DESIGNING TCP/IP CHECKSUM FUNCTION FOR ACCELERATION IN FPGA

Etim B. Eyo and Thomas A. Nwodoh
EMAIL: etimeyo10@yahoo.com, nwodoh@digitechcorp.com
PH: +234-703-591-1154, +234-803-553-4359
Department of Electronic Engineering
University of Nigeria, Nsukka

ABSTRACT

Over the years network transmission speeds have improved greatly without a corresponding increase in the processing speed of the host processor. Traditionally, network protocol processing is handled in the CPU of the host computer. However, with devices featuring advanced connectivity and Internet functionality, protocol processing has created a heavy workload on the general processing processors, with additional constraints by the slower I/O bus speed limits. Consequently, for a higher throughput and speedy delivery of information between hosts on the internet, there is the need to identify those performance-critical TCP/IP functions and accelerate them in order to match the transmission speeds with the protocol processing speeds. Based on profiling results, a micro-level function, namely checksum is observed to be a computational intensive function. In this paper, the checksum function is selected and implemented in an FPGA. The checksum calculation is implemented based on 16-bit one's complement adders. In all, by minimizing the functional overhead, such as, instruction fetching and decoding, bus speed constraints, latency due to buffer/memory transfer; and providing flexibility by configuration possibilities, the high speed and cost advantage are made possible.

KEYWORDS: FPGA, TCP/IP, CPU, Checksum, VHDL, Timing simulation, Throughput.

INTRODUCTION

Advanced services such as multimedia delivery and voice over IP have invoked an explosive increase of the data transmitted on the Internet. Adequate network bandwidths are required to guarantee the high quality of transmissions. This has been solved by the wide deployment of optical fibers over the Internet, which provides sufficient bandwidth for transmission.

The development speed of network transmission rates dramatically outpaces the development speed of the general-purpose processors and the I/O bus speeds. The network transmission speeds are reaching 10 Gbps (OC-192), 40 Gbps (OC-768) and heading towards 80Gbps. Hence, the bottleneck of the network speed has shifted to the network processing, especially the data processing within TCP/IP domain and the relatively slower bus speeds. In Zhonghe [2], it is revealed that, we will need 100% efficiency from Pentium III, 1-GHz processor or 30% efficiency from a Pentium 4, 2.4 GHz processor to process the 1-Gbps TCP/IP protