications. This results in simple and general purpose interfaces presented by the software to the hardware. The speed of the software interfaces is falling behind the speed with which the network hardware operates. It is common to now see Gigabit network cards deployed in general workstations. Even though the CPU clock speed is scaling, the high rate at which these cards present data to the networking software limits the communication performance and increases latency. Thus, the network software is a severe bottleneck for faster packet processing.

A solution which eases this bottleneck is to transfer some of the tasks done by the networking software onto dedicated hardware, an approach called hardware offloading. With this approach, there is a definite increase in performance as the hardware would process data much faster than software. More the tasks that are offloaded, faster the data processing becomes. Potentially, it is thus possible to match the network hardware speed and achieve maximum system performance.

Further sections exemplify hardware offloading by citing some industry proven techniques.

1.3.2 Industry examples: TCP Offload Engine (TOE) and SAN/iSCSI

This section elaborates on how hardware offloading has resulted in performance enhancements. The two examples cited are TCP/IP and iSCSI offload.

1. TOE

   Over the past few years, the general purpose processor industry has seen tremendous growth in the processor speeds. Starting from 60MHz at the time of introduction, the CPU clock speed has scaled to the current 3GHz [8]. This would make one think that now TCP/IP processing would also be much faster. This

however is not completely true. Although the TCP/IP processing speed has increased, the rate of increase is not parallel to the rate at which the CPU clock speed has scaled. The limiting factor is now the memory and I/O subsystems. TCP/IP is inherently a memory intensive operation, so the bad effects of memory latency are more pronounced. This latency results in wasted CPU idle cycles. So, a faster CPU only results in more idle cycles.

TCP/IP provides certain features which make it so popular for communication over the Internet. Some of these are reliable delivery, flow control and congestion control. However, to implement these features, a significant amount of software is required in the form of timers, counters, units that run algorithms, perform arithmetic and maintain the connection state. TCP/IP processing involves a good deal of memory. For example, the TCP connection state information is close to 256 bytes, so a TCP implementation which could potentially handle 32,000 connections needs 8 MB memory just to store the state information. So, it is clear that TCP/IP in software incurs a large software overhead and involves significant amount of memory. Both factors limit the performance of software TCP/IP.

There have been development efforts which target to increase TCP/IP performance via software modifications. For example, in native TCP/IP, the loss of a single packet results in the retransmission of many packets [9]. An improvement was to retransmit only the packet which was not received, called the Selective ACK. This reduces the amount of data transmitted in case of a retransmission. TCP Reno is another software enhancement, which creates two zones of data transmission, called Slow Start and Congestion Avoidance. The primary aim of this scheme is to avoid congestion in the network, thus reducing the probability of retransmissions without much sacrifice in the transfer rate. Reno also includes a Fast retransmit and Fast recovery scheme, aimed at early detection and remedy for lost packets. This is an attempt to enhance performance by reducing latency. However, the performance benefits from such schemes are behind the improvement achieved by hardware offloading.

Since the memory access time is a bottleneck for TCP/IP performance, multithreading could be employed for better performance. Here, another thread is executed while the first one is waiting for data from the memory. Multithreading has two drawbacks. First, it is assumed that multiple threads are ready for execution at any point of time, which may not be the case for multiple high-bandwidth connections transferring large amounts of data. Another drawback is that multithreading requires more cache memory, which does not make it a cost-effective enhancement [7].

The advantages of TCP hardware offloading are highlighted by Terminator [10], a TCP offload engine developed by Chelsio Communications, Inc. It was found that during performance evaluation, for packet sizes of 1KB on a uni-directional 10 Gbps link, the throughput achieved with offload was four times that without offload.

2. iSCSI

   The original SCSI protocol was used to communicate between devices connected to the internal and external computer busses. Internet SCSI [iSCSI] was developed to enable large file transfers via blocks between any two computers connected to the Internet. iSCSI is used to enable Storage Area Networking (SAN).

   iSCSI runs over TCP/IP and due to the overhead TCP incurs, is much slower than its SAN competitor, the Fibre Channel interface. However, the advantage iSCSI offers is that, unlike Fibre Channel, it uses TCP/IP/Ethernet, which makes it cheaper to install and maintain. So, in order to boost the performance and make it acceptable for SAN, the iSCSI and TCP protocols are offloaded onto hardware.

1.3.3 SIP Offload Engine (SOE)

The importance of SIP has been elaborated on in Section 1.1. In anticipation of its growth, it would be a good idea to make advancements which would allow for faster

processing of the protocol. We have seen how hardware offloading boosts processing speed in TCP and iSCSI. It is proposed to apply the same idea and offload the SIP protocol onto hardware.

How much of a protocol to offload is a complexity-performance tradeoff. If we offload more of the protocol, the complexity increases but so does the performance. To contain the complexity and still achieve significant performance boost, it is proposed to offload only the SIP message parsing onto hardware. Chapter 3 shows how such a limited offload can reduce CPU utilization by as much as 90%.

1.4 Thesis organization

The organization of the rest of the thesis is described. Chapter 2 and its subsections introduce SIP and its message structure, format and header fields. Chapter 3 presents an analysis which describes the potential reduction in CPU utilization via hardware offload. Chapter 4 presents the architecture of the hardware which would be doing the offloaded processing. Chapter 5 presents a detailed description of all the modules involved in the design. Chapter 7 discusses a sample implementation of the design on a Xilinx FPGA and talks about the design statistics in terms of gates occupied, memory used and other factors. Chapter 8 completes the thesis by identifying areas which have scope for further improvements.

Chapter 2

SIP

This chapter briefly describes the structure of the SIP protocol and explains its components.

2.1 Introduction

SIP follows a Request/Response type of transaction model. It follows a layered structure, with the syntax and encoding being the lowermost layer. It follows the Augmented Backnus Naur Format (ABNF) for encoding [11]. A SIP client would send a Request message to the server, which would reply with a Response message. The request/response message formats follow the standard track for Internet Message Format [12]. It follows the UTF-8 charset [13], which is the same as ASCII for 00-7F. The exact message types and formats are discussed in further sections.

2.2 SIP Grammar and its complexity

Grammar can be defined as a set of rules by which the protocol can be specified. It can be thought of as a tool to enable a protocol and ensure a uniform interpretation of a protocol by all its users.

Parsing can be thought of as the action of examining the lines of a protocol and extracting different tokens from it. Parsing too follows the same set of rules that

make up the grammar.

Sections 2.2.1 and 2.2.2 discuss the reasons why SIP grammar could be considered as significantly complex to parse.

2.2.1 Lexical analysis

In a lexical perspective, the following points contribute to the complexity of SIP grammar [12].

1. Line length limit

   The standard places a limit of 998 characters per line. This places a requirement on the software parser to be able to buffer so many characters at a stretch. This buffer comes at a cost for the software. In the SOE, the need for a buffer is eliminated by directly writing these bytes in memory, i.e no intermediate storage is required.

2. Random order of Header fields

   Header fields are lines composed of a field name, followed by a semicolon (":"), followed by a field body, and terminated by CRLF. The parser needs to be able to track the semicolon and CRLF delimiters. The characters till the semicolon need to be buffered seperately, as they form the header field name. The characters after the semicolon until the CRLF form the header value and are to be buffered seperately. The SOE performs the tracking of these delimiters by using a state machine based approach. This simplifies the seperation of the characters between delimiters. The header fields could occur in a random fashion, which make it difficult for the CPU to extract any particular header it is looking for.

3. Folding of header fields

   Although the limit for a line is 998 characters, a stream of characters could still be contained in multiple lines by "folding" the lines. This is a method in which a contiguous stream of characters is split into multiple lines by inserting CRLF sequence where we wish to start a newline. By starting this newline with

a space or tab, we indicate that this is a continuation of the previous stream of characters. The software parser needs to be able to detect folded lines and "unfold" them correctly to extract the original character stream. The way the SOE tackles this issue is by detecting the CRLF-SP/HT sequence and deleting these occurences. The next stream of characters is seamlessly concatenated with the previous stream and treated as one stream. Note that no buffering is required in the SOE to cater to folding lines, which is where the advantage comes over a software implementation.

2.2.2 Syntactical Analysis

In a syntactical perspective, the following points contribute to the complexity of SIP grammar [12].

1. Special Characters

   Quoted characters and special characters like '/' which are usually used as a delimiter can be used as a part of the message by enclosing it like "/".

2. Complex comments specification

   A comment is a string of characters enclosed in parantheses. The comments could be nested within multiple lines via folding. There could be special characters present in the comment.

3. Multiple other specifications

   There exist separate specifications for the Date and Time which only add to the complexity.

4. Multiple field definitions

   The headers are classified into multiple categories based on their function. Examples of these are Origination fields, Destination fields, ID fields, Informational fields and Trace fields.

2.3 SIP Messages

The SIP protocol enables end users to communicate with each other via *messages*. These are pieces of information which are processed by the client/server. In its basic form, a message could either be a request sent from a client to a server or a reply from the server to the client.

2.3.1 SIP message format

A message consists of a start-line, one or more header fields, an empty line indicating the end of the header fields, and an optional message-body [2].This is shown in Figure 2.1.

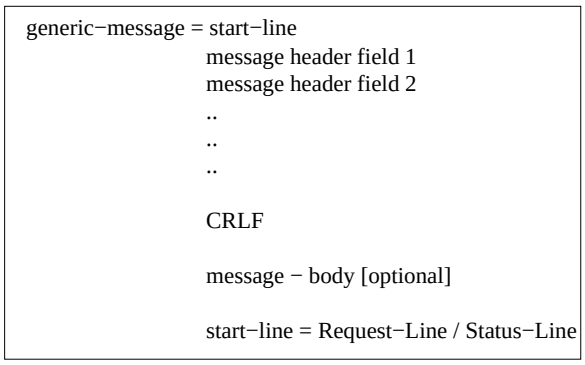

Figure 2.1: Structure of a SIP message

2.3.2 Types of messages

A SIP message could either be a *Request* or a *Response*.

1. SIP Request Message

   A request is recognized by the presence of a request-Line as the start-line. The request-line is of the format:

   Request-Line = Method SP Request-URI SP SIP-Version CRLF

The method is an action associated with a session between end users. The methods and their function are listed below:

- REGISTER : Used to register contact information with the server.
- INVITE, ACK, CANCEL : Used for setting up sessions.
- BYE : Used to teminate a session.

The Request-URI is the recipient of the SIP message. The SIP_Version is currently SIP/2.0 and is to be included in all messages. The CRLF terminates the Request-Line.

2. SIP Response Message

A response is recognized by the presence of a status-line as the start-line. The status-line is of the format:

Status-Line = SIP-Version SP Status-code SP Reason-Phrase CRLF

The Status-Code represents the result of the action taken due to the request. The result of a request is categorized below:

(a) 100-199 : Request received, process in progress.

(b) 200-299 : The request was received, understood and accepted.

(c) 300-399 : Further action needs to be taken to complete processing of the request.

(d) 400-499 : The request cannot be processed at the server. Could be due to bad syntax.

(e) 500-599 : The server failed to process the request. The request could have been valid.

(f) 600-699 : Global failure. This request cannot be processed by any server.

The Reason-Phrase is an english-like equivalent of the Status-Code. For ex. for Status-Code 200, the Reason-Phrase is "OK".

Both the Request/Response messages may have multiple message headers. These headers are discussed further in section 2.4.

2.4 SIP Header Fields

SIP Header fields form a part of the SIP message. Each header conveys some information for the destination.

2.4.1 Header field format

The format for a SIP message header is shown. Each header field consists of a name followed by a colon (":") and the field value.

field-name : [field-value]

Note that the field value could extend over multiple lines as explained in section 2.2.1.

2.4.2 Types of Header Fields

The types of header fields can thought of to be based on the function performed by that header. Although this list may not be complete, the major header types are listed.

1. Originator fields : From, Destination, To
2. Routing fields: Via
3. Authentication: Proxy-Authenticate.

2.5 Requirements for parsing SIP messages

A SIP message essentially consists of the SIP Start-Line and multiple message headers. Based on which part of the SIP message is being parsed, the following requirements apply.

- Requirements for SIP Request-Line / Status-Line

  1. The Request-Line should follow the format “Request line = Method name SP Request-URI SP SIP-Version CRLF”.
  2. The status-Line should follow the format “Status Line = SIP-version SP Status-code SP Reason-phrase CRLF”.
  3. No CR or LF allowed except at the end of the line.
  4. No folding lines in any of the elements of the Request-Line
  5. Request-URI must not contain unescaped spaces or control chars.
  6. Request URI could also be a telephone URI, i.e a nummeric telephone number.
  7. SIP-version must be SIP/2.0 in uppercase

- Requirements for the Header fields

  Following is a list of requirements for parsing the header fields to correctly extract the header field names and their values.

  1. The header is to be parsed in a <field_name : field_value> fashion.
  2. Any number of whitespace allowed on either side of the semicolon in the above point.
  3. Header fields can be extended over multiple lines, as long as each new line starts with a SP or HT.
  4. Support multiple header field rows with the same field name, with the all field values specified in a comma seperated list.
  5. Within a field value, any number of <parameter name=parameter value> pairs can exist within a header field value.
  6. Must allow for abbreviated forms of header names.

Chapter 3

Processor overhead savings analysis

This chapter starts by briefly discussing the drawbacks of SIP message parsing in software. It then introduces the data structure used in the hardware implementation of the message parsing. An analysis is presented which shows how this proprietary data structure format allows us to achieve significant processor cycle savings. The intention of the analysis is to breakdown in a quantifiable terms, the difference between software and hardware processing of a SIP message. A part of the analysis consists of an imitation of a software implementation of a SIP message parsing module. The amount of software processing accounted for is considered as bare minimum for a single-CPU, without considering enhanced CPU features like multi-processor environments. The processing required is represented in terms of CPU cycles required. A similar analysis is done for the hardware approach to arrive at a number for the CPU cycles required. A comparison of these numbers gives us the CPU cycle savings between the two approaches.

3.1 Software approach to SIP message parsing

This section elaborates on why the SIP message parsing is considered to be CPU intensive[1]. We've seen the structure of the SIP message in in Chapter 2. The

software code parsing the SIP message needs to extract all the header field names and their respective values. This could be a cumbersome task owing to the following two reasons:

1. Delimiters

   The standard allows unrestricted use of delimiters such as spaces and tabs before, in-between and after the SIP header field names and values. Header field values could span multiple lines via the delimiter LWS, as explained in section 3. The software needs to recognize, use and eliminate these delimiters to make correct sense of the byte-stream.

2. Unrestricted order of field placement

   There are certain SIP header fields related to routing (ex. TO,VIA) the CPU needs to examine before it starts to process the rest of the SIP message. Although the standard recommends the placement of the routing header fields at the beginning of the SIP message, it does not mandate it. So the CPU should allow for random placement of the header fields in the SIP message. This forces the CPU to parse the entire message before it even knows if it should process the message. In the SOE approach, the hardware would directly pass the routing headers to the CPU.

Section 3.2 elaborates on the format of the data structure used in the SOE. The format of this data structure is what allows the CPU savings, discussed in section 3.3.

3.2 Data structure for hardware offload

The data structure used to store the SIP header fields and values is shown in Figure 3.1. This data structure is implemented in a 31-bit memory. A motivation behind choosing a 32-bit data width is the fact that the TCP/IP header fields are aligned in 32-bit words. Different information is stored at pre-decided memory locations. The

CPU is aware of these addresses and thus knows where to read header information from. An explanation of each of the elements of the data structure follows.

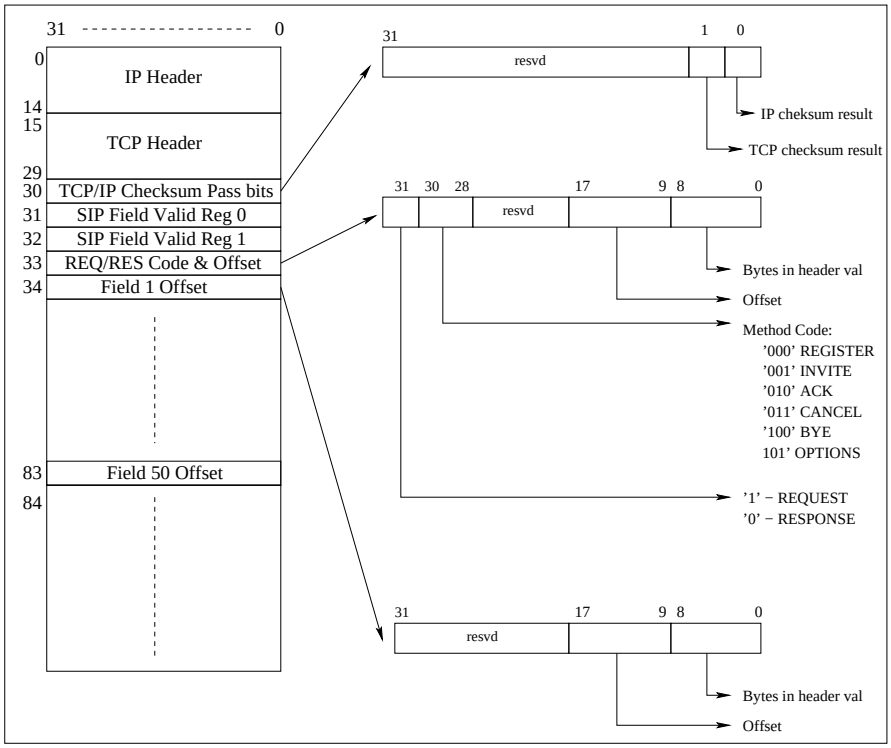

Figure 3.1: SIP Data Strcuture

1. IP Header

   All the IP header bytes received are stored as in in locations 0 through 14.

2. TCP Header

   All the TCP header bytes received are stored as in in locations 15 through 29.

3. TCP/IP checksum pass bits

   The least-significant 2 bits of address 30 are used to store the result of the TCP and IP checksum calculations. Bit 0 stores the IP checksum result while bit 1 stores the TCP checksum calculation result. If the bit is set, it indicates

a successful checksum match, i.e the calculated and the received checksums match. If this bit is cleared, it indicates that the checksums mismatch.

4. SIP Field Valid Registers 0,1

   These are two 32-bit registers which tell the CPU which SIP headers were found in the SIP packet. Each header is alloted one bit to indicate its presence in the SIP packet. If the bit is set, that header was found in the SIP packet. Table 3.1 tells us the bit-header bindings each SIP header.

Table 3.1: *Presence bit allocation for SIP Methods/Headers*

Hex Addr	**Bit #**	**SIP Keyword**	**Hex Addr**	**Bit #**	**SIP Keyword**
31	0	Accept	31	1	Accept-Encoding
31	2	Accept-Language	31	3	Ack
31	4	Alert-Info	31	5	Allow
31	6	Authentication-Info	31	7	Authorization
31	8	Bye	31	9	Cancel
31	10	Call-ID	31	11	Call-Info
31	12	Contact	31	13	Content-Disposition
31	14	Content-Encoding	31	15	Content-Language
31	16	Content-Length	31	17	Content-Type
31	18	CSeq	31	19	Date
31	20	Error-Info	31	21	Expires
31	22	From	31	23	In-Reply-To
31	24	Invite	31	25	Max-Forwards
31	26	Min-Expires	31	27	MIME-Version
31	28	Options	31	29	Organization
31	30	Priority	31	31	Proxy-Authenticate
32	0	Proxy-Authorization	32	1	Proxy-Require
32	2	Register	32	3	Record-Route
32	4	Reply-To	32	5	Require
32	6	Retry-After	32	7	Route
32	8	Server	32	9	Subject
32	10	Supported	32	11	Timestamp
32	12	To	32	13	Unsupported
32	14	User-Agent	32	15	Via
32	16	Warning	32	17	WWW-Authenticate

5. Request/Response Type and Code

   Address 33 in the SIP Data Structure holds information regarding the client-to-server request sent or the server-to-client response. Table 3.2 summarizes the various bits used and their meaning.

Table 3.2: *Request/Response bits*

0xAddress	**Bits**	**Meaning**
33	31	1- SIP Request, 0 - SIP Response
33	30-28	SIP Method. For bits 2-0, values are "000" - REGISTER, "001" - INVITE, "010" - ACK, "011" - CANCEL, "100" - BYE and "101" - OPTIONS
33	17-9	The offset in the data structure where the bytes start.
33	8-0	The number of bytes the REQ/RESP occupies data structure starting from the above offset.

6. Method/Header Offset calculation

   The bytes received in the Request/Response/Header are stored in the SIP Data Structure. These bytes can be retreived by using the offset provided. The offset tells us the start address and number of bytes stored for that method/header. Using this information and a global start address of 0x86, we can directly access the value of any method/header.

3.3 Processor savings analysis

The analysis starts by considering the following sample SIP packet.

INVITE sip:bob@biloxi.com SIP/2.0

Via: SIP/2.0/UDP pc33.atlanta.com

Max-Forwards: 70

To: Bob <sip:bob@biloxi.com>

From: Alice <sip:alice@atlanta.com>;tag=1928301774

Call-ID: a84b4c76e66710@pc33.atlanta.com

CSeq: 314159 INVITE

The analysis then compares the CPU utilization involved in calculation of TCP/IP checksums and extraction of the SIP message headers for the following two methods:

- Complete software approach

  The software does the TCP/IP checksum calculations and SIP header extraction.

Table 3.3: *Cycles required for an all-software approach*

Function	**Pseudo Code**	**Operator**	**# Cycles**	**Scale**
IP Header Checksum Calculation	L1:READ Reg1, 4 bytes	READ	5	
	XOR Reg2, Reg1	XOR	5	
	Loop L1	INCR	5	
TCP Header Checksum Calculation	L2:READ Reg1, 4 bytes	READ	92.5	
	XOR Reg2, Reg1	XOR	92.5	
	Loop L2	INCR	92.5	
Extract SIP Method	READ Reg1, 4 bytes	READ	2	
	COMPARE Reg1,"INVI"	COMPARE	48	
	READ reg2, 4 bytes			
	COMPARE Reg1, "TEx̂"			
Extract SIP Header field name	L3: READ Reg1, 4 bytes	READ	24	
	COMPARE Reg1,":"	COMPARE	96	
	JumpNotEqual L3			
Align SIP header field	WRITE Reg1, write_address	WRITE	24	
		INCR	24	
Extract SIP Header field	L4: READ Reg1, 4 bytes	READ	56	
	COMPARE Reg1, CRLF	COMPARE	192	
	JumpNotEqual L4			
Align SIP header field	WRITE Reg1, write_address	WRITE	56	
		INCR	56	
Read aligned data	READ Reg1, 4 bytes	READ	80	
		INCR	80	
	Total	READS	259.5	1
		XOR	97.5	1
		COMPARE	336	1
		INCR	257.5	1
		WRITE	80	1
	TOTAL CYCLES	**1030.5**		

- Hardware offload approach

  The TCP/IP checksum and SIP header extraction functions are offloaded onto dedicated hardware. The software does trivial address calculations to obtain the header field values.

Table 3.4: *Cycles required for a hardware-offloaded approach*

Function	**Pseudo Code**	**Operator**	**# Cycles**	**Scale**
IP Header Checksum Calculation	READ Reg1, 4 bytes	READ	1	
TCP Header Checksum Calculation	READ Reg1, 4 bytes	READ	1	
Read SIP Method	READ Reg1, 4 bytes	READ	1	
Read SIP Header field name	READ Reg1, 4 bytes	READ	2	
	READ Reg2, 4 bytes	COMPARE	8	
	COMPARE, Reg1, "1"	INCR	2	
	COMPARE Reg2, "1",			
Read SIP Header field value	ADD Base_address, index	READ	54	
	READ Reg1, 4 bytes	INCR	54	
	ADD Field_address, Reg1	ADD	2	
	L1: READ Reg1, 4 bytes	COMPARE	6	
	COMPARE Field_length,0			
	JumpNotEqual L1			
	Total	READS	59	1
		COMPARE	14	1
		INCR	56	1
		ADD	2	2
	TOTAL CYCLES	**133**		

From the Tables 3.3 and 3.4, it can be seen that the cycles required by the hardware-offload method (133 cycles) is significantly lower than those required when using an all-software approach (1030.5 cycles). This translates into a savings of over 88%.

Chapter 4

Design Architecture

4.1 Introduction

In chapter 3 we saw the potential savings the SOE could achieve. In this section we take a high level look at the design architecture. The block diagram will be discussed. Functions and implementation details of individual blocks will also be elaborated on.

The design examines the incoming SIP/TCP/IP stream. The individual SIP, TCP header and IP header bytes are delineated and sent to dedicated processing blocks. Once the data is processed, each block writes its share of data to the SIP data structure. The order of write access to the SIP data structure is the IP block first, followed by the TCP block and then finally the SIP block. Once the SIP data structure is completed, it is assumed that a PCI device would read this data and write it to the system memory, where it would be examined by the CPU.

The packets at the input of the design could be expected to arrive in a non-stop fashion. To cater to this, a pipelined approach was taken at a block level so that the flow of packets is not interrupted. Further, no backpressure is exerted on the input FIFO. The architecture can be classified as cut-through, i.e the incoming packet is not stored before processing.

The following sections list all the blocks involved in the design. Their functions and a brief idea of their implementation are given.

4.2 Block diagram and Explanation

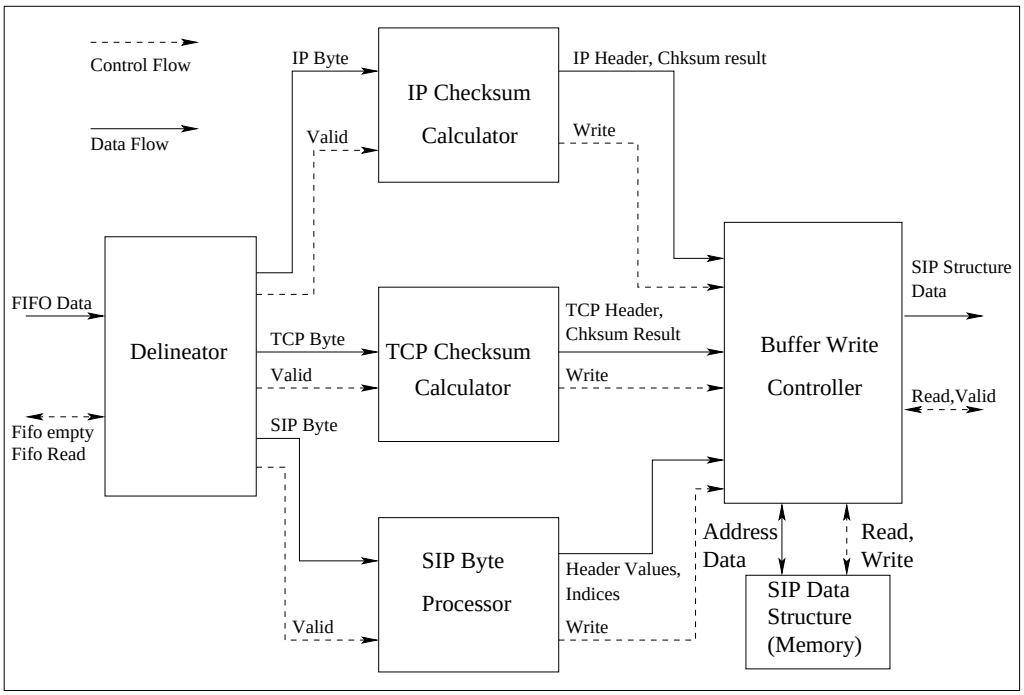

Figure 4.1: Block Diagram

The above figure shows the block level implementation of the SIP Offload Engine. We shall discuss the functions and brief implementation in subsequent sections.

4.2.1 Delineator

The following sections list the functions and implementation of the Delineator block.

- Functions

  1. Interfaces to a FIFO on the input side.
  2. Examines incoming SIP/TCP/IP byte stream.
  3. Classifies the incoming bytes as IP bytes, TCP bytes and SIP bytes

4. Routes IP, TCP, SIP bytes to the IP, TCP and SIP Processor blocks
5. Identifies the incoming TCP and IP checksum values, passes them to the respective blocks for checksum verification

- Implementation

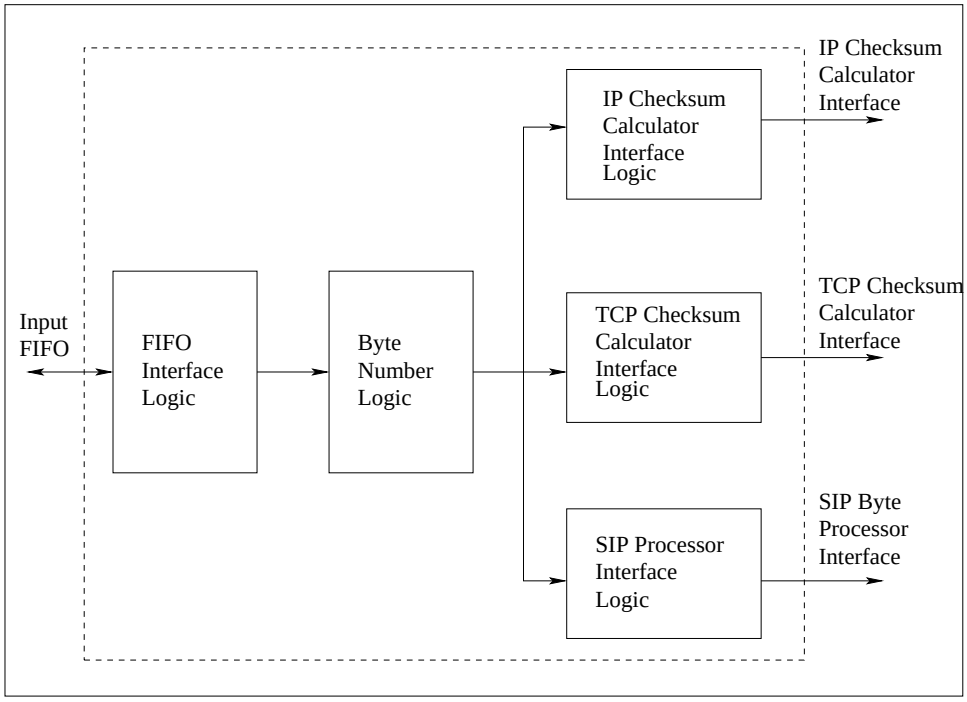

Figure 4.2: Delineator Implementation Diagram

The delineator reads data from the input FIFO and examines the incoming byte stream. First the IP bytes will be seen, then the TCP bytes followed by the SIP bytes. Note that the SIP bytes are also flagged as TCP bytes for TCP checksum calculation.

1. Identifying IP bytes

   An indication is received from the FIFO to flag the start of a new packet. This packet is the SIP/TCP/IP stream that needs to be processed.

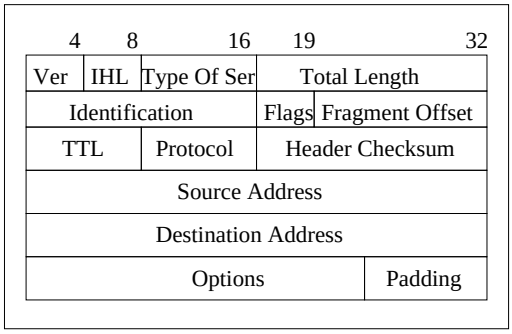

Figure 4.3: Format of the IP Header

Figure 4.3 shows the fields present in the IP Header. The first byte in this new stream marks the first IP header byte (Version & Internet Header Length). The Internet Header Length tells us how many successive bytes are going to be IP bytes. For example, if this value is 0101, it signifies that the next (5*4=20) bytes are IP header bytes. Accordingly, the next 20 bytes are flagged as IP Bytes. These bytes are processed by the IP Checksum calculator.

2. Identifying TCP Bytes

   For the purpose of TCP checksum calculation, all bytes following the IP bytes till the end of the packet are TCP bytes. The end of packet occurs when the total length (IP header) number of bytes are counted. Accordingly these bytes are flagged by the delineator, when corresponding with the TCP Checksum calculator. Note that the pseudo TCP header bytes, shown in 4.4 are also flagged by the Delineator, as these are to be included in the TCP checksum calculation. An exception to this is the field “TCP Length”. This field is the sum of the TCP header octets and the data octets. The 12 octets of the pseudo header are not included in this sum.

This value is calculated by the TCP Checksum Calculator.

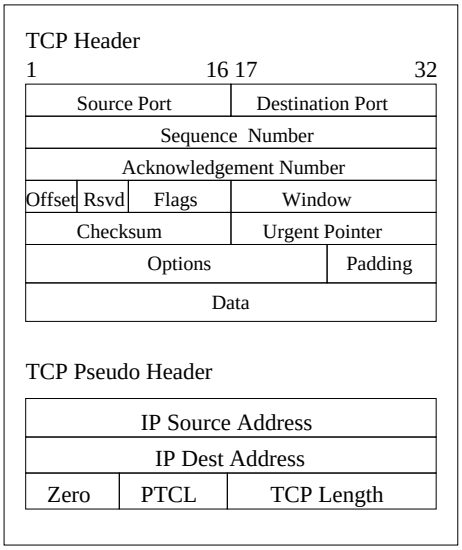

Figure 4.4: Format of the TCP Header

3. Identifying SIP Bytes

   The SIP bytes form the TCP payload portion. The TCP payload starts a byte after the TCP header ends. The end of the TCP Header is indicated by the 4-bit Data Offset field in the TCP header. For example, if the Data Offset field = 1000, then after (8*4 = 32) bytes, the SIP bytes start. The total number of SIP bytes is determined by the 16-bit Total Length field in the IP header. The formula is: Number of SIP bytes = Bytes indicated by Total Length (IP Header bytes + TCP Header bytes) All SIP bytes are flagged by the Delineator.

4.2.2 IP Checksum Calculator

The following sections list the functions and implementation of the IP Checksum Calculator Block.

- Functions

  1. Calculate the checksum over the IP header bytes.
  2. Compare the calculated checksum with the received checksum.
  3. Deliver the IP header bytes and the checksum Pass/Fail result bit to the Buffer Write Controller

- Implementation

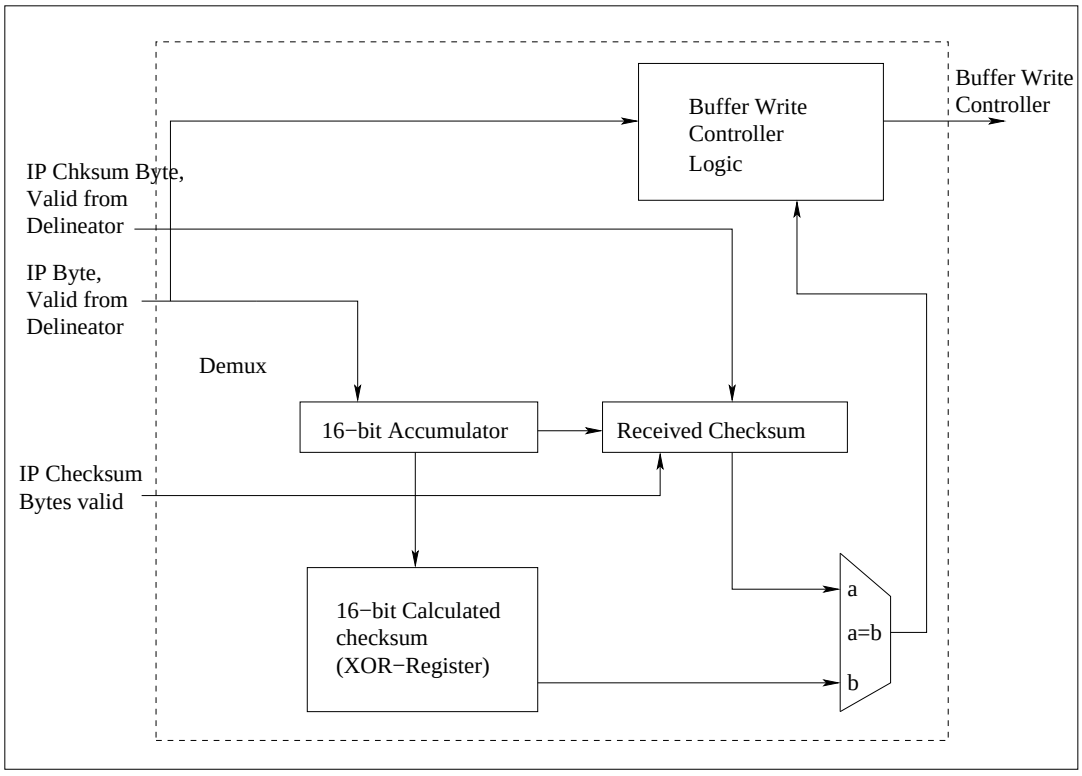

Figure 4.5: IP Checksum Calculator Implementation Diagram

The IP checksum calculator maintains a 16 bit XOR register. Every time 16 bits of IP header data are accumulated, they are XORed with the existing 16 bits of the XOR register. Once the last byte of the IP header comes in, the value in the XOR register is compared with the received checksum. Accordingly, a checksum Pass/Fail indication is determined. All received IP header bytes and the checksum result are written into the SIP data structure by the Buffer Write Controller.

4.2.3 TCP Checksum Calculator

The following sections list the functions and implementation of the TCP Checksum Calculator Block.

- Functions

  1. Calculate the TCP checksum.
  2. Compare the calculated checksum with the received checksum.
  3. Deliver the TCP header bytes and the checksum Pass/Fail result bit to the Buffer Write Controller. Note that only the TCP header bytes are written to the SIP data structure. The TCP payload bytes are used for the checksum calculations.

- Implementation

  The TCP checksum calculator maintains a 16 bit XOR register which functions in a manner similar to the IP checksum calculator. The received TCP bytes and the checksum results are written into the SIP data structure by the Buffer Write Controller.

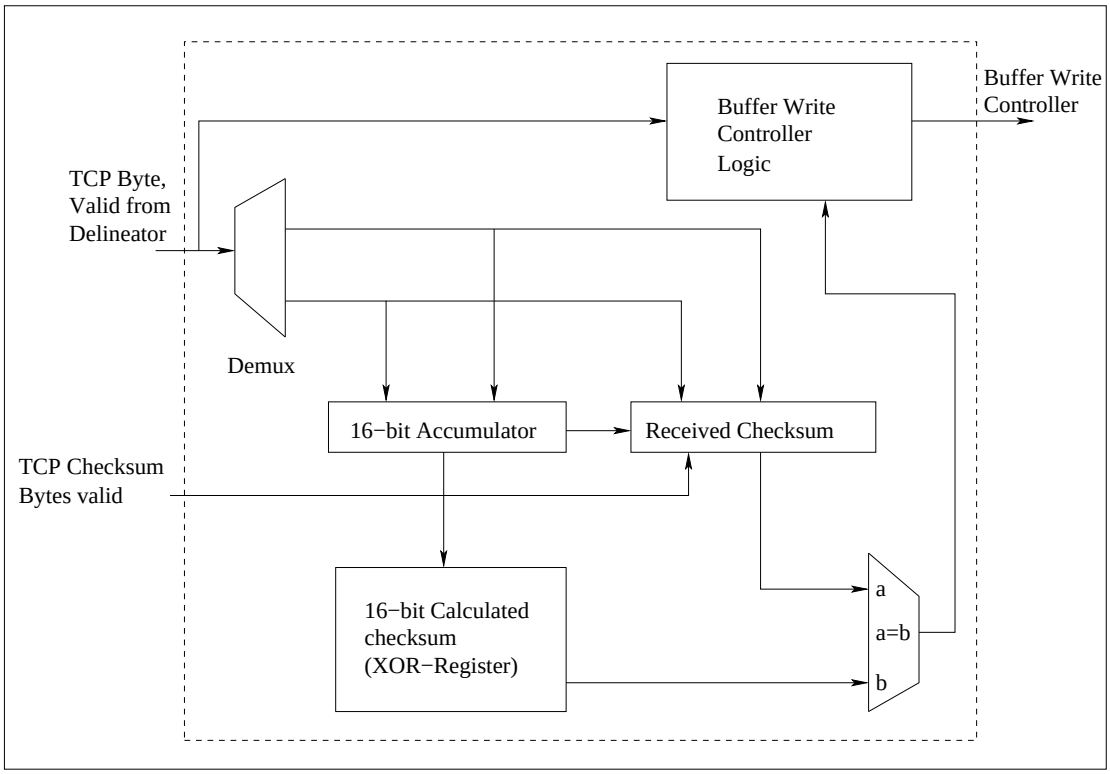

Figure 4.6: TCP Checksum Calculator Implementation Diagram

4.2.4 SIP Byte Processor

The following sections list the functions and implementation of the IP Checksum Calculator Block.

- Importance of the SIP Byte Processor The SIP Byte Processor is primarily responsible for recognizing valid SIP keywords from the incoming SIP byte stream. Its importance in the entire design stems from the fact that it generates most of the fields required in the SIP data structure. A significant percentage of the CPU overhead savings is effected by this module.

- Functions

  1. Creates tokens from the incoming SIP byte stream.

2. Search for the validity of these tokens as SIP keywords.
3. Indicate the presence of validated keyword in the SIP data structure.

- Implementation

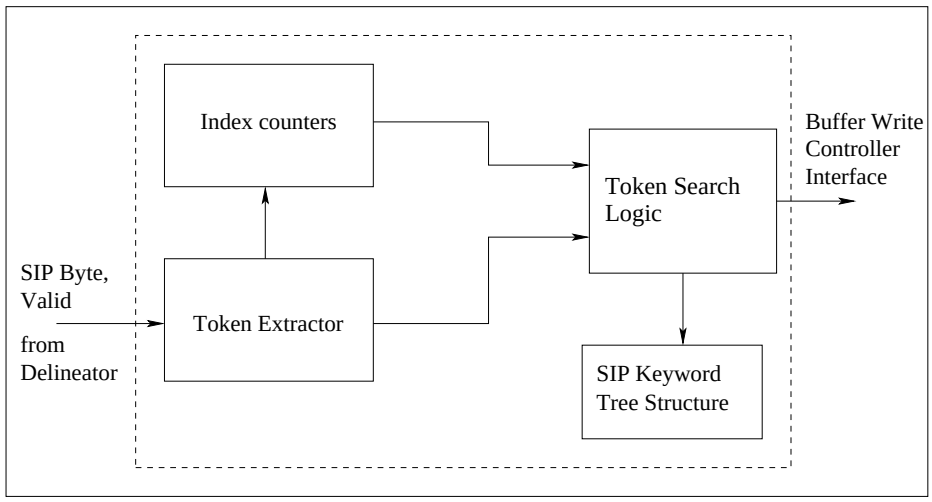

Figure 4.7: SIP Byte Processor Implementation Diagram

The SIP Byte Processor is implemented in a pipeline based approach. Details of this pipeline are discussed in section 4.3. A Finite State Machine is implemented to extract tokens by separating the whitespaces and tabs from the incoming byte stream. The tokens are sent to the pattern matching engine to check for its validity as a SIP keyword. The pattern matching engine is based on the Aho-Corasick algorithm for exact string matching. A keyword could either be a message or a header field name. In both cases, the value follows the keyword in a byte stream. The value of the keyword is isolated from the byte stream by ignoring all the spaces. This isolated value is sent to the Buffer Write Controller which would store it in the data structure. A counter is run while the isolated value is being stored. Once the entire value is stored, the value of the counter now serves as an index to where in the data structure this keyword value lies. The CPU software can then easily use this index to access the value.

4.2.5 Buffer Write Controller

The following sections list the functions and implementation of the IP Checksum Calculator Block.

- Functions

  1. Possesses exclusive write control over the SIP data structure.
  2. Accepts for data from the potential write requestors. Writes this data to the SIP data structure.

- Implementation

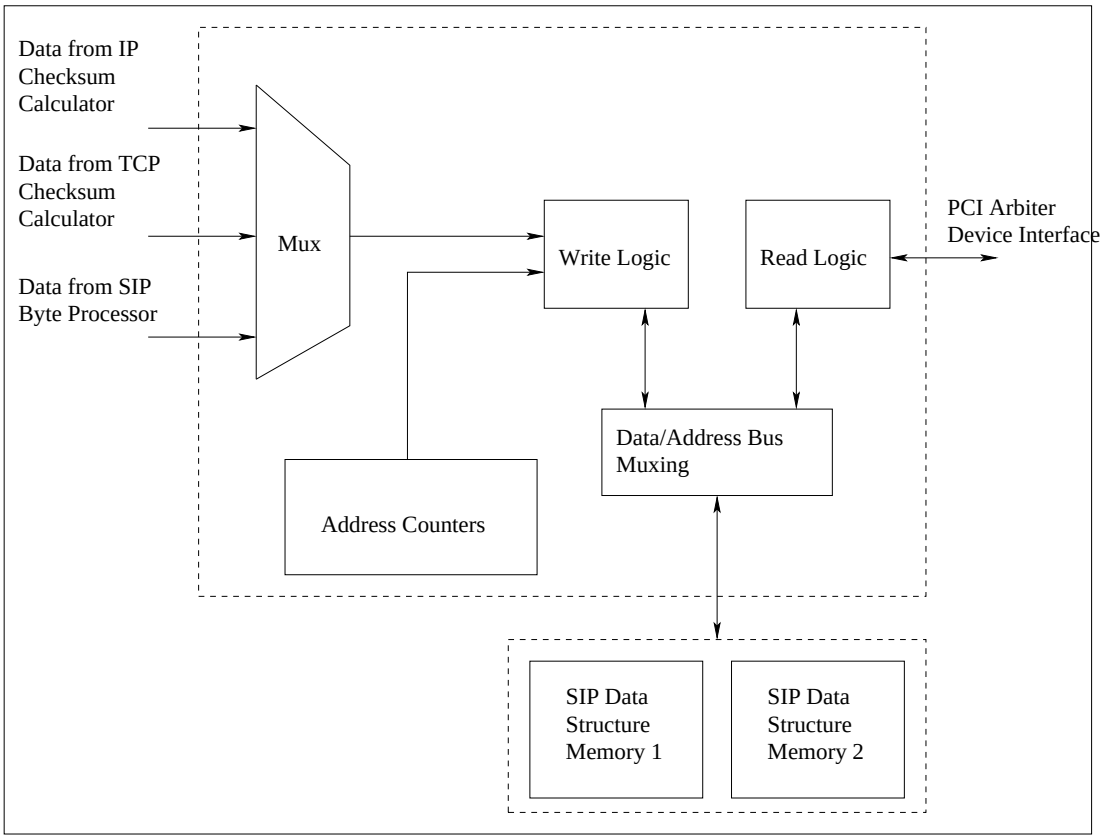

Figure 4.8: Buffer Write Controller Implementation Diagram

A first come first serve approach is used to accept write requestor data to the memory. This approach is possible because each of the write requestors issue their requests at different points of time. First the IP checksum calculator would post its request, followed by the TCP checksum calculator and finally the SIP byte processor. The data to be written is multiplexed based on who the requestor is.

4.3 Design Pipeline Explained

The input can be considered as a continous flow of packets. One way to approach the design would be to first completely buffer the incoming packet, process it, then look at the next packet. In this case, there could be a situation where we would need to exert backpressure on the input and halt the packet flow. This would happen when a large packet is being processed while multiple small packets are coming in. It would not be a good idea to exert any backpressure on the input, as this would have repercussions all the way upstream to the Network Interface Card.

To avoid the scenario mentioned in the above paragraph, we take a pipelined approach. In this way, there is no need to exert any backpressure and we acheive better throughput.

Chapter 5

Design Module Description

In section 4, we had a top-level look at a block-level architecture of the design. We saw the functions of the blocks and had a brief idea of how each block is implemented. The current section aims to provide more details. For each block, the pin interfaces with other bslocks and a detailed implementation description are given. This would include the FSM, data structures, arithmetic units and specific logic involved.

5.1 Module Delineator (*delin*)

This section and its subsection present a detailed description of the Delineator module. Section 5.1.1 presents the pin interfaces with blocks it interacts with. Section 5.1.2 describes its implementation in detail.

5.1.1 Pin Interface

This section describes the pin interfaces with the other blocks.

Table 5.1: *Interface with the system*

No.	Pin Name	Dirn.	Width	Description
1	reset	IN	1	The async system reset.
2	clk	IN	1	The system clock.

Table 5.2: *Interface with Packet FIFO*

No.	Pin Name	Dirn.	Width	Description
1	fifo_empty	IN	1	Indicates the presence of data in the FIFO
2	fifo_data	IN	9	Data read from the FIFO. Interpreted as 8 bit data byte and a 1 bit indication.
3	delin_fifo_read	OUT	1	An indication that the delineator has accepted the FIFO byte.

Table 5.3: *Interface with IP Checksum Calculator*

No.	Pin Name	Dirn.	Width	Description
1	delin_startofpkt	OUT	1	An indication to reset the checksum registers.
2	delin_ipbyte_valid	OUT	1	An indication that the current byte is valid. The IP checksum calculator processes the byte only if this signal is set.
3	delin_ip_byte	OUT	8	The byte to be processed.
4	delin_ipchksm_msbyte	OUT	1	An indication that the current byte is the IP checksum MS byte. The IP checksum calculator needs these bytes to verify the checksum
4	delin_ipchksm_lsbyte	OUT	1	An indication that the current byte is the IP checksum LS byte.
6	delin_last_ipbyte	OUT	1	An indication that the current byte is the last valid IP byte. When asserted, the IP checksum calculator matches the received and calculated checksums.

Table 5.4: *Interface with TCP Checksum Calculator*

No.	Pin Name	Dirn.	Width	Description
1	delin_startofpkt	OUT	1	An indication to reset the checksum registers.
2	delin_tcpbyte_valid	OUT	1	An indication that the current byte is valid. The TCP checksum calculator processes the byte only if this signal is set.
3	delin_tcp_hdrbyte	OUT	1	An indication that the current byte is a TCP Header byte. Only these are sent to the Buffer Write Controller to be written in the SIP Data Structure.
4	delin_tcp_byte	OUT	8	The byte to be processed.
5	delin_protocol_byte	OUT	1	An indication that the current byte is the IP protocol byte. This indication is used in the TCP checksumming process, as a zero pad is required before the protocol field (Fig. 4.4).
6	delin_tcpchksm_msbyte	OUT	1	An indication that the current byte is the TCP checksum MS byte. The TCP checksum calculator needs these bytes to verify the checksum.
7	delin_tcpchksm_lsbyte	OUT	1	An indication that the current byte is the TCP checksum LS byte.
8	delin_last_tcpbyte	OUT	1	An indication that the current byte is the last valid TCP byte. When asserted, the TCP checksum calculator performs the last XOR operation with the calculated TCP length. It then matches the received and calculated checksums.

Table 5.5: *Interface with SIP Byte Processor*

No.	Pin Name	Dirn.	Width	Description
1	delin_sipbyte_valid	OUT	1	An indication that the current byte is valid. The SIP Byte processor processes the byte only if this signal is set.
2	delin_sipbyte	OUT	8	The byte to be processed.

5.1.2 Architecture

We will start this section by discussing the interface timing between the Delineator and the input FIFO. We will then proceed to elaborate on the hardware implementation of the blocks in the logic schematic shown in 4.2.

- Input FIFO Interface Timing

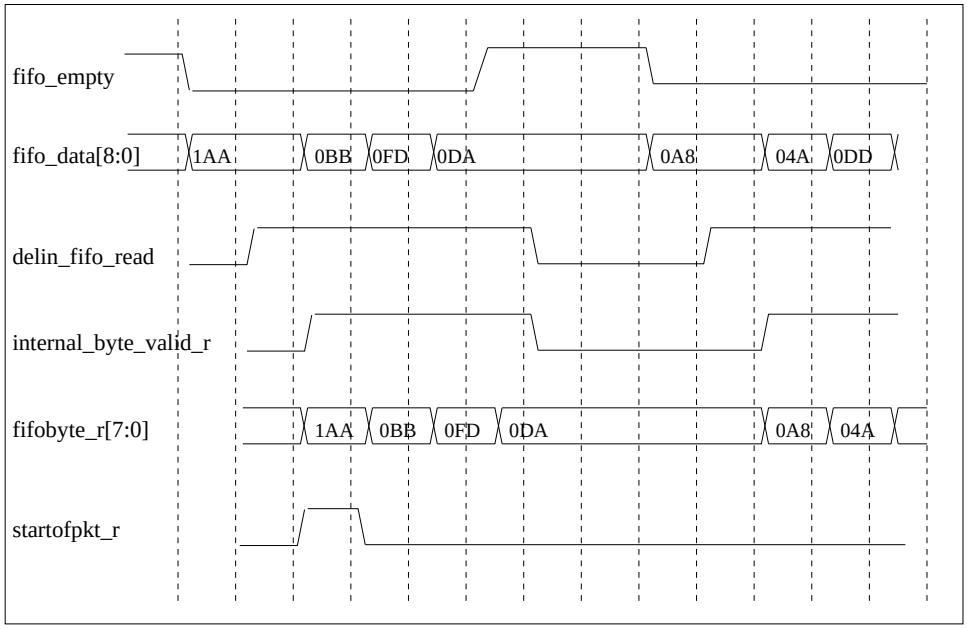

Figure 5.1: Interface waveform for Input FIFO

Figure 5.1 shows the behavior between the Delineator and the input FIFO. When *fifo_empty* is sampled low, the data is read into an internal register. Upon each data unit registered by the Delineator, *delin_fifo_read* is asserted for a clock. Note that for the fifo data to be valid, *fifo_empty* should be low and *delin_fifo_read* should be high. Apart from the data byte, the fifo data delivers a bit indicating the start of a new packet. so the 9-bit fifo data is registered as separate information, i.e *fifobyte_r[7:0]* and *startofpkt_r*. Both these registers are validated by *internal_byte_valid_r*.

- Generation of Internal registers

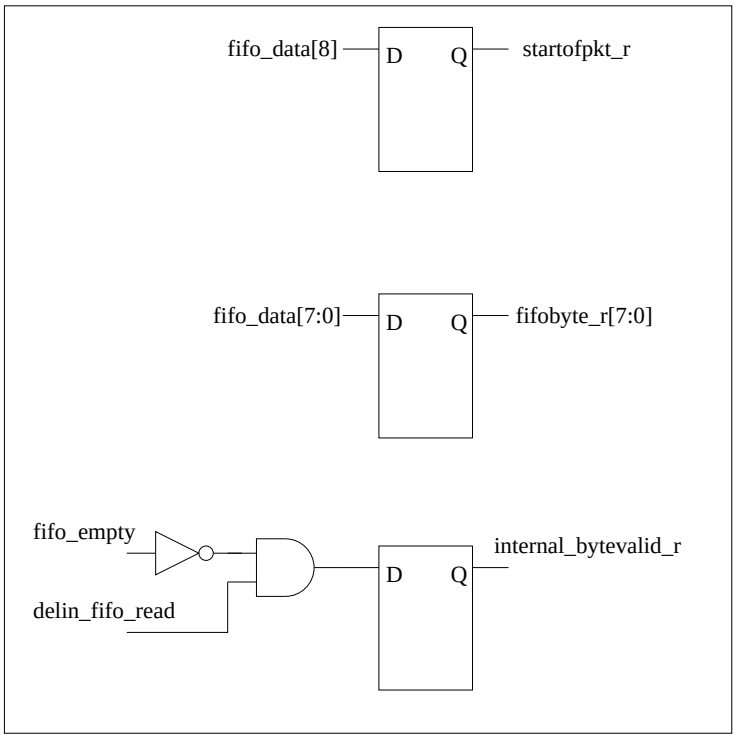

Figure 5.2: Generation of Internal Registers

Figure 5.2 shows the separation of the Input FIFO data into individual registers.

- Generation of Input FIFO Interface

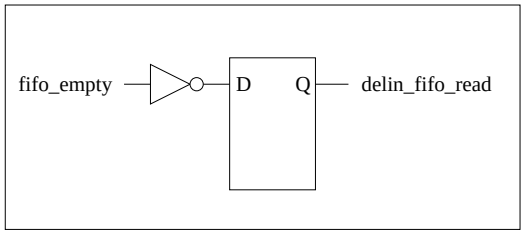

Figure 5.3: Generation of Input FIFO Interface

- Timing Waveform for internal counters and output interfaces

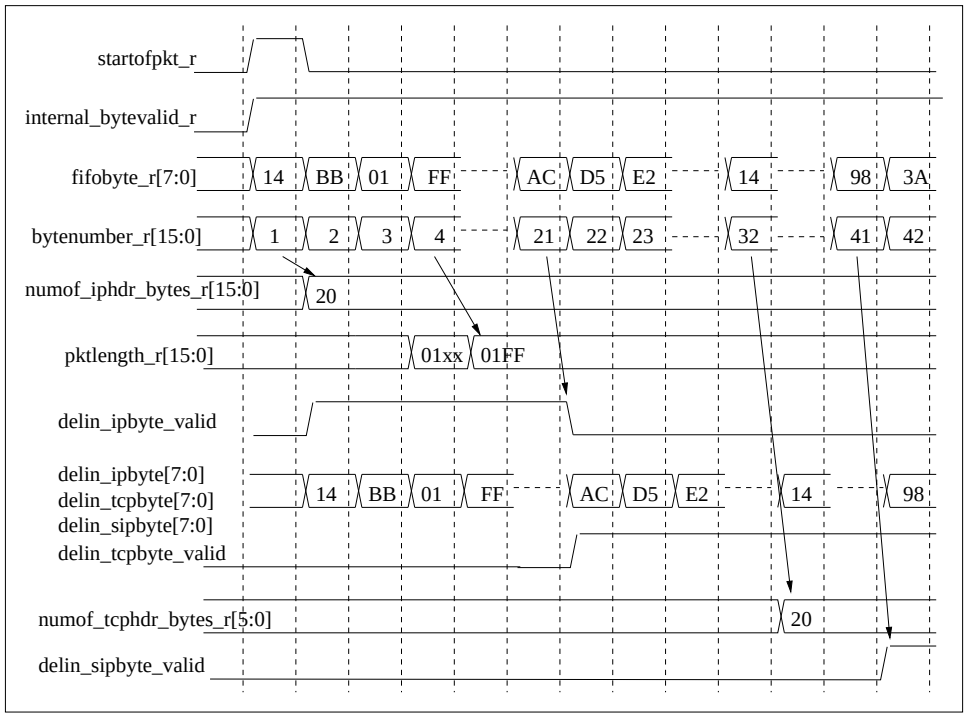

Figure 5.4: Internal counters and interface timing

The timing waveform shown in 5.4 shows the behavior of the output interfaces.

- Generation of the internal byte counter

  The byte counter *bytenumber_r* is a 16-bit counter. It is cleared when a new packet arrives in the input FIFO, flagged by *startofpkt_r*. It is then incremented once for each byte read from the FIFO. Based on the value of this counter, fields of interest are extracted from the TCP and IP headers. A list of the fields and the reason why they are extracted is are given in table 5.6.

Table 5.6: *Table of TCP/IP header fields extracted*

Byte No.	**Header, Name**	**Bits**	**Description**
1	IP, IHL	4	Used to calculate the number of IP header bytes in the packet.
3,4	IP, Total Length	16	Used to determine the total length of the Packet.
11,12	IP, Checksum	16	The received IP checksum. This is stored and compared to the calculated checksum.
13	TCP, Data Offset	4	The length of the TCP Header, tells us when the SIP bytes start in the incoming stream.
17,18	TCP, Checksum	16	The received TCP checksum. This is stored and compared to the calculated checksum.

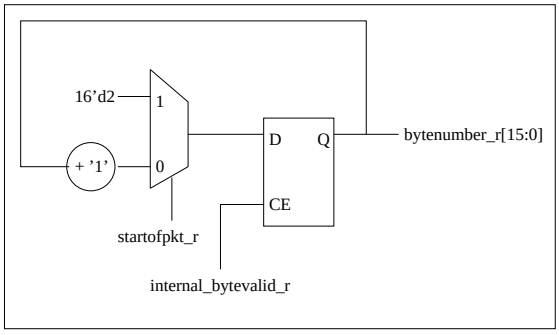

Figure 5.5: Generation of the byte counter

- Generation of the total length, TCP & IP header byte registers

  Depending on the value of *bytenumber_r*, multiple required fields are locally registered.

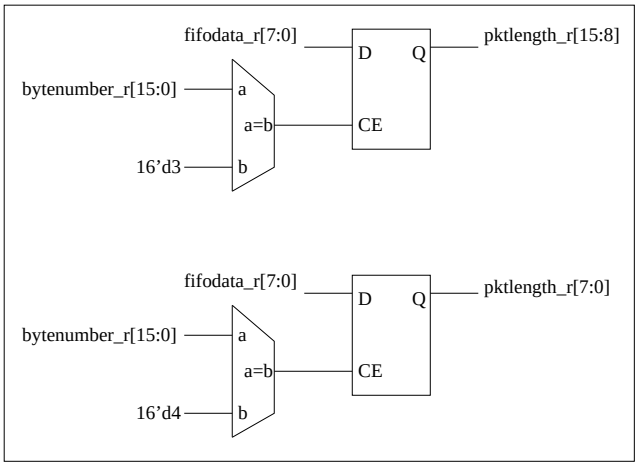

Figure 5.6: Generation of the packet length register

Similarly, the number of TCP and IP header bytes in the current packet are locally registered. It is interesting to note that the location of the TCP header bytes depends on the number of IP header bytes present. These values are later on examined to determine the timing of the output interfaces to the next modules in the design.

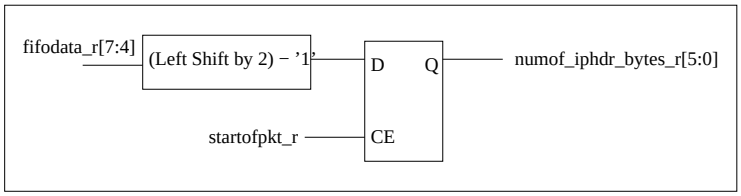

Figure 5.7: Generation of total IP header bytes register

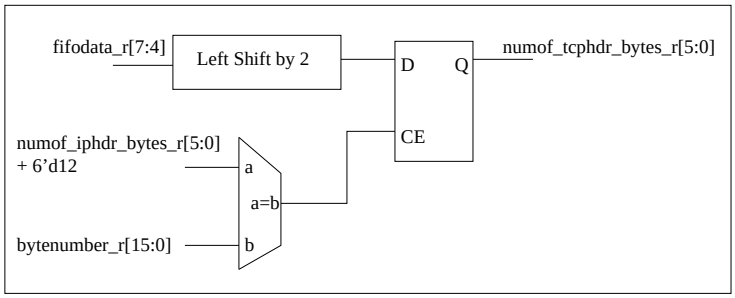

Figure 5.8: Generation of total TCP header bytes register

- Driving the IP Checksum Calculator Interface

  The following hardware diagrams elaborate on how the IP Checksum Calculator interface is driven, based on the local registers and counters. The hardware behind each output pin of the module, on that interface, is shown.

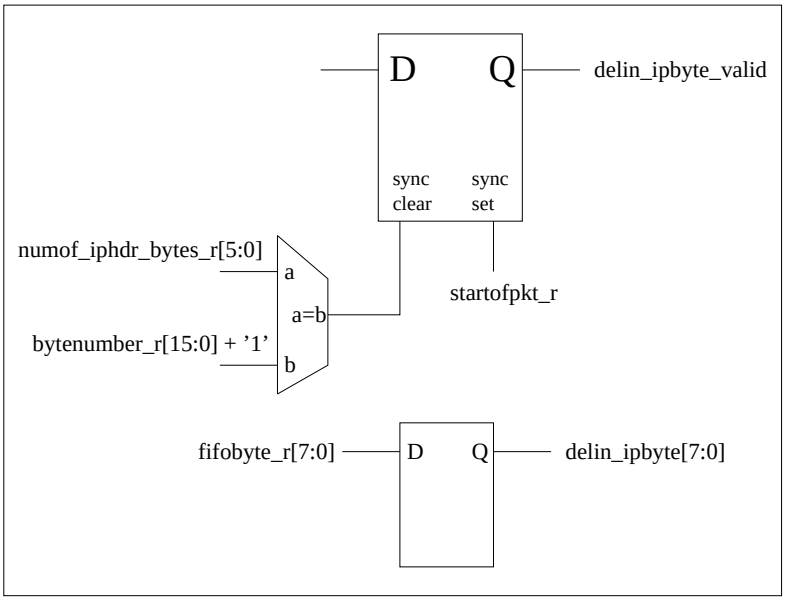

Figure 5.9: Driving the IP Checksum Calculator Interface - 1

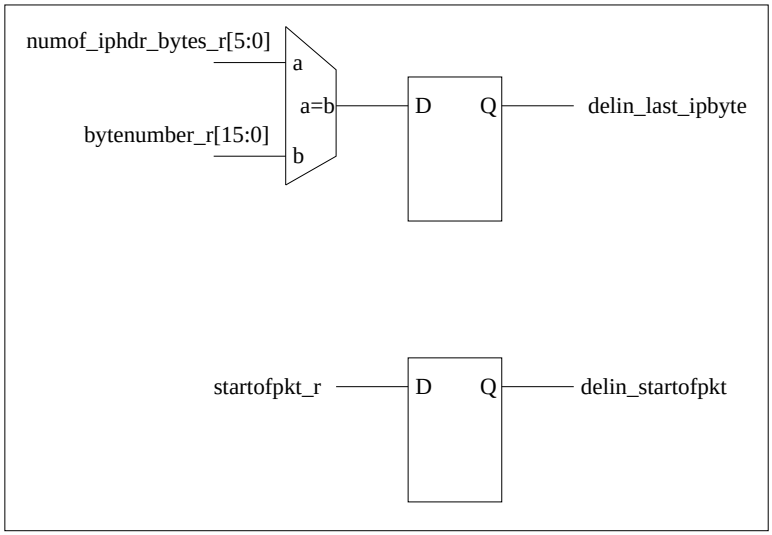

Figure 5.10: Driving the IP Checksum Calculator Interface - 2

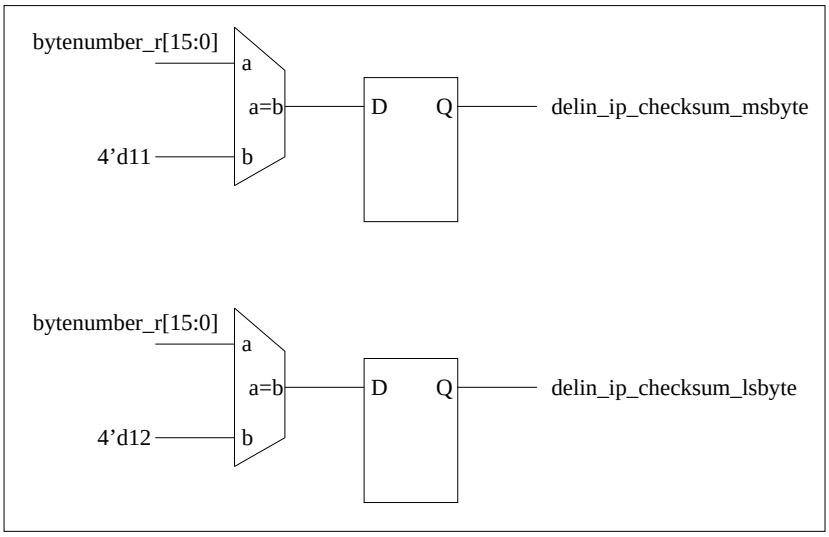

Figure 5.11: Driving the IP Checksum Calculator Interface - 3

- Driving the TCP Checksum Calculator Interface

  The following hardware diagrams elaborate on how the TP Checksum Calculator interface is driven, based on the local registers and counters. Thes hardware behind each output pin of the module, on that interface, is shown.

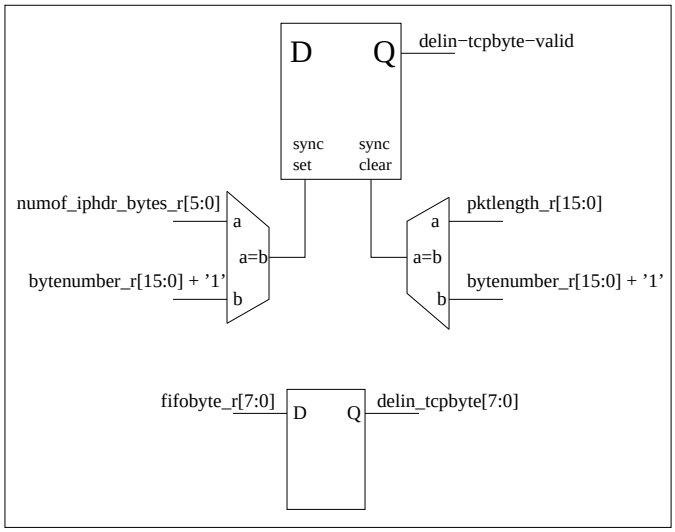

Figure 5.12: Driving the TCP Checksum Calculator Interface - 1

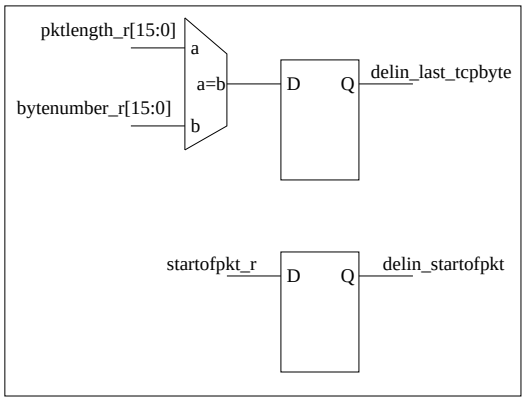

Figure 5.13: Driving the TCP Checksum Calculator Interface - 2

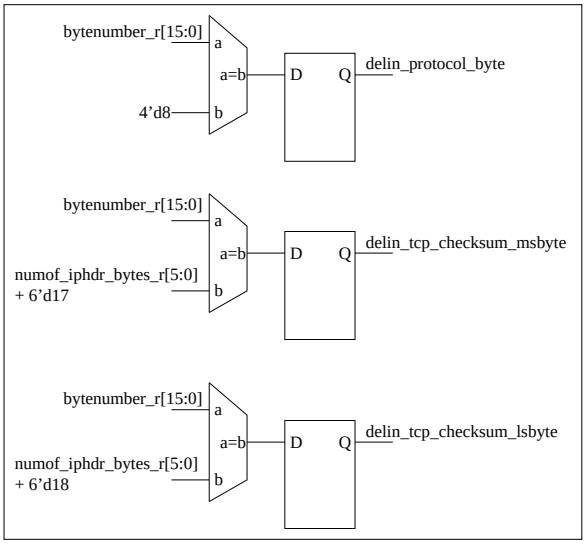

Figure 5.14: Driving the TCP Checksum Calculator Interface - 3

- Driving the SIP Byte Processor Interface

  Figure 5.15 shows the hardware behind the SIP byte processor Interface.

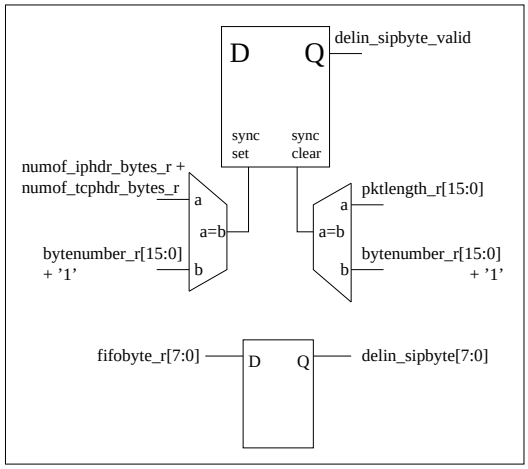

Figure 5.15: Driving the SIP Byte Processor Interface

5.2 Module IP Checksum Calculator (*ipchksum*)

This section presents a detailed description. Section 5.2.1 presents the pin interfaces with connected blocks. Section 5.2.2 describes its implementation in detail.

5.2.1 Pin Interface

This section describes the pin interfaces with the other blocks.

Table 5.7: *Interface with the system*

No.	Pin Name	Dirn.	Width	Description
1	reset	IN	1	The async system reset.
2	clk	IN	1	The system clock.

Table 5.8: *Interface with the Delineator*

No.	Pin Name	Dirn.	Width	Description
1	delin_startofpkt	IN	1	An indication to reset the checksum registers.
2	delin_ipbyte_valid	IN	1	An indication that the current byte is valid. The IP checksum calculator processes the byte only if this signal is set.
3	delin_ip_byte	IN	8	The byte to be processed.
4	delin_ipchksm_msbyte	IN	1	An indication that the current byte is the IP checksum MS byte. The IP checksum calculator needs these bytes to verify the checksum
4	delin_ipchksm_lsbyte	IN	1	Flags the checksum LS byte.
6	delin_last_ipbyte	IN	1	Indicates last valid IP byte. If asserted, the IP checksum calculator matches the received and calculated checksums.

Table 5.9: *Interface with the Buffer Write Controller*

No.	Pin Name	Dirn.	Width	Description
1	ipchksum_write	OUT	1	An indication that the current byte is a valid IP header byte. the Buffer Write Controller samples this valid and transfers this byte to the SIP data structure.
2	ipchksum_byte	OUT	8	The byte to be writtn to the SIP data structure.
3	ipchksum_result_valid	OUT	1	This signal is asserted once the comparison between the calculated checksum and received checksum is made. It is valid for one clock cycle.
4	ipchksum_result_bit	OUT	1	This bit is looked at when ipchksum_result_valid is asserted. If high, it indicates that the received checksum matches with the calculated checksum. If low, it indicates a mismatch in the two checksums.

5.2.2 Architecture

We will start this section by the timing diagram between the Delineator input interface and their effect on registers internal to the IP Checksum Calculator. This is followed by the Buffer Write Controller timing diagram. We will then proceed to elaborate on the hardware implementation of the blocks in the logic schematic shown in 4.5.

- Timing waveform for interface with Delineator.

  Figure 5.16 shows the timing relation between the IP Checksum Calculator and the Delineator. The 16-bit register *ipbyte_accum_r* is an accumulator which stores the incoming byte-wide data as 16-bit data, to match the datawidth

of the IP Header checksum. The incoming byte is either stored in the MS byte or the LS byte of *ipbyte_accum_r*, depending on the value of *byte_select_r*. After every second valid byte is received, i.e 16-bits have beel accumulated, an XOR operation is performed between *ipbyte_accum_r* and a 16-bit XOR register, *ip_checksum_r*. An exception to this continous XOR operation is when the input byte is a checksum byte. In this case, 0x00 is latched into *ipbyte_accum_r*. This is to be in accordance with the IP header checksum calculation method, which zeroes out the original checksum bytes when calculating the checksum over the rest of the IP header. A checksum byte at the input is signaled by either *delin_ipchksm_msbyte* or *delin_ipchksm_lsbyte* being asserted. After all the IP bytes are processed, a write strobe is passed to indicate the result of the checksum verification.

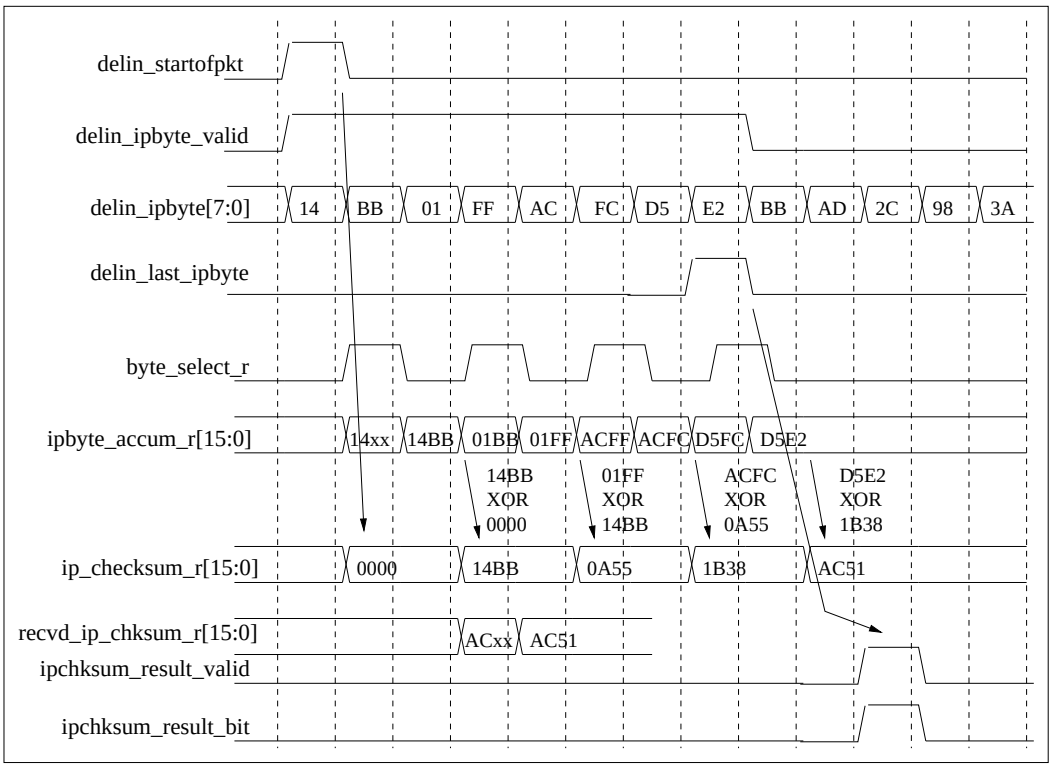

Figure 5.16: Timing relation with Delineator Interface

- Timing waveform for Buffer Write Controller Interface

  Figure 5.17 shows the timing details of the Buffer Write Controller interface. Each byte that comes in from the Delineator has to be passed over to the Buffer Write Controller.

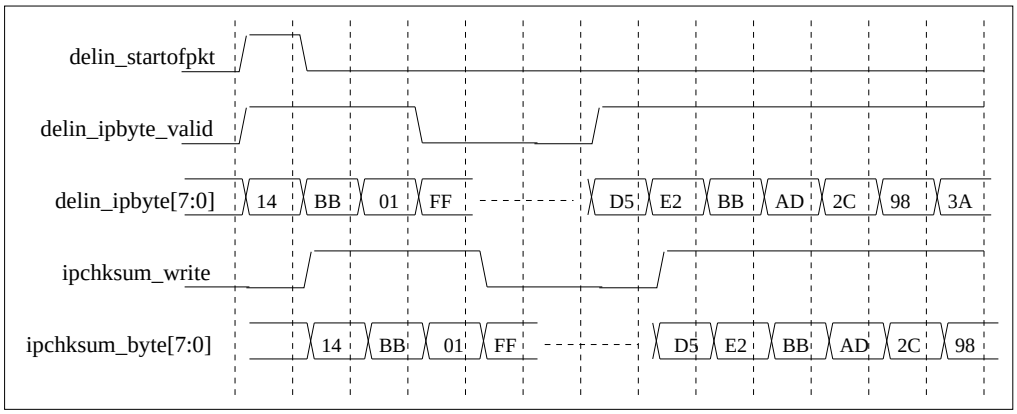

Figure 5.17: Timing relation with Buffer Write Controller Interface

- Generation of *byte_select_r*

  This is a flag which helps is directing bytes from the incoming stream into a 16-bit buffer. This buffer is further used for checksum calculations.

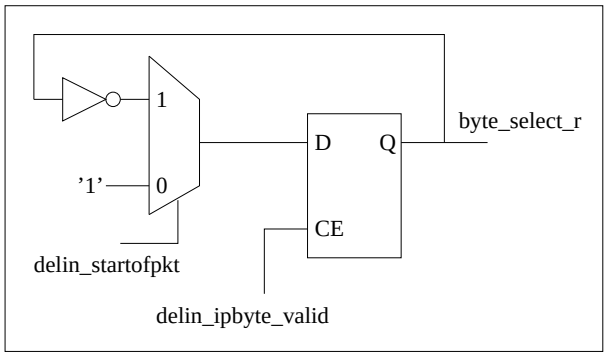

Figure 5.18: Generation of the IP byte select mux flag

- Generation of *ipbyte_accum_r[15:0]*

  This is a 16-bit accumulator which is used to store the IP bytes in pairs for checksum calculations.

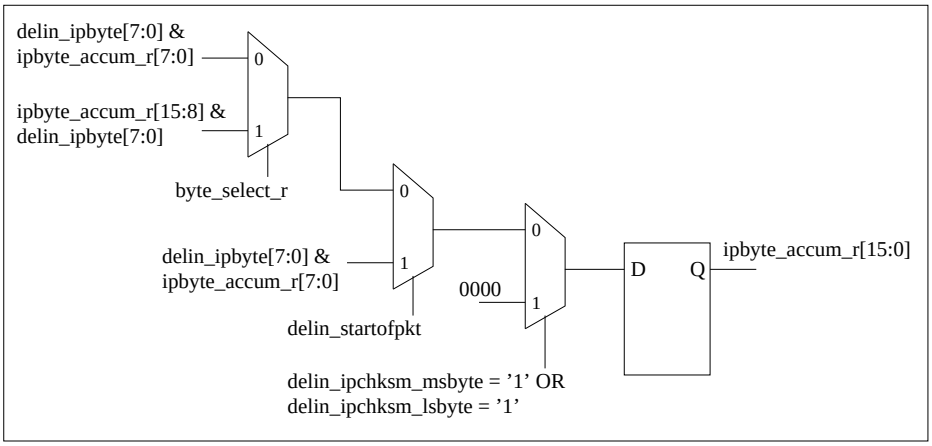

Figure 5.19: Generation of the IP byte accumulator

- Generation of *ip_checksum_r[15:0]*

  This is a 16-bit register which holds the result of the continous XOR operation.

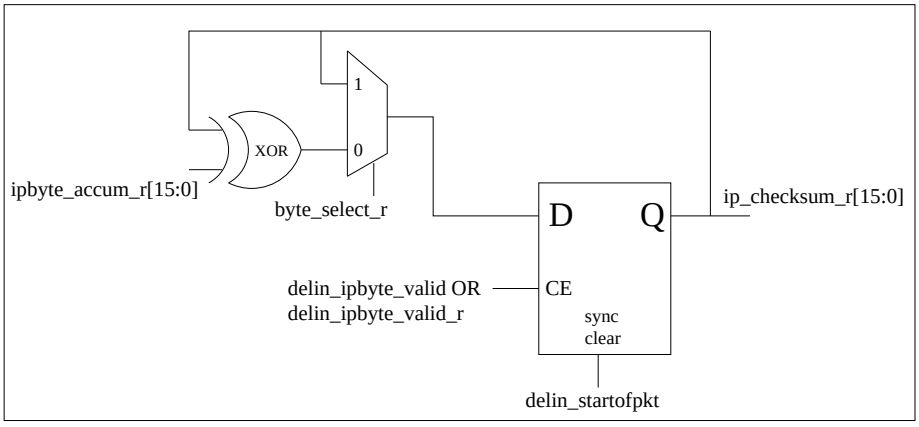

Figure 5.20: Generation of the IP checksum register

- Generation of *recvd_ip_chksum_r[15:0]*

  This is a 16-bit register, used to latch the two checksum bytes from the incoming IP byte stream. This value is used for comparison with the calculated checksum.

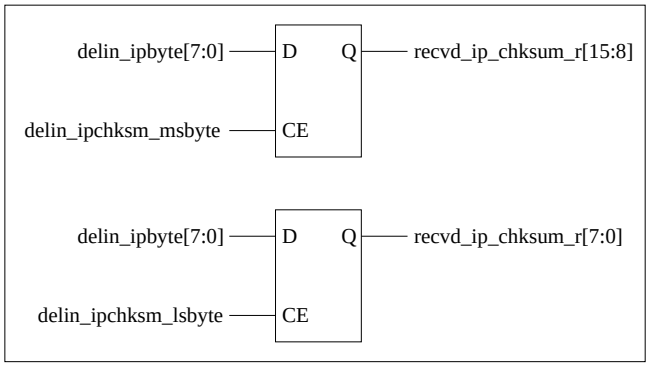

Figure 5.21: Generation of the received IP checksum register

- Generation of the Buffer Write Controller Interface

  Figures 5.22 and 5.23 shows the hardware behind the generation of the Buffer Write Controller Interface.

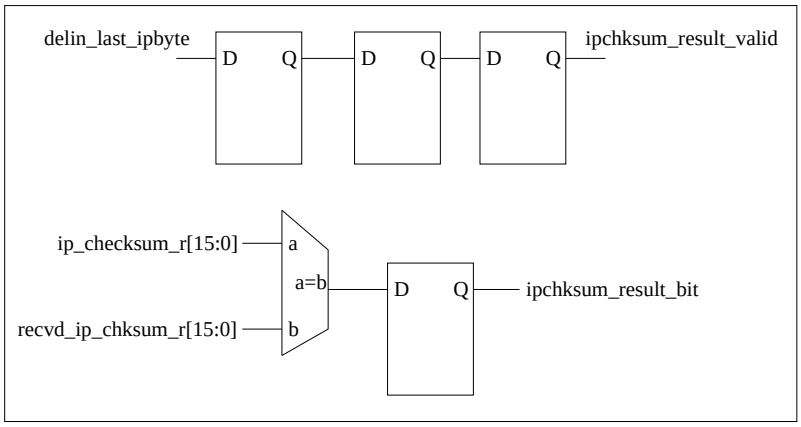

Figure 5.22: Generation of the IP Buffer Write Controller Interface - 1

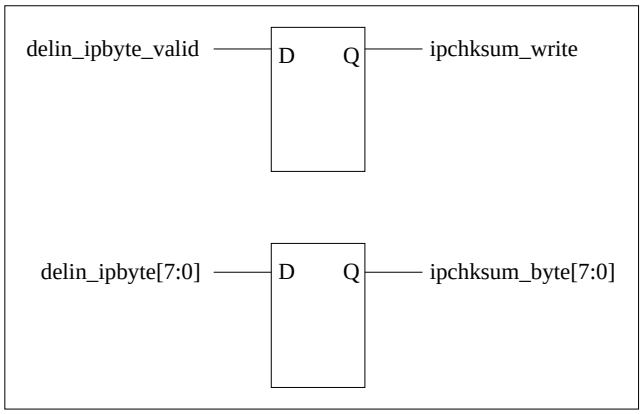

Figure 5.23: Generation of the IP Buffer Write Controller Interface - 2

5.3 Module TCP Checksum Calculator (*tcpchksm*)

This section and its subsection present a detailed description of the TCP Checksum Calculator module. Section 5.3.1 presents the pin interfaces with blocks it interacts with. Section 5.3.2 describes its implementation in detail.

5.3.1 Pin Interface

This section describes the pin interfaces with the other blocks.

Table 5.10: *Interface with the system*

No.	**Pin Name**	**Dirn.**	**Width**	**Description**
1	reset	IN	1	The async system reset.
2	clk	IN	1	The system clock.

Table 5.11: *Interface with the Buffer Write Controller*

No.	Pin Name	Dirn.	Width	Description
1	tcpchksm_write	OUT	1	An indication that the current byte is a valid TCP byte. The Buffer Write Controller transfers this byte to the SIP data structure.
2	tcpchksm_byte	OUT	8	The byte to be written.
3	tcpchksm_result_valid	OUT	1	This signal is asserted once the comparison between the calculated checksum and received checksum is made.
4	tcpchksm_result_bit	OUT	1	This bit is looked at when ipchksum_result_valid is asserted. If high, it indicates that the received checksum matches with the calculated checksum, else a mismatch.
5	tcpchksm_last_tcphdr_byte	OUT	1	An indication that the current byte is the last TCP header byte. This byte is used by the Buffer Write Controller to increment internal memory indices.

Table 5.12: *Interface with Delineator*

No.	Pin Name	Dirn.	Width	Description
1	delin_startofpkt	IN	1	An indication to reset the checksum registers.
2	delin_tcpbyte_valid	IN	1	An indication that the current byte is valid. The TCP checksum calculator processes the byte only if this signal is set.
3	delin_tcp_hdrbyte	IN	1	An indication that the current byte is a TCP Header byte. Only these are sent to the Buffer Write Controller to be written in the SIP Data Structure.
4	delin_last_tcphdr_byte	IN	1	An indication that the current byte is the last TCP header byte. This byte is passed on to the Buffer Write Controller.
5	delin_tcp_byte	IN	8	The byte to be processed.
6	delin_protocol_byte	IN	1	an indication that the current byte is the IP protocol byte. This indication is used in the TCP checksumming process, as a zero pad is required before the protocol field (Fig. 4.4).
7	delin_tcpchksm_msbyte	IN	1	An indication that the current byte is the TCP checksum MS byte. The TCP checksum calculator needs these bytes to verify the checksum.
8	delin_tcpchksm_lsbyte	IN	1	An indication that the current byte is the TCP checksum LS byte.
9	delin_last_tcpbyte	OUT	1	An indication that the current byte is the last valid TCP byte. When asserted, the TCP checksum calculator performs the last XOR operation with the calculated TCP length. It then matches the received and calculated checksums.

5.3.2 Architecture

This section begins with a timing diagram between the Delineator input interface and its effect on registers internal to the TCP Checksum Calculators. This is followed by the Buffer Write Controller timing diagram. We then proceed to elaborate on the hardware implementation of the blocks in the logic schematic shown in 4.6.

- Timing waveform for interface with Delineator.

  Figure 5.24 shows the timing relation between the TCP Checksum Calculator and the Delineator. The 16-bit register *ipbyte_accum_r* is an accumulator which stores the incoming byte-wide data as 16-bit data, to match the datawidth of the TCP Header checksum. The incoming byte is either stored in the MS byte or the LS byte of *tcpbyte_accum_r*, depending on the value of *byte_select_r*. After every second valid byte is received, i.e 16-bits have beel accumulated, an XOR operation is performed between *tcpbyte_accum_r* and a 16-bit XOR register, *tcp_checksum_r*. An exception to this continous XOR operation is when the input byte is a checksum byte. In this case, 0x00 is latched into *tcpbyte_accum_r*. This is to be in accordance with the TCP header checksum calculation method, which zeroes out the original checksum bytes when calculating the checksum over the rest of the TCP header. A checksum byte at the input is signaled by either *delin_tcpchksm_msbyte* or *delin_tcpchksm_lsbyte* being asserted. Another exception is when the signal *delin_protocol_byte* is asserted. When sampled active, the value in *delin_tcp_byte* is pre-padded with zeroes and directly latched into *tcpbyte_accum_r*. After all the TCP bytes are processed, a write strobe is passed to indicate the result of the checksum verification.

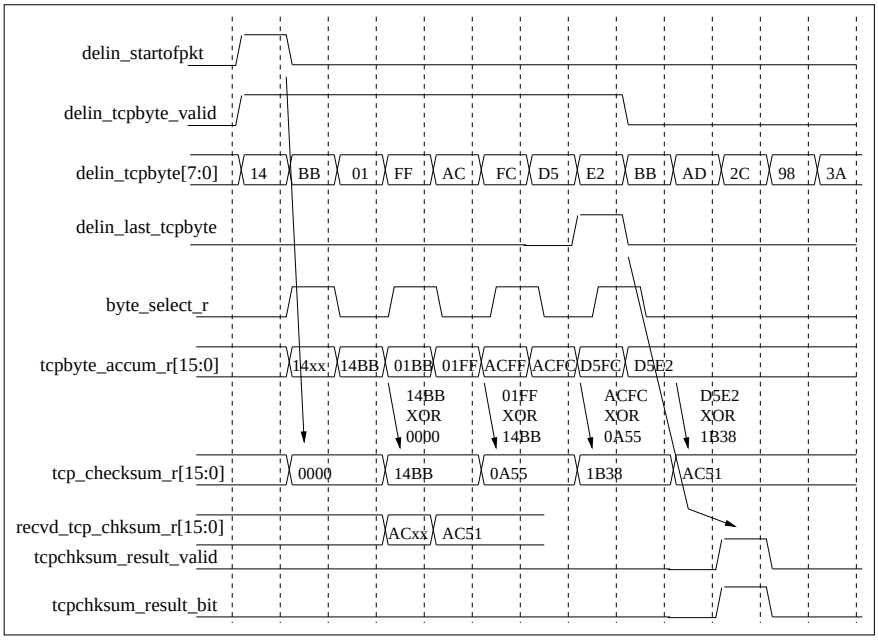

Figure 5.24: Timing relation with Delineator Interface

- Timing waveform for Buffer Write Controller Interface

  Figure 5.25 shows the timing interface. Each byte that comes in from the Delineator has to be passed over to the Buffer Write Controller.

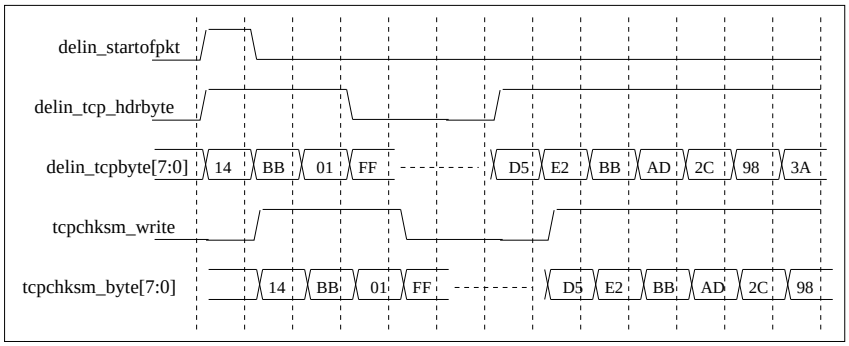

Figure 5.25: Timing relation with Buffer Write Controller Interface

- Generation of *byte_select_r*

  This flag helps in muxing bytes from the incoming stream to a 16-bit buffer used for checksum calculations. However, when the signal *delin_protocol_byte* is asserted, this flag is not toggled to avoid misaligning the next byte. Instead, this byte is padded with zeroes and XOR'ed with the XOR register.

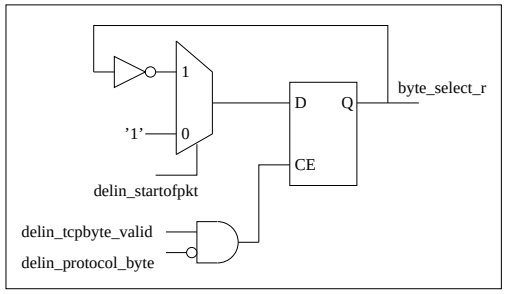

Figure 5.26: Generation of the TCP byte select mux flag

- Generation of *tcpbyte_accum_r[15:0]*

  This is a 16-bit accumulator which is used to store the TCP bytes in pairs for checksum calculations.

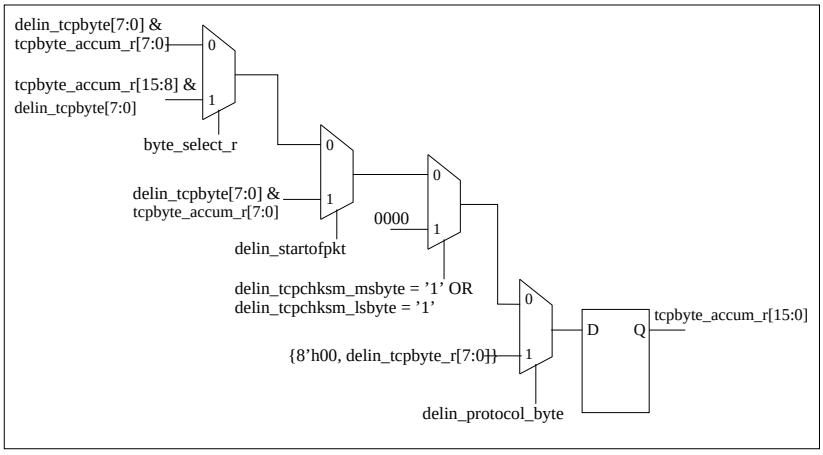

Figure 5.27: Generation of the TCP byte accumulator

- Generation of *tcp_checksum_r[15:0]*

  This is a 16-bit register which holds the result of the continous XOR operation.

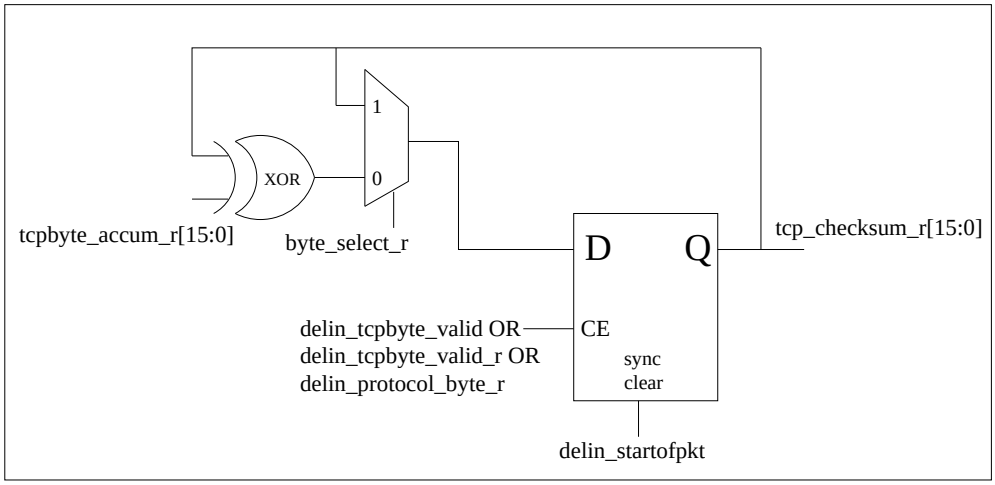

Figure 5.28: Generation of the TCP checksum register

- Generation of *recvd_tcp_chksum_r[15:0]*

  This is a 16-bit register, used to latch the two checksum bytes from the incoming TCP byte stream. This value is used for comparison with the calculated checksum.

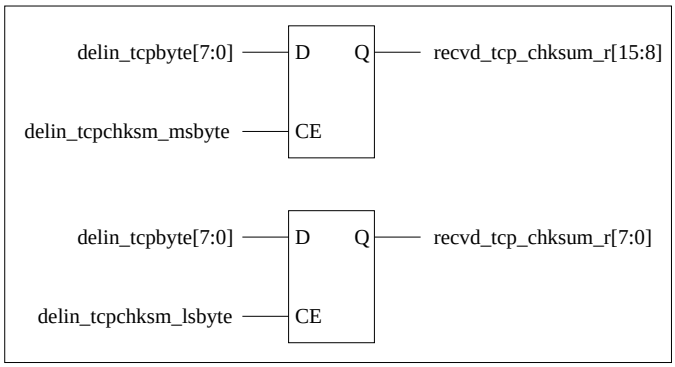

Figure 5.29: Generation of the received TCP checksum register

- Generation of the Buffer Write Controller Interface

  Figures 5.30 and 5.31 shows the hardware behind the generation of the Buffer Write Controller Interface.

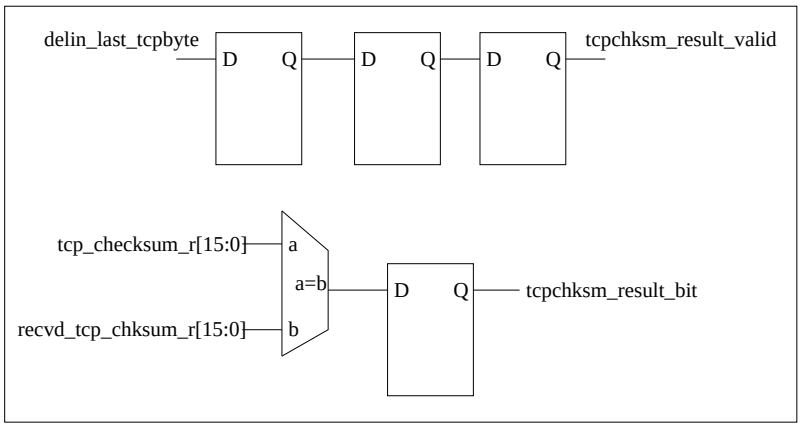

Figure 5.30: Generation of the TCP Buffer Write Controller Interface - 1

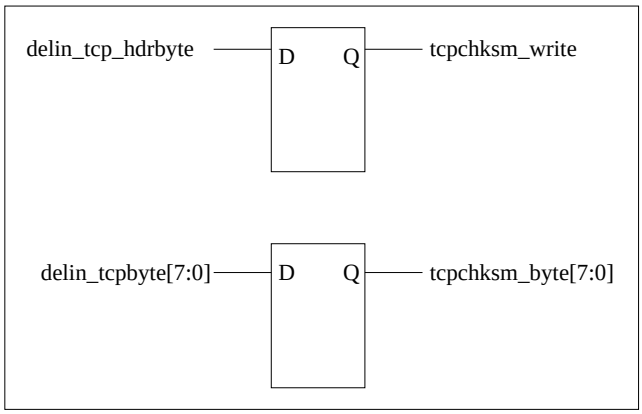

Figure 5.31: Generation of the TCP Buffer Write Controller Interface - 2

5.4 Module SIP Byte Processor (*sipproc*)

This section and its subsection present a detailed description of the SIP Byte Processor module. Section 5.4.1 presents the pin interfaces with blocks it interacts with. Section 5.4.2 describes its implementation in detail.

5.4.1 Pin Interface

This section describes the pin interfaces with the other blocks.

Table 5.13: *Interface with the system*

No.	Pin Name	Dirn.	Width	Description
1	reset	IN	1	The async system reset.
2	clk	IN	1	The system clock.

Table 5.14: *Interface with Delineator*

No.	Pin Name	Dirn.	Width	Description
1	delin_sipbyte_valid	IN	1	An indication that the current byte is valid. The SIP Byte processor processes the byte only if this signal is set.
2	delin_sipbyte	IN	8	The byte to be processed.

Table 5.15: *Interface with Buffer Write Controller*

No.	Pin Name	Dirn.	Width	Description
1	sipproc_fldval_write	OUT	1	An indication for the Buffer Write Controller to write the current byte into the SIP Data Structure. This byte is part of a field value. This write indication is one clock pulse wide.
2	sipproc_fldval_byte	OUT	8	The field value byte to be written to the SIP Data Structure.
3	sipproc_hdr_done	OUT	1	An indication that the SIP Byte processor has finished processing a header name,value pair. This indication is used to determine the offset from which the next header value will be stored.
4	sipproc_match_found	OUT	1	An indication that the SIP Byte processor has found a match. Looking at this indication, the Buffer Write Controller calculates the offset pointer for that field name and stores it in the SIP Data Structure.
5	sipproc_match_id	OUT	5	The unique match ID identifying which field name was correctly matched. The Buffer Write Controller uses this ID as a memory index to the SIP Data Strcuture, to store the pointer offset. It also sets the corresponding bit in the 44-bit Field name found register.
6	sipproc_match_bytes	OUT	9	The number of bytes in the field value. This value is used by the Buffer Write Controller to calculate the pointer offset for the next field value.

5.4.2 Architecture

This section begins with a flowchart , shown in Figure 5.32 depicting the way the logic proceeds in the design. This is followed by the state machine diagram , shown in Figure 5.35 and its transition table. The interface timing with the Delineator block and the way internal registers behave is shown in Figure 5.36. This is followed by implementation details of the internal registers. the generation of the output Buffer Write Controlller interface is discussed.

- Logic Flowchart. The overall flowchart for the module is shown in Figure 5.32. Each block in the flowchart can be thought of as a task to perform.

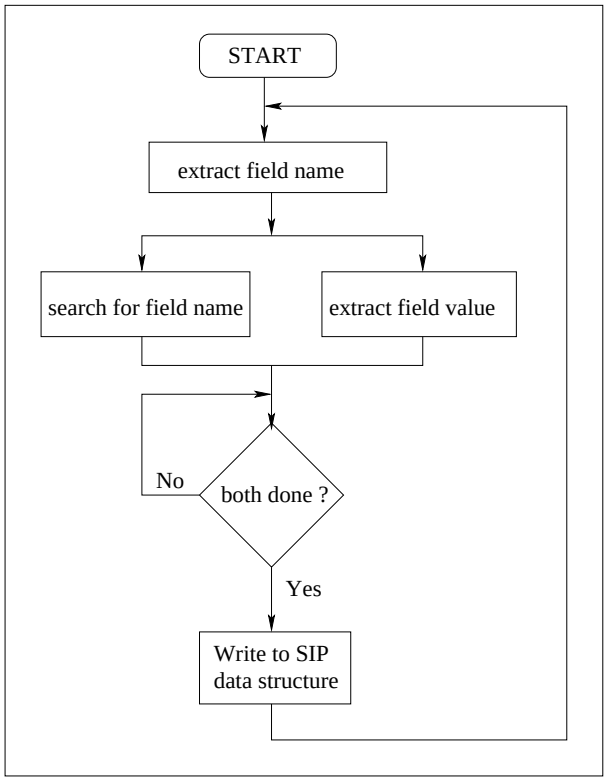

Figure 5.32: Flowchart for the SIP Byte Processor

- Modified Aho-Corasick search algorithm.

String 1				String 2				String 3				String 4				String 5			Result
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	Match ID
a	c	c	e	p	t														0
a	c	c	e	p	t	–	e	n	c	o	d	i	n	g					1
							l	a	n	g	u	a	g	e					2
		k																	3
	l	e	r	t	–	i	n	f	o										4
		l	o	w															5
	u	t	h	e	n	t	i	c	a	t	i	o	n	–	i	n	f	o	6
				o	r	i	z	a	t	i	o	n							7
b	y	e																	8
c	a	n	c	e	l														9
		l	l	–	i	d													10
						n	f	o											11
	o	n	t	a	c	t													12
				e	n	t	–	d	i	s	p	o	s	i	t	i	o	n	13
								e	n	c	o	d	i	n	g				14
								l	a	n	g	u	a	g	e				15
									e	n	g	t	h						16
								t	y	p	e								17
	s	e	q																18
d	a	t	e																19
e	r	r	o	r	–	i	n	f	o										20
	x	p	i	r	e	s													21
f	r	o	m																22
i	n	–	r	e	p	l	y	–	t	o									23
		v	i	t	e														24
m	a	x	–	f	o	r	w	a	r	d	s								25
	i	n	–	e	x	p	i	r	e	s									26
		m	e	–	v	e	r	s	i	o	n								27
o	p	t	i	o	n	s													28
	r	g	a	n	i	z	a	t	i	o	n								29
p	r	i	o	r	i	t	y												30
		o	x	y	–	a	u	t	h	e	n	t	i	c	a	t	e		31
										o	r	i	z	a	t	i	o	n	32
						r	e	q	u	i	r	e							33
r	e	g	i	s	t	e	r												34
		c	o	r	d	–	r	o	u	t	e								35
		p	l	y	–	t	o												36
		q	u	i	r	e													37
		t	r	y	–	a	f	t	e	r									38
	o	u	t	e															39
s	e	r	v	e	r														40
	u	b	j	e	c	t													41
		p	p	o	r	t	e	d											42
t	i	m	e	s	t	a	m	p											43
	o																		44
u	n	s	u	p	p	o	r	t	e	d									45
	s	e	r	–	a	g	e	n	t										46
v	i	a																	47
w	w	w	–	a	u	t	h	e	n	t	i	c	a	t	e				48
	a	r	n	i	n	g													49

Figure 5.33: SIP keyword search structure

The Aho-Corasick search algorithm is used for exact pattern matching. First, a keyword search structure is constructed by examining all the possible keywords, as shown in Figure 5.33. The current state is maintained based on each character received. If characters are received in a valid sequence, then the state keeps changing until a valid pattern is reached. If an out-of-sequence character is received, then a failure condition exists.

The approach used in the SIP byte processor is a modified version to speed up the searching. Instead of searching one character at a time, a string consisting of four characters is searched for at a time. Considering the maximum string length of 19 characters for the keyword "Proxy-Authentication", this would take a maximum of 5 searches cycles. Keywords occupy from 1 upto 5 strings, leading to a search time of 1-5 cycles. The search method is shown in Figure 5.34. At the end of the each clock cycle, a code is generated as the result of the search. This code is the input for the second search cycle and so on. This continues till a match is found. Note that a mismatch could occur at any cycle of searching.

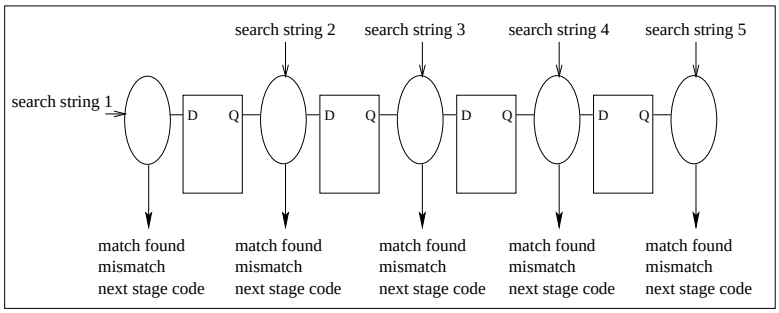

Figure 5.34: SIP keyword search flow

- State Machine Diagram

  Figure 5.35 shows the state machine for the SIP byte processor. The next state is evaluated for each byte that comes in. As soon as a valid byte appears on the Delineator interface, it's ASCII value is examined and a set of flags are generated, shown in Table 5.16. Based on these values, the next state is decoded. Based on the current state, various counters and interfaces are driven, as will be explained in further sections.

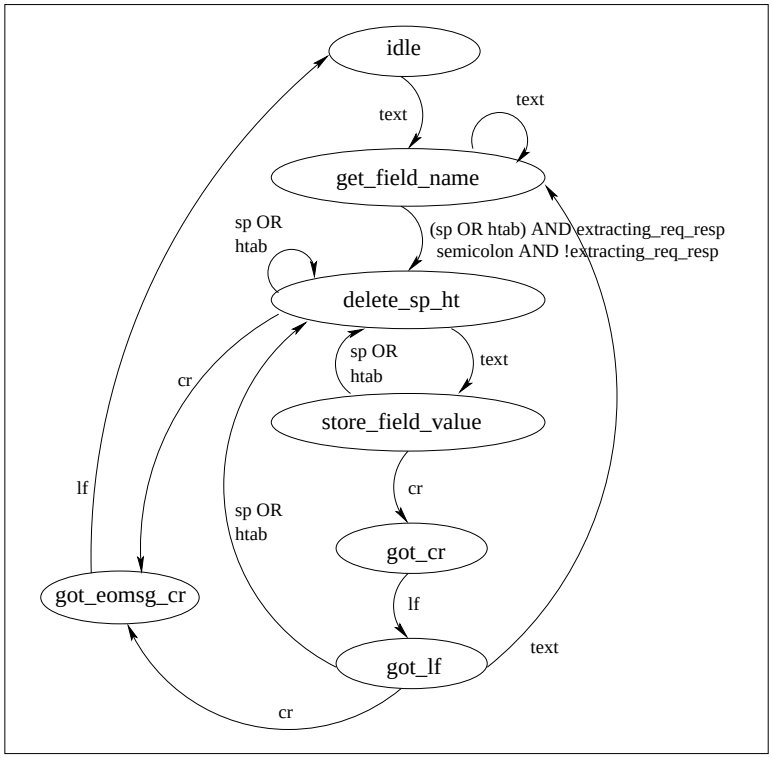

Figure 5.35: State Machine Diagram

- Timing Waveform for Interface with Delineator.

  Figure 5.36 shows the timing relation with the Delineator interface. It can be seen how the status flags are generated based on the ASCII value, the byte is then encoded and stored in a search string. While the string is being searched,

the field value is sent to the Buffer Write Controller to be written in the SIP Data Structure. Once the search is complete, the match ID and the byte offset are also conveyed for update.

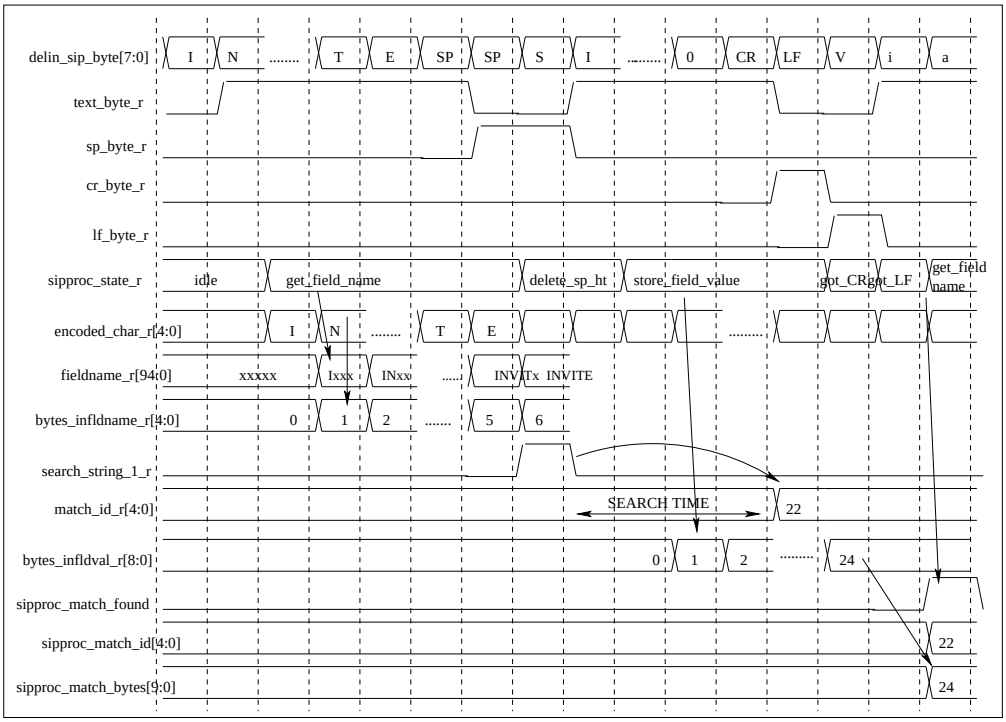

Figure 5.36: Timing waveform for Delineator interface

- ASCII table for SIP characters.

  Table 5.16 shows the ASCII value of SIP characters and the flags associated with them.

Table 5.16: *Delimiter Table*

No.	**Flag Name**	**ASCII Val.**	**Description**
1	text_byte_r	0x41 - 0x5A	Text bytes A-Z.
2	text_byte_r	0x61 - 0x7A	Text bytes a-z.
3	text_byte_r	0x2D	The minus/dash '-'. this byte has to be considered as text because it could also be a valid part of a field name/value. Ex. Call-ID.
4	sp_byte_r	0x20	A space byte, used to delimit tokens or fold lines for longer field values.
5	ht_byte_r	0x09	A horizontal tab byte, also used to delimit tokens or fold lines for longer field values.
6	cr_byte_r	0x0D	A carriage return byte. Used in conjunction with the Line feed indication to signify the end of a ¡field_name : field_value¿ pair.
7	lf_byte_r	0x0A	A line feed byte. Used in conjunction with the carriage return indication to signify the end of a field_name : field_value¿ pair.
8	semicolon_byte_r	0x3B	A semicolon byte. Used to delimit the field value from the field name.

- Generation of the byte status flags.

  This logis is a part of the token extractor block shown in figure 4.7. Multiple flags are generated, which affect the state machine transitions. These flags are generated by examining the ASCII value of the byte and accordingly classifying the byte as shown in Table 5.16.

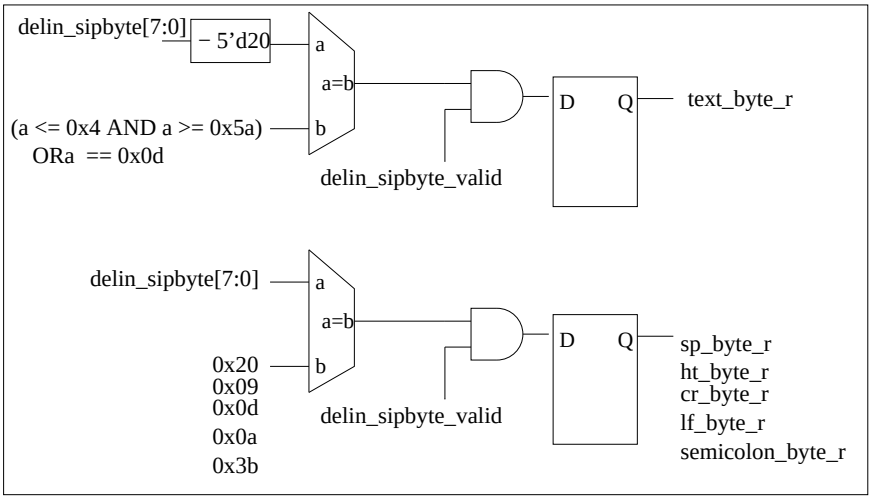

Figure 5.37: Generation of the byte status flags

- Generation of the processor current state *proc_state_r*

  Based on the current state, the next state is decoded and registered into *proc_state_r*. The next state is decided based on the status of the byte flags.

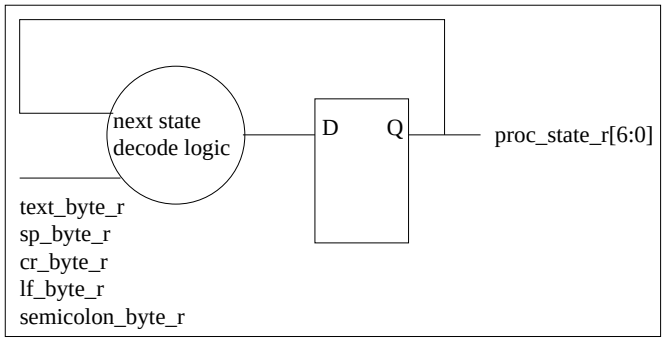

Figure 5.38: Generation of the processor current state

- Encoding the SIP byte.

  The bytes identified to be a part of a SIP search string are collected in a buffer and are checked against the SIP keywords for a match. The Aho-Corasick pattern matching algorithm [14] is used to check for SIP keywords. The algorithm performs a comparison as a part of its functionality. To consume lesser hardware, the 8-bit SIP byte is encoded in a 5-bit character. This is possible because not all 256 combinations of the SIP byte are valid SIP characters. For example, '&' or '*' are never a part of a SIP keyword. After examintaion of the list of SIP keywords in [2], it is noticed that the keywords are made up of the alphabets (A-Z, a-z) and the dash ('-') sign. After converting lower case alphabet ASCII to uppercase ASCII, we would need 27 unique characters (5 bits) to identify any SIP character.

  To understand the encoding, let us take an example. If we were to match the SIP keyword "INVITE" using byte-wide characters, it would require a (6*8=48) bit comparator. If we searched with the encoded characters, it would result in a (6*5=30) bit comparator. Reducing the width of the comparison also allows for

faster search results. The encoded value is acheived using the simple flowchart shown in Figure 5.39.

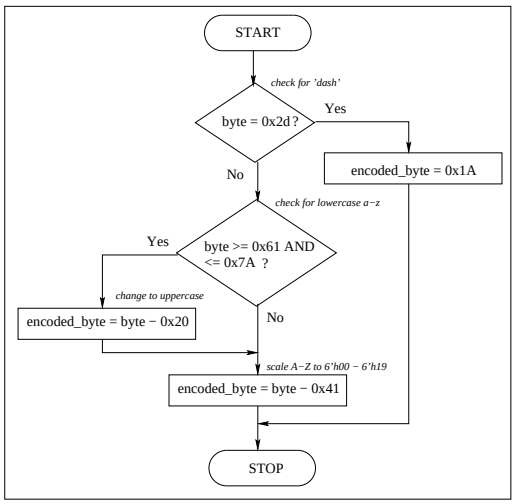

Figure 5.39: Flowchart for encoding the SIP byte

Table 5.17: *Table for encoded character values*

Alphabet	**Encoded value**	**Alphabet**	**Encoded value**
a	5'h00	n	5'h0d
b	5'h01	o	5'h0e
c	5'h02	p	5'h0f
d	5'h03	q	5'h10
e	5'h04	r	5'h11
f	5'h05	s	5'h12
g	5'h06	t	5'h13
h	5'h07	u	5'h14
i	5'h08	v	5'h15
j	5'h09	w	5'h16
k	5'h0a	x	5'h17
l	5'h0b	y	5'h18
m	5'h0c	z	5'h19
-	5'h1a		

- Generation of the search string.

  The string to be searched is stored in a buffer *fieldname_r[113:0]*. This buffer is sized to store the longest SIP keyword i.e *Proxy-Authorization*. The width required to store it would be (19*5) = 95 bits, since this keyword has 19 characters. We need an index to store each encoded character. To index 19 characters, we implemented the index as a 5-bit counter *bytes_in_fldname_r[4:0]*.

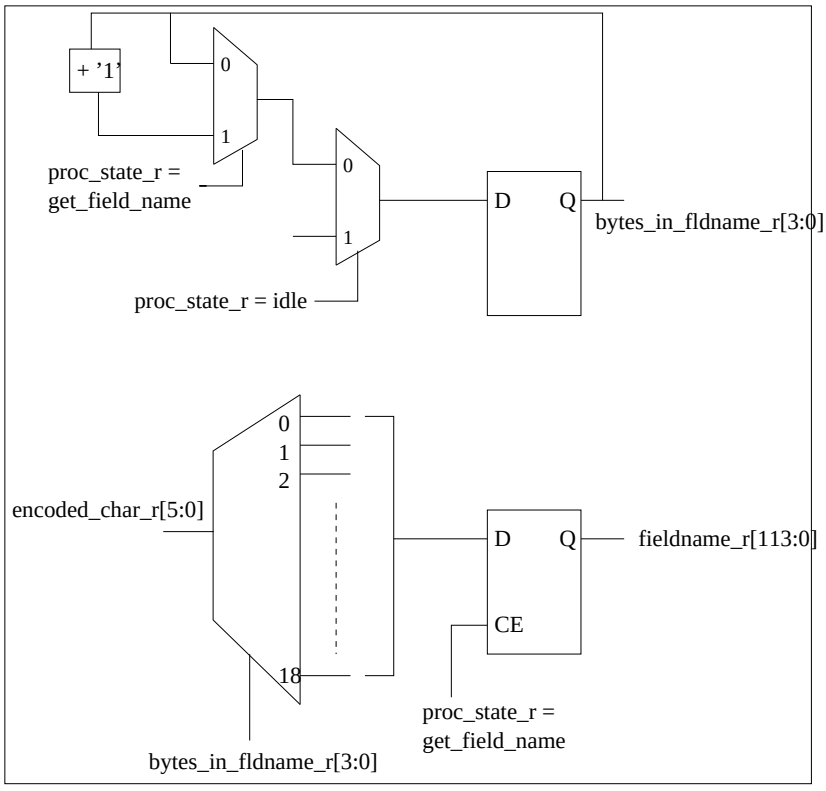

Figure 5.40: Generation of the search string

- Generation of the search string flags.

  A set of five flags are generated once the SIP search string is extracted. Each flag signifies which cycle of the search operation is currently taking place. When each flag is asserted, a corresponding code is generated, which is used for matching in the next search cycle. The behavior of these flags is shown in Figure 5.41.

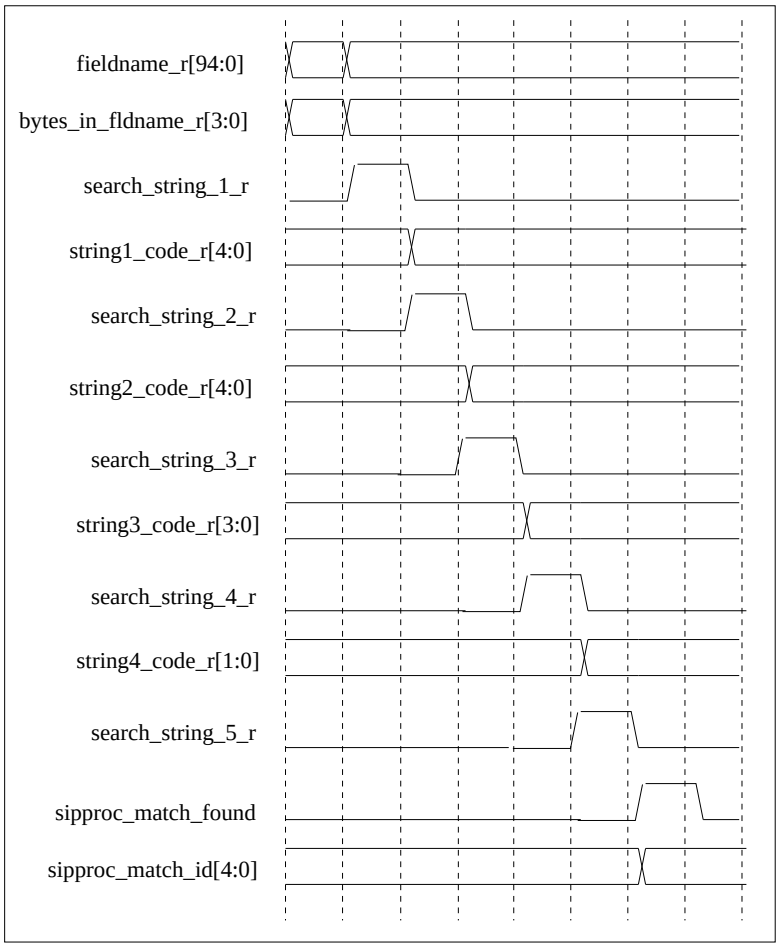

Figure 5.41: Timing for the search operation

The generation of the search flags is shown in Figure 5.42

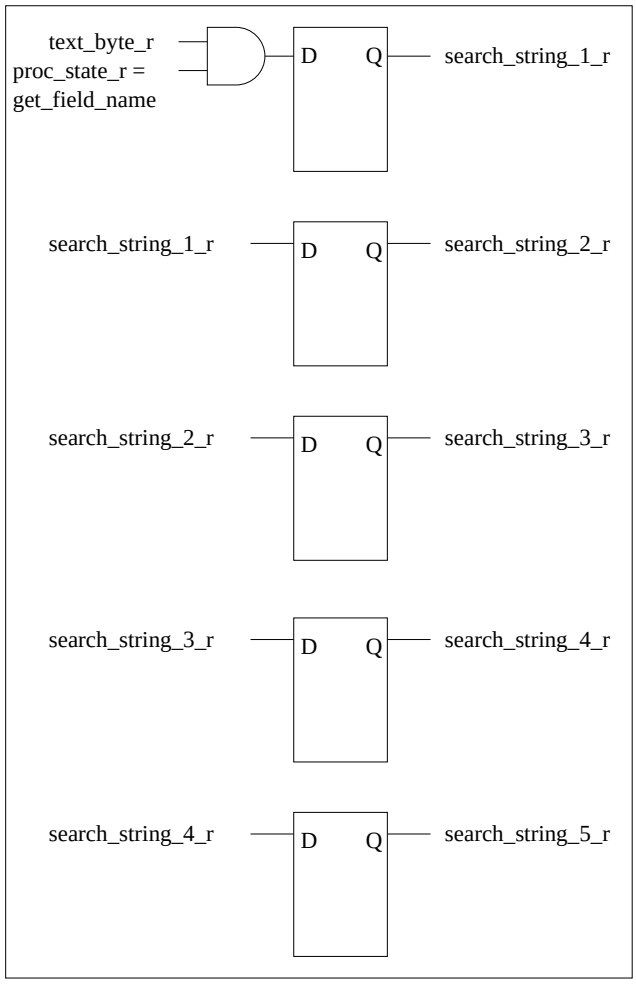

Figure 5.42: Generation of the search string flags

- Generation of the string codes.

  At each search cycle, a 5-bit code is generated which acts as an input to the next search cycle. The value of this code depends on the value of the current search string. This code acts like a unique link to the next string. Tables 5.18 through 5.22 show the codes to be generated at each search cycle.

Table 5.18: *Output codes for String1*

String1	**Bytes**	**Match**	**Match ID**	**String1 Code**
acce	X	No	X	5'b00000
aler	X	No	X	5'b00001
allo	X	No	X	5'b00010
auth	X	No	X	5'b00011
bye	3	Yes	7	X
canc	X	No	X	5'b00100
call	X	No	X	5'b00101
cont	X	No	X	5'b00110
cseq	4	Yes	17	X
date	4	Yes	18	X
erro	X	No	X	5'b00111
expi	X	No	X	5'b01000
from	4	Yes	21	X
in-r	X	No	X	5'b01001
invi	X	No	X	5'b01010
max-	X	No	X	5'b01011
min-	X	No	X	5'b01100
mime	X	No	X	5'b01101
opti	X	No	X	5'b01110
orga	X	No	X	5'b01111
prio	X	No	X	5'b10000
prox	X	No	X	5'b10001
regi	X	No	X	5'b10010
reco	X	No	X	5'b10011
repl	X	No	X	5'b10100
requ	X	No	X	5'b10101
retr	X	No	X	5'b10110
rout	X	No	X	5'b10111
serv	X	No	X	5'b11000
subj	X	No	X	5'b11001
supp	X	No	X	5'b11010
time	X	No	X	5'b11011
to	2	Yes	43	X
unsu	X	No	X	5'b11100
user	X	No	X	5'b11101
via	3	Yes	46	X
www-	X	No	X	5'b11110
warn	X	No	X	5'b11111

Table 5.19: *Output codes for String2*

String1 Code	String2	Bytes	Match	Match ID	String2 Code
5'b00000	pt	6	Yes	0	X
5'b00000	pt-e	X	No	X	5'b00000
5'b00000	pt-l	X	No	X	5'b00001
5'b00001	t-in	X	No	X	5'b00010
5'b00010	w	5	Yes	4	X
5'b00011	enti	X	No	X	5'b00011
5'b00011	oriz	X	No	X	5'b00100
5'b00100	el	6	Yes	8	X
5'b00101	-id	7	Yes	9	X
5'b00101	-inf	X	No	X	5'b00101
5'b00110	act	7	Yes	11	X
5'b00110	ent-	X	No	X	5'b00110
5'b00111	r-in	X	No	X	5'b00111
5'b01000	res	3	Yes	20	X
5'b01001	eply	X	No	X	5'b01000
5'b01010	te	6	Yes	23	X
5'b01011	forw	X	No	X	5'b01001
5'b01100	expi	X	No	X	5'b01010
5'b01101	-ver	X	No	X	5'b01011
5'b01110	ons	3	Yes	27	X
5'b01111	niza	X	No	X	5'b01100
5'b10000	rity	8	Yes	29	X
5'b10001	y-au	X	No	X	5'b01101
5'b10001	y-re	X	No	X	5'b01110
5'b10010	ster	8	Yes	33	X
5'b10011	rd-r	X	No	X	5'b01111
5'b10100	y-to	8	Yes	35	X
5'b10101	ire	7	Yes	36	X
5'b10110	y-af	X	no	X	5'b10000
5'b10111	e	5	Yes	38	X
5'b11000	er	6	Yes	39	X
5'b11001	ect	7	Yes	40	X
5'b11010	orte	X	No	X	5'b10001
5'b11011	stam	X	No	X	5'b10010
5'b11100	ppor	X	No	X	5'b10011
5'b11101	-age	X	No	X	5'b10100
5'b11110	auth	X	No	X	5'b10101
5'b11111	ing	7	Yes	48	X

Table 5.20: *Output codes for String3*

String2 Code	**String3**	**Bytes**	**Match**	**Match ID**	**String3 Code**
5'b00000	ncod	X	No	X	4'b0000
5'b00001	angu	X	No	X	4'b0001
5'b00010	fo	10	Yes	3	X
5'b00011	cati	X	No	X	4'b0010
5'b00100	atio	X	No	X	4'b0011
5'b00101	o	9	Yes	10	X
5'b00110	disp	X	No	X	4'b0100
5'b00110	enco	X	No	X	4'b0101
5'b00110	lang	X	No	X	4'b0110
5'b00110	leng	X	No	X	4'b0111
5'b00110	type	12	Yes	16	X
5'b00111	fo	10	Yes	19	X
5'b01000	-to	11	Yes	22	X
5'b01001	ards	12	Yes	24	X
5'b01010	res	11	Yes	25	X
5'b01011	sion	12	Yes	26	X
5'b01100	tion	12	Yes	28	X
5'b01101	then	X	No	X	4'b1000
5'b01101	thor	X	No	X	4'b1001
5'b01110	quir	X	No	X	4'b1010
5'b01111	oute	12	Yes	34	X
5'b10000	ter	11	Yes	37	X
5'b10001	d	9	Yes	41	X
5'b10010	p	9	Yes	42	X
5'b10011	ted	11	Yes	44	X
5'b10100	nt	10	Yes	45	X
5'b10101	enti	X	No	X	4'b1011

Table 5.21: *Output codes for String4*

String3 Code	String4	Bytes	Match	Match ID	String4 Code
4'b0000	ing	15	Yes	1	X
4'b0001	age	15	Yes	2	X
4'b0010	on-i	X	No	X	2'b00
4'b0011	n	13	Yes	6	X
4'b0100	osit	X	No	X	2'b01
4'b0101	ding	16	Yes	13	X
4'b0110	uage	16	Yes	14	X
4'b0111	th	14	Yes	15	X
4'b1000	tica	X	No	X	2'b10
4'b1001	izat	X	No	X	2'b11
4'b1010	e	13	Yes	32	X
4'b1011	cate	16	Yes	47	X

Table 5.22: *Output codes for String5*

String4 Code	String5	Bytes	Match	Match ID
2'b00	nfo	19	Yes	5
2'b01	ion	19	Yes	12
2'b10	te	18	Yes	30
2'b11	ion	19	Yes	31

- Generation of the Buffer Write Controller Interface

  The generation of the Buffer Write Controller Interface is shown in figure 5.43. This interface is asserted when the field value bytes and the match index are to be written to the SIP Data Structure.

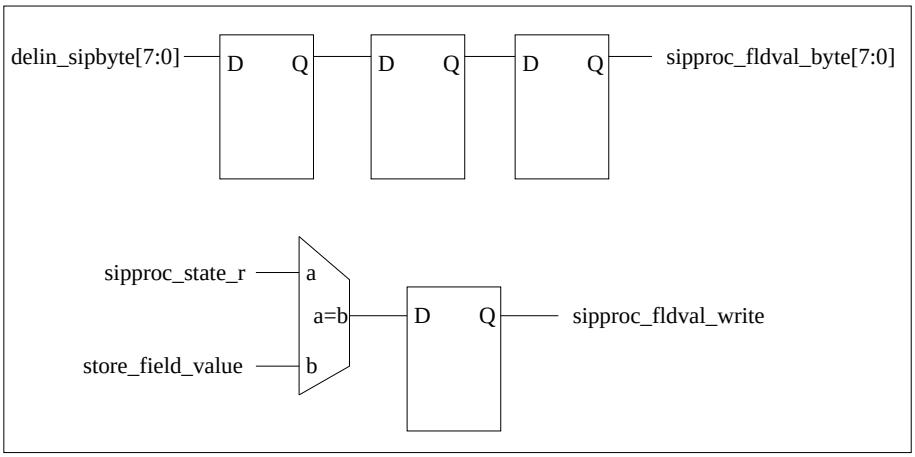

Figure 5.43: Generation of the Buffer Write Controller Interface

5.5 Module Buffer Write Controller (*bufwrctr*)

This section and its subsection present a detailed description of the Buffer Write controller Module. Section 5.5.1 presents the pin interfaces with blocks it interacts with. Section 5.5.2 describes its implementation in detail.

5.5.1 Pin Interface

This section describes the pin interfaces with the other blocks.

Table 5.23: *Interface with the system*

No.	Pin Name	Dirn.	Width	Description
1	reset	IN	1	The async system reset.
2	clk	IN	1	The system clock.

Table 5.24: *Interface with the IP Checksum Calculator*

No.	Pin Name	Dirn.	Width	Description
1	ipchksum_write	IN	1	Upon sampled active, the Buffer Write Controller transfers this byte to the SIP data structure.
2	ipchksum_byte	IN	8	The byte to be writtn to the SIP data structure.
3	ipchksum_result_valid	IN	1	This signal is asserted once the comparison between the calculated checksum and received checksum is made. It is valid for one clock cycle.
4	ipchksum_result_bit	IN	1	If high, it indicates that the received checksum matches with the calculated checksum, else a mismatch. This result is stored in the SIP Data Structure.

Table 5.25: *Interface with the TCP Checksum Calculator*

No.	Pin Name	Dirn.	Width	Description
1	tcpchksm_write	IN	1	An indication that the current byte is a valid TCP byte. The Buffer Write Controller samples this valid and transfers this byte to the SIP data structure.
2	tcpchksm_byte	IN	8	The byte to be writtn to the SIP data structure.
3	tcpchksm_result_valid	IN	1	This signal is asserted once the comparison between the calculated checksum and received checksum is made. Note that the TCP checksum is calculated over the TCP header, the pseudoheader and the payload. This signal is asserted for one clock cycle.
4	tcpchksm_result_bit	IN	1	This bit is looked at when ipchksum_result_valid is asserted. If high, it indicates that the received checksum matches with the calculated checksum. If low, it indicates a mismatch in the two checksums. This result is stored in the SIP Data Structure and later accessed by the CPU.
5	tcpchksm_last_tcphdr_byte	IN	1	An indication that the current byte is the last TCP header byte. This byte is used by the Buffer Write Controller to increment internal memory indices.

Table 5.26: *Interface with the SIP Byte Processor*

No.	Pin Name	Dirn.	Width	Description
1	sipproc_fldval_write	IN	1	An indication for the Buffer Write Controller to write the current byte into the SIP Data Structure. This byte is part of a field value. This write indication is one clock pulse wide.
2	sipproc_fldval_byte	IN	8	The field value byte to be written to the SIP Data Structure.
3	sipproc_hdr_done	IN	1	An indication that the SIP Byte processor has finished processing a header name,value pair. This indication is used to determine the offset from which the next header value will be stored.
4	sipproc_match_found	IN	1	An indication that the SIP Byte processor has found a match. Looking at this indication, the Buffer Write Controller calculates the offset pointer for that field name and stores it in the SIP Data Structure.
5	sipproc_match_id	IN	5	The unique match ID identifying which field name was correctly matched. The Buffer Write Controller uses this ID as a memory index to the SIP Data Strcuture, to store the pointer offset. It also sets the corresponding bit in the 44-bit Field name found register.
6	sipproc_match_bytes	IN	9	The number of bytes in the field value. This value is used by the Buffer Write Controller to calculate the pointer offset for the next field value.

Table 5.27: *Interface with Memory 0*

No.	Pin Name	Dirn.	Width	Description
1	bufwrctr_mem0_wr_en	OUT	1	The write enable for Memory 0.
2	bufwrctr_mem0_wr_data	OUT	32	The data to be written to Memory 0. Data would be written by the Buffer Write Controller only. The external requestor would never write data.
3	bufwrctr_mem0_address	OUT	9	The address to perform the read/write operation from on Memory 0.
4	mem0_rd_data	IN	32	The 32-bit data read from Memory 0. Note that only the external requestor would wish to read data.

Table 5.28: *Interface with Memory 1*

No.	Pin Name	Dirn.	Width	Description
1	bufwrctr_mem1_wr_en	OUT	1	The write enable for Memory 1.
2	bufwrctr_mem1_wr_data	OUT	32	The data to be written to Memory 1. Data would be written by the Buffer Write Controller only. The external requestor would never write data.
3	bufwrctr_mem1_address	OUT	9	The address to perform the read/write operation from on Memory 1.
4	mem1_rd_data	IN	32	The 32-bit data read from Memory 1. Note that only the external requestor would wish to read data.

Table 5.29: *Interface with the external read requestor*

No.	Pin Name	Dirn.	Width	Description
1	bufwrctr_structure_ready	OUT	1	an indication to the external requestor that the SIP Data Structure is ready to be read out.
2	external_rd_enable	IN	1	The external read request.
3	external_rd_address	IN	9	The address to perform the read operation from.
4	bufwrctr_rd_data	OUT	32	The 32-bit data read from Memory 0/1.
5	bufwrctr_data_valid	OUT	1	The validator for the read data.

5.5.2 Architecture

We will start this section by discussing the address partitioning of the SIP Data Structure. We will then proceed to elaborate on the interface timing between the Buffer Write Controller and the SIP Data Structure. This would be followed by the hardware implementation of the blocks in the logic schematic shown in 4.8.

- SIP Data Structure Address Partitioning

  Data bytes received from the IP Checksum calculator, TCP checksum calculator and SIP processor are written in designated spaces in the SIP Data structure, as shown in Figure 5.44. For the TCP and IP header bytes, the space allocated is sufficient to store the maximum number of bytes that could be expected.

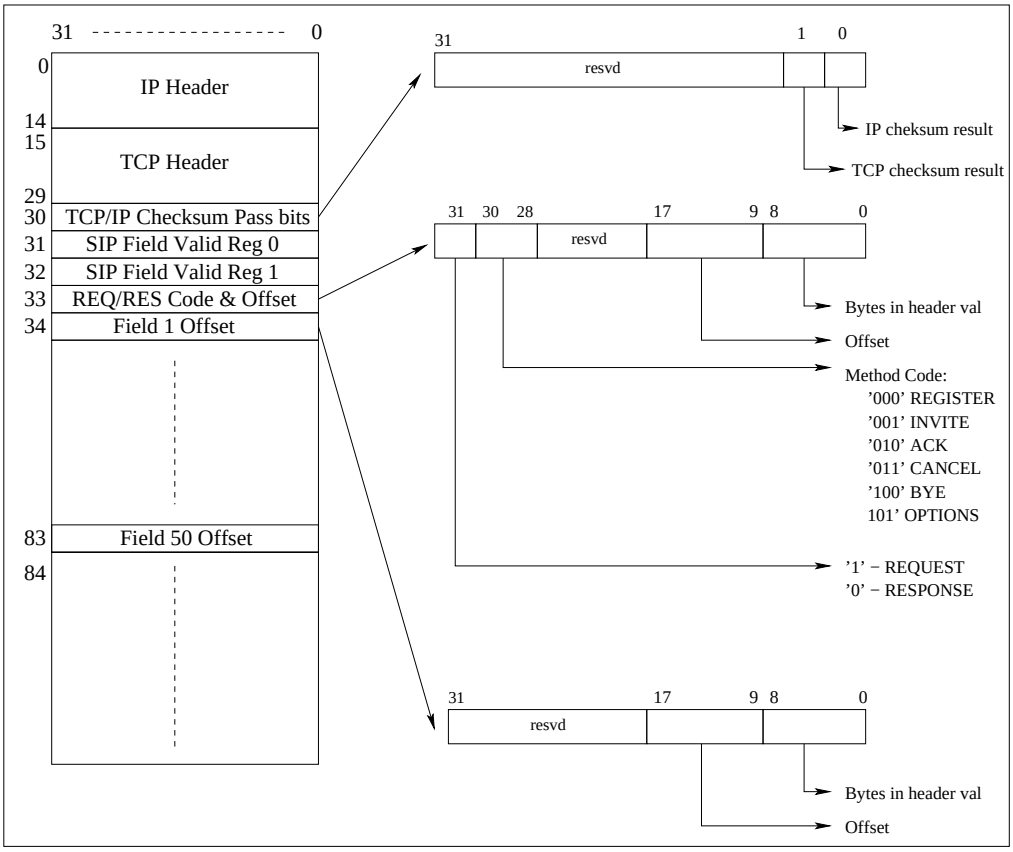

Figure 5.44: Address space partitioning for the SIP Data Structure

- Timing Interface with SIP Data Structure

  The timing diagram in Figure 5.45 depicts the manner in which the input bytes are accumulated to form a 32-bit word and written to memory.

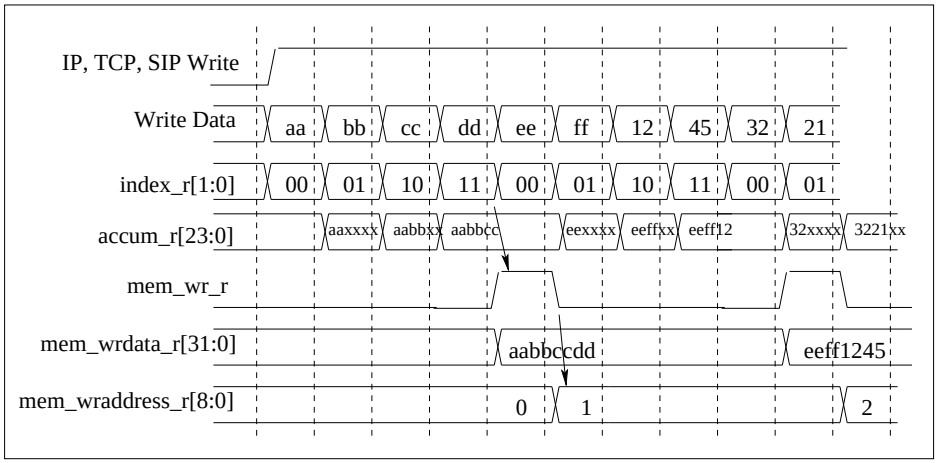

Figure 5.45: Timing Interface with the SIP Data Structure

- Schematic for the Read/Write Interfaces

  There are two SIP data structures, Memory-0 and Memory-1, operating in a ping-pong manner. While one structure is being written to, the other one is being read out. Once the current packet is completely processed, i.e the SIP fields are extracted and stored, a memory switch occurs, flagged by the toggling of *mem_select_r*. Once this toggle occurs, the memory which was just written into is now read out, while the other memory is now written into. This implementation is shown in Figure 5.46.

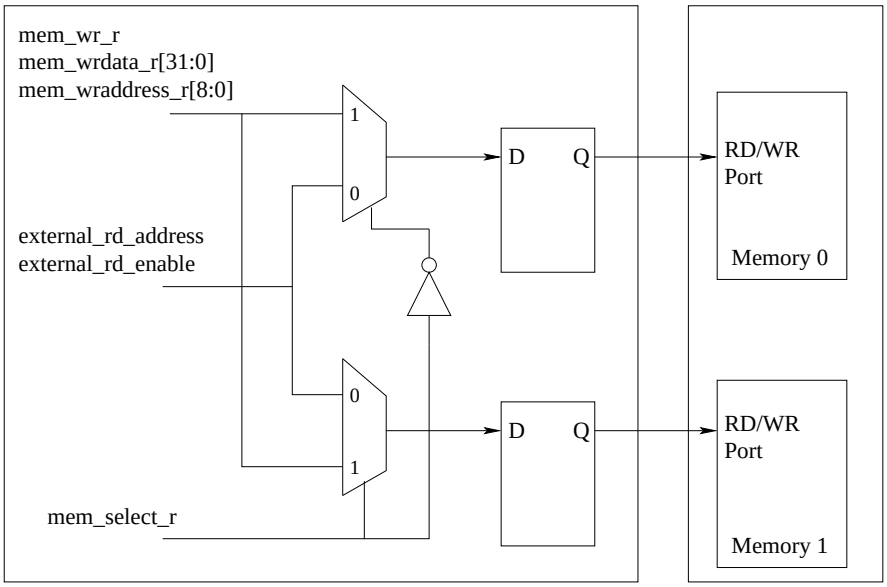

Figure 5.46: Memory Schematic

- Generation of *mem_select_r* This is a flag which has to toggle every time a packet has been processed. We can use the signal *tcpchksm_result_valid* to effect this toggle, as it occurs at the end of every packet.

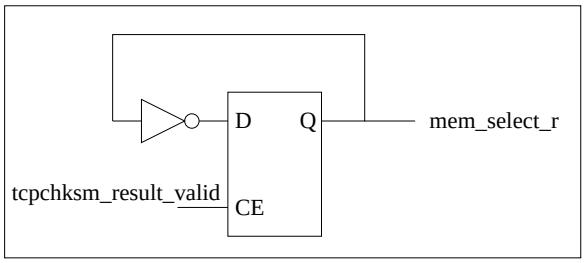

Figure 5.47: Generation of the memory select flag

- Generation of the Read/Write Interfaces.

  The Buffer Write Controller writes to Memory 0/1 based on various stimuli

from the IP Checksum Calculator, TCP Checksum Calculator and SIP Byte Processor. Based on the source of the stimulus, it generates a write address and write data. It relies on the fact these stimuli occur at unique moments and in a fixed sequence. Table 5.30 shows the sequence of the write stimulus and the address, data being generated upon that stimulus. Additional action to be taken upon receipt of that stimulus is also noted.

Table 5.30: *Table of sequence of write stimulus*

Stimulus	**Actions**	**Address**	**Data**
ipchksum_write	Increment word index, increment mem index (4 writes)	0 + mem index	accum_r
ipchksum_result_valid	clear mem index, clear word index, store result in chksum_result_r	Dont write	X
tcpchksm_write	increment word index, increment mem index (4 writes)	15 + mem index	accum_r
tcpchksm_last_tcphdr byte	Clear mem index, clear word index, clear siphdr memoffst	Don't write	X
sipproc_fldval_write	increment word index, increment mem index (4 writes)	79 + mem index	accum_r
sipproc_hdr_done	update sip fieldreg 0/1, siphdr memoffst = mem index	35 + sipproc matchid	{siphdr mem-offst,sipproc match bytes}
tcpchksm_result_valid	clear mem index, clear word index, toggle mem select	30, 31, 32	chksum result, sip fieldreg0, sip fieldreg1

Chapter 6

Software Simulation and Verification

This chapter starts with the description of the test enviroment developed, the components that make the testbench and the the verification strategy.

6.1 Test Environment Description

- Block Diagram

  Figure 6.1 shows the architectural block diagram of the testbench.

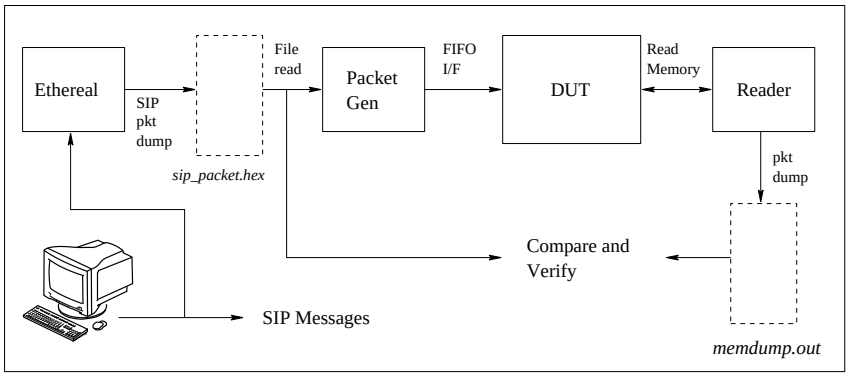

Figure 6.1: Timing relation with Delineator Interface

- Inputs

  A black-box approach is taken to verify the Device-Under-Test (DUT). We use a softphone called Express Talk v. 1.04 [15] to initiate SIP calls. A popular network protocol analyzer, *Ethereal* [16] is used to capture the outgoing SIP messages and dump it to a hex file *sip_packet.hex*. The contents of this packets are input to the DUT byte-by-byte.

- Outputs

  The DUT processes the SIP byte stream and populates the SIP Data Structure. After this, it asserts a "done" indication to the external world, indicating that the structure is ready to be read out. The testbench reads the structure and outputs a text file *memdump.out*. The contents of this outfile are formatted and can be examine for correctness of the DUT operation by comparing it with the contents of *sip_packet.hex*

- Components

  The main components of the testbench are briefly described below:

  1. Packet Generator

     The *Packet Generator* loads all the bytes in the input file *sip_packet.hex* into a buffer. It emulates a FIFO interface to input these bytes to the DUT.

  2. Reader

     The *Reader* reads out all locations of the SIP Data Structure, formats the data to increase readability and writes to the file *memdump.out*.

6.2 Testbench Architecture

This section describes the components of the testbench in more detail.

6.2.1 Module Packet Generator (*pktgen*)

- Pin Interface

Table 6.1: *Interface with the system*

No.	Pin Name	Dirn.	Width	Description
1	reset	IN	1	The async system reset.
2	clk	IN	1	The system clock.

Table 6.2: *Interface with the DUT*

No.	Pin Name	Dirn.	Width	Description
1	fifo_empty	OUT	1	Indicates to the DUT the presence of data in the FIFO
2	fifo_data	OUT	9	The data delivered to the DUT.
3	delin_fifo_read	IN	1	An indication that the DUT has accepted the FIFO data.

- Architecture

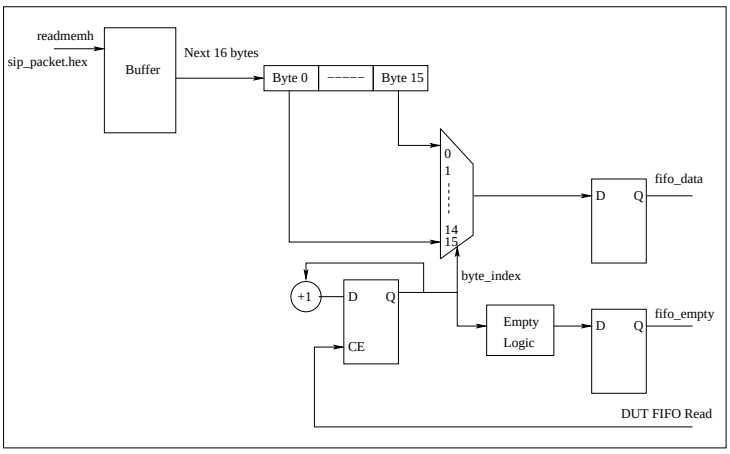

Figure 6.2: Architectural Block Diagram - Packet Generator Module

The Packet Generator reads in all the data from the input file and stores it in a buffer. The buffer width is equal to the number of bytes Ethereal outputs in one line, which in our test enviroment is 16 bytes.

The pin *fifo_empty* is then de-asserted to make data available to the DUT. When it samples the DUT's *delin_fifo_read* as asserted, the byte pointed by an index counter *byte_index_r* is output on *fifo_data[8:0]*. The index is then incremented to point to the next byte in one 16-byte buffer location. Once all 16 bytes are comsumed by the DUT, the next location(16-bytes) is read out and another 16 bytes are ready to be delivered. This happens transparently to the DUT, giving an effect of a continous byte-stream. The start of packet indication, represented by the most significant bit of *fifo_data[8:0]*, is set each time the very first byte of a SIP message is delivered to the DUT.

6.2.2 Module Reader (*reader*)

- Pin Interface

Table 6.3: *Interface with the system*

No.	Pin Name	Dirn.	Width	Description
1	reset	IN	1	The async system reset.
2	clk	IN	1	The system clock.

Table 6.4: *Interface with the DUT*

No.	Pin Name	Dirn.	Width	Description
1	bufwrctr_structure_ready	IN	1	An indication from the DUT that the SIP Data Structure is ready to be read out.
2	external_rd_enable	OUT	1	The read request.
3	external_rd_address	OUT	9	The address to perform the read operation from.
4	bufwrctr_rd_data	OUT	32	The 32-bit data read from the SIP Data Structure.
5	bufwrctr_data_valid	OUT	1	The validator for the read data.

- Architecture

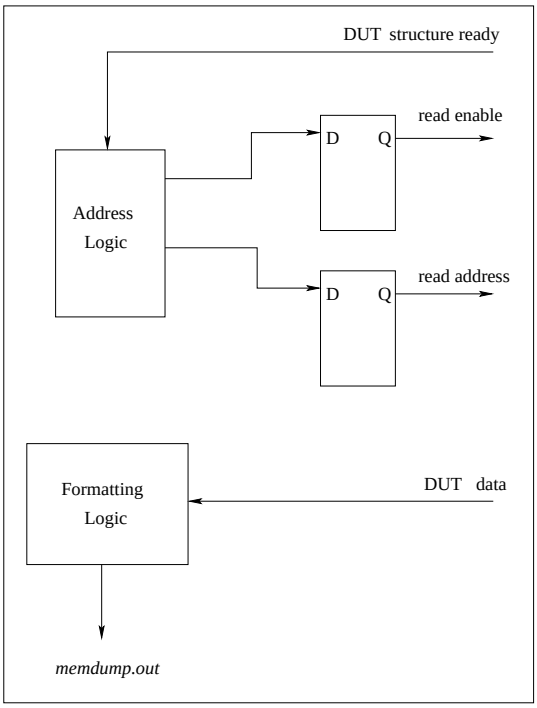

Figure 6.3: Architectural Block Diagram - Reader Module

The *reader* has to wait for the DUT structure ready indication before it can start to read the SIP Data Structure. Once this handshake is asserted, it starts reading every location, formats the data received based on the address and writes it to the file *memdump.out*. This file is examined against the original input file to verify DUT functionality. The contents of this file are written out on an address based field-wise manner, and not as a series of hex bytes. This increases readability and minimizes the time needed to examine the file.

6.3 Verification Test Plan

This section describes the verification plan. The goal is to test the DUT using a set of input script files. Each version of the input file *sip_packet.hex* represents a test case/script.

The DUT can be divided into a set of features. Typically, one input script would test some/all of these features, depending on the contents of the input file. These features, in no particular order, can be listed as:

1. Extraction of TCP/IP header/datagram length fields
2. Delineation of bytes into IP/TCP/SIP bytes
3. IP checksum calculation
4. TCP checksum calculation
5. Writing TCP/IP header bytes in SIP Data Structure
6. Request/Response Identification
7. Pattern matching for message header field names
8. Writing SIP header field values in SIP Data Structure
9. Populating the SIP Field Valid Registers
10. Memory offset calculation for extraction of header field values

6.3.1 Feature Tests

Feature tests are those which test some/all features of the DUT. The scripts presented to the DUT test all the features listed. These tests are run with one SIP message contained in one script.

Table 6.5: *Table for feature tests*

No.	**Script Name**	**Features covered from list**
1	*sip_packet1.hex*	1, 2, 3, 4, 5, 6, 7, 8, 9, 10

6.4 Test Results Summary

Table 6.6 summarizes the tests run on a PASS/FAIL basis.

Table 6.6: *Test result summary table*

No.	**Category**	**Script Name**	**Result/Comments**
1	Feature tests	*sip_packet1.hex*	PASS

Chapter 7

FPGA Implementation

This chapter tells us the FPGA device used and its utilization summary.

7.1 Choosing the Device

The resource utilization before implementation was estimated at 1100 slices with a 57 I/O pin requirement . Based on this the Virtex device Xilinx xcv100 containing 1200 slices with 98 I/O pins was chosen for implementation.

7.2 Device Utilization Summary

Table 7.1 summarizes the device resource utilization for the xcv100. Please refer to the appendix for detailed reports on the synthesis, map, implementation and timing.

Table 7.1: *Device Utilization Summary*

Resource	**Occupied**	**Out of**	**Percentage**
Slices	916	1,200	76%
Pads (IOB's)	57	98	58%
Block RAM's	8	10	80%
FF's	694	2,400	28%
4 input LUT's	1,493	2,400	62%

Additional design details:

1. Clock Period: 15.899ns
2. Maximum operating frequency: 62.8 MHz
3. Total equivalent gate count for design: 146,394
4. Total data throughput supported: 502.4 Mbps

Chapter 8

Future Work and Conclusion

This chapter describes the future scope of the work presented in terms of feature additions and optimizations to the existing design.

8.1 Architecture Optimizations

8.2 FPGA Implementation Optimizations

The FPGA optimizations can be categorized into *area* and *timing* optimizations. These are discussed in the following sections.

8.2.1 Area Optimizations

The Xilinx FPGA provides the user with many primitives which could be used to reduce the area. The building block of an FPGA is the Configurable Logic Block (CLB) [17]. Each CLB is made up of 4 Look-Up Tables (LUT's) and 4 D-FF storage elements. The LUT is a 16x1 RAM whose essential function is to implement a 4-input function. Optimizations can be done when the LUT is used as a register array or as a shift register, explained below.

- LUT as a register array.

Suppose we need a 16-bit register array. If we directly code it as a 16-bit register array, we would be occupying 4 CLB's, given 4 D-FF's per CLB. Now we can instantiate an LUT as a register array, in which case we use only LUT. This results in using only 0.25 of a CLB.

- LUT as a shift register.

  In a pipleined design, it is often required to register data multiple times to account for the pipeline latency. This would involve a certain number of D-FF's, depending on the datawidth and the number of cycles we need to register it for. The LUT could be used as a shift register with programmable delay from 1 - 16 clocks. For example, if we need to register a 4-bit data for 10 clocks, we would need 40 D-FF, which would utilize 10 CLB's. If we use the LUT as a shift register, we would need only 4 LUT's (1 CLB).

8.2.2 Timing Optimizations

Timing optimizations can be done using methods like pipelining and parallelism. Specific to FPGA architecture, we can use the FPGA Floorplanner to place communicating modules close to each other. This reduces the routing delay, which results in a better timing.

8.3 Feature Additions

This section discusses the features that could be added to the existing design. These are listed below.

1. Within a header field value, we could also extract the <parameter_name = parameter_value> pair.

2. We could implement a set of rules for the message. For example, the 'TO' field MUST be present in an INVITE message.

3. We could support the abbreviated form of header field names. For ex. the standard allows the use of the character 't' in place of 'TO'.

4. We could support occurence of the same header name multiple times within the same message.

   For Example, consider the following fragment of a SIP message:

   Route: sip:alice@atlanta.com
   Subject: Lunch
   Route: sip:bob@biloxi.com
   Route: sip:carol@chicago.com

   The above message fragment has multiple "Route" specifications. They could be processed and stored with an indication in the data structure that multiple header field values exist for the "Route" header.

5. Error Recovery. We could implement recovery strategies from erroneous conditions like incomplete packets.

8.4 Verification

The current test environment could be modified to support enhanced test capabilities like :

- Use assertion based verification. The current test strategy allows the entire SIP message to be processed and then verifies correctness by examining the content of the SIP data structure. Instead, if assertions could be used in the code itself, we could catch any bugs earlier in the testing phase.

- Regression mode. In this test mode, SIP messages could be randomly generated and sent to the SIP processor

- Create error conditions. Various error conditions could be created via the SIP message to verify the DUT functionality. For example, an error could be introduced in the TCP/IP checksum to test if the SIP processor correctly flags this error.

8.5 Conclusion

It can be concluded that it is possible to offload SIP message processing onto hardware. The functional verification proves the correctness of the design.

Bibliography

[1] J. Ott, "A short history of sip," Marcus Evans SIP Conference, Geneva, Switzerland, Tech. Rep., 2001.

[2] *Session Initiation Protocol.* Request for Comments 3261, 2002.

[3] "Cisco 7900 series unified ip phones," Cisco Systems, Inc., Tech. Rep. [Online]. Available: http://www.cisco.com/en/US/products/hw/phones/ps379/index.html

[4] "Windows messenger 4.7." [Online]. Available: http://www.microsoft.com/windows/messenger/

[5] *Session Initiation Protocol (SIP) Public Switched Telephone Network (PSTN) Call Flows.* Request for Comments 3666, 2003.

[6] K. Singh and H. Schulzrinne, "Interworking between sip/sdp and h.323," *Columbia University Technical Report*, 2000.

[7] K. Singh, G. Nair, and H. Schulzrinne, "Centralized conferencing using sip," *IPTel'2001*, 2001.

[8] A. Currid, "Tcp offload to the rescue," *ACM Queue*, 2004.

[9] *Transmission Control Protocol.* Request for Comments 793, 1981.

[10] "Time for toe," Chelsio Communications, Tech. Rep.

[11] *Augmented BNF for Syntax Specifications : ABNF.* Request for Comments 2234, 1997.

[12] *Internet Message Format.* Request for Comments 2822, 2001.

[13] *UTF-8, a transformation format of ISO 10646.* Request for Comments 2279, 1998.

[14] M. J. C. Alfred V. Aho, "Efficient string matching: an aid to bibliographic search," *Communications of the ACM*, pp. 333–340, 1975.

[15] "Express talk voip softphone for windows." [Online]. Available: http://www.nch.com.au/talk

[16] "Ethereal: A network protocol analyzer." [Online]. Available: http://www.ethereal.com

[17] *Virtex-2 Platform FPGA User Guide*, 2005.

APPENDIX

Appendix A

Sample Input File

This appendix list the input file *invite.hex* which was used to test the design. In each line, to the left are the bytes that are input to the design, to the right are their ASCII characters.

00000c07ac000016ce0091f408004500 //.....E.
030501a400008011c1a79807cc0cc325 //.....
4d6313ce13c402f116f4494e56495445 //Mc.....INVITE
207369703a72616a6140697074656c2e //sip:raja@iptel.
6f7267205349502f322e300d0a566961 //orgSIP/2.0..Via
3a205349502f322e302f554450203135 //:SIP/2.0/UDP15
322e372e3230342e31323a353037303b //2.7.204.12:5070;
72706f72743b6272616e63683d7a3968 //rport;branch=z9h
4734624b30333734340d0a546f3a203c //G4bK03744..To:¡
7369703a72616a6140697074656c2e6f //sip:raja@iptel.o
72673e0d0a46726f6d3a202222203c73 //rg¿..From:""¡s
69703a403135322e372e3230342e3132 //ip:@152.7.204.12
3a353037303e3b7461673d343637310d //:5070¿;tag=4671.
0a43616c6c2d49443a20313134363433 //.Call-ID:114643
303436322d333734342d4c454e4f564f //0462-3744-LENOVO
2d3243343846433334403135322e372e //-2C48FC34@152.7.

3230342e31320d0a435365713a203334 //204.12..CSeq:34
3420494e564954450d0a4d61782d466f //4INVITE..Max-Fo
7277617264733a2032300d0a55736572 //rwards:20..User
2d4167656e743a204578707265737320 //-Agent:Express
54616c6b20312e30340d0a436f6e7461 //Talk1.04..Conta
63743a203c7369703a403135322e372e //ct:¡sip:@152.7.
3230342e31323a353037303e0d0a416c //204.12:5070¿..Al
6c6f773a20494e564954452c2041434b //low:INVITE,ACK
2c2043414e43454c2c204f5054494f4e //,CANCEL,OPTION
532c204259452c20494e464f2c205245 //S,BYE,INFO,RE
4645522c204e4f544946590d0a537570 //FER,NOTIFY..Sup
706f727465643a207265706c61636573 //ported:replaces
0d0a436f6e74656e742d547970653a20 //..Content-Type:
6170706c69636174696f6e2f7364700d //application/sdp.
0a436f6e74656e742d4c656e6774683a //.Content-Length:
203238310d0a0d0a763d300d0a6f3d2d //281....v=0..o=-
203831333037313239392038313330 37 //81307129981307
3132303120494e20495034203135322e //1201INIP4152.
372e3230342e31320d0a733d45787072 //7.204.12..s=Expr
6573732054616c6b0d0a633d494e2049 //essTalk..c=INI
6d61703a302050434d552f383030300d //map:0PCMU/8000.
0a613d7274706d61703a382050434d41 //.a=rtpmap:8PCMA
2f383030300d0a613d7274706d61703a ///8000..a=rtpmap:
332047534d2f383030300d0a613d7274 //3GSM/8000..a=rt
706d61703a313320434e2f383030300d //pmap:13CN/8000.
0a613d7274706d61703a313031207465 //.a=rtpmap:101te
6c6570686f6e652d6576656e742f3830 //lephone-event/80
30300d0a613d666d74703a3130312030 //00..a=fmtp:1010
2d31360d0a613d73656e64726563760d //-16..a=sendrecv.
0a0d0a ...

Appendix B

Sample Output File

Shown below is an extract from the output file *memdump.out*. Only certain relevant parts of the original file are shown to limit its length. Here we can clearly see how the SIP data structure looks like once the message has been processed.

IP HEADER BYTES

IP Header @ Address: 0 ; Bytes: 45 00 03 05
IP Header @ Address: 1 ; Bytes: 01 a4 00 00
IP Header @ Address: 2 ; Bytes: 80 11 c1 a7
IP Header @ Address: 3 ; Bytes: 98 07 cc 0c
IP Header @ Address: 4 ; Bytes: c3 25 4d 63

UDP HEADER BYTES

TCP Header @ Address: 15 ; Bytes: 13 ce 13 c4
TCP Header @ Address: 16 ; Bytes: 02 f1 16 f4

CHECKSUM RESULTS

IP checksum address @ : 30 ; Bit 0; RESULT = PASS
IP checksum @ address : 30 ; Bit 1; RESULT = PASS

HEADER FIELDS IN MESSAGE

Field 5 Extracted : Allow
Field 10 Extracted : Call-ID
Field 12 Extracted : Contact
Field 16 Extracted : Content-Length
Field 17 Extracted : Content-Type
Field 18 Extracted : CSeq
Field 22 Extracted : From
Field 24 Extracted : Invite
Field 25 Extracted : Max-Forwards
Field 10 Extracted : Supported
Field 12 Extracted : To
Field 14 Extracted : User-Agent

STARTLINE

Startline is a Request Line
INVITE Method extracted
Read from address 84 ; Number of bytes to read: 26

FIELD OFFSETS

Field 5 : Read from address 146 ; Number of bytes to read: 54
Field 10 : Read from address 120 ; Number of bytes to read: 44
Field 12 : Read from address 139 ; Number of bytes to read: 24
Field 16 : Read from address 165 ; Number of bytes to read: 3

Field 17 : Read from address 162 ; Number of bytes to read: 15
Field 18 : Read from address 131 ; Number of bytes to read: 10
Field 22 : Read from address 111 ; Number of bytes to read: 36
Field 25 : Read from address 133 ; Number of bytes to read: 2
Field 42 : Read from address 159 ; Number of bytes to read: 8
Field 44 : Read from address 105 ; Number of bytes to read: 20
Field 46 : Read from address 134 ; Number of bytes to read: 17
Field 47 : Read from address 91 ; Number of bytes to read: 55

SIP HEADER BYTES

SIP Header Bytes @ Address: 84 ; Bytes: s i p :
SIP Header Bytes @ Address: 85 ; Bytes: r a j a
SIP Header Bytes @ Address: 86 ; Bytes: @ i p t
SIP Header Bytes @ Address: 87 ; Bytes: e l . o
SIP Header Bytes @ Address: 88 ; Bytes: r g S
SIP Header Bytes @ Address: 89 ; Bytes: I P / 2
SIP Header Bytes @ Address: 90 ; Bytes: . 0 / 2
SIP Header Bytes @ Address: 91 ; Bytes: S I P /
SIP Header Bytes @ Address: 92 ; Bytes: 2 . 0 /
SIP Header Bytes @ Address: 93 ; Bytes: U D P
SIP Header Bytes @ Address: 94 ; Bytes: 1 5 2 .
SIP Header Bytes @ Address: 95 ; Bytes: 7 . 2 0
SIP Header Bytes @ Address: 96 ; Bytes: 4 . 1 2
SIP Header Bytes @ Address: 97 ; Bytes: : 5 0 7
SIP Header Bytes @ Address: 98 ; Bytes: 0 ; r p
SIP Header Bytes @ Address: 99 ; Bytes: o r t ;
SIP Header Bytes @ Address:100 ; Bytes: b r a n
SIP Header Bytes @ Address:101 ; Bytes: c h = z
SIP Header Bytes @ Address:102 ; Bytes: 9 h G 4

SIP Header Bytes @ Address:103 ; Bytes: b K 0 3
SIP Header Bytes @ Address:104 ; Bytes: 7 4 4 3
SIP Header Bytes @ Address:105 ; Bytes: ¡ s i p
SIP Header Bytes @ Address:106 ; Bytes: : r a j
SIP Header Bytes @ Address:107 ; Bytes: a @ i p
SIP Header Bytes @ Address:108 ; Bytes: t e l .
SIP Header Bytes @ Address:109 ; Bytes: o r g ¿
SIP Header Bytes @ Address:110 ; Bytes: ” ” ¡
SIP Header Bytes @ Address:111 ; Bytes: s i p :
SIP Header Bytes @ Address:112 ; Bytes: @ 1 5 2
SIP Header Bytes @ Address:113 ; Bytes: . 7 . 2
SIP Header Bytes @ Address:114 ; Bytes: 0 4 . 1
SIP Header Bytes @ Address:115 ; Bytes: 2 : 5 0
SIP Header Bytes @ Address:116 ; Bytes: 7 0 ¿ ;
SIP Header Bytes @ Address:117 ; Bytes: t a g =
SIP Header Bytes @ Address:118 ; Bytes: 4 6 7 1
SIP Header Bytes @ Address:119 ; Bytes: 1 1 4 6
SIP Header Bytes @ Address:120 ; Bytes: 4 3 0 4
SIP Header Bytes @ Address:121 ; Bytes: 6 2 - 3
SIP Header Bytes @ Address:122 ; Bytes: 7 4 4 -
SIP Header Bytes @ Address:123 ; Bytes: L E N O
SIP Header Bytes @ Address:124 ; Bytes: V O - 2
SIP Header Bytes @ Address:125 ; Bytes: C 4 8 F
SIP Header Bytes @ Address:126 ; Bytes: C 3 4 @
SIP Header Bytes @ Address:127 ; Bytes: 1 5 2 .
SIP Header Bytes @ Address:128 ; Bytes: 7 . 2 0
SIP Header Bytes @ Address:129 ; Bytes: 4 . 1 2
SIP Header Bytes @ Address:130 ; Bytes: 3 4 4
SIP Header Bytes @ Address:131 ; Bytes: I N V I
SIP Header Bytes @ Address:132 ; Bytes: T E V I
SIP Header Bytes @ Address:133 ; Bytes: 2 0 V I

SIP Header Bytes @ Address:134 ; Bytes: E x p r
SIP Header Bytes @ Address:135 ; Bytes: e s s
SIP Header Bytes @ Address:136 ; Bytes: T a l k
SIP Header Bytes @ Address:137 ; Bytes: 1 . 0
SIP Header Bytes @ Address:138 ; Bytes: 4 1 . 0
SIP Header Bytes @ Address:139 ; Bytes: ¡ s i p
SIP Header Bytes @ Address:140 ; Bytes: : @ 1 5
SIP Header Bytes @ Address:141 ; Bytes: 2 . 7 .
SIP Header Bytes @ Address:142 ; Bytes: 2 0 4 .
SIP Header Bytes @ Address:143 ; Bytes: 1 2 : 5
SIP Header Bytes @ Address:144 ; Bytes: 0 7 0 ¿
SIP Header Bytes @ Address:145 ; Bytes: I N V I
SIP Header Bytes @ Address:146 ; Bytes: T E ,
SIP Header Bytes @ Address:147 ; Bytes: A C K ,
SIP Header Bytes @ Address:148 ; Bytes: C A N
SIP Header Bytes @ Address:149 ; Bytes: C E L ,
SIP Header Bytes @ Address:150 ; Bytes: O P T
SIP Header Bytes @ Address:151 ; Bytes: I O N S
SIP Header Bytes @ Address:152 ; Bytes: , B Y
SIP Header Bytes @ Address:153 ; Bytes: E , I
SIP Header Bytes @ Address:154 ; Bytes: N F O ,
SIP Header Bytes @ Address:155 ; Bytes: R E F
SIP Header Bytes @ Address:156 ; Bytes: E R ,
SIP Header Bytes @ Address:157 ; Bytes: N O T I
SIP Header Bytes @ Address:158 ; Bytes: F Y T I
SIP Header Bytes @ Address:159 ; Bytes: r e p l
SIP Header Bytes @ Address:160 ; Bytes: a c e s
SIP Header Bytes @ Address:161 ; Bytes: a p p l
SIP Header Bytes @ Address:162 ; Bytes: i c a t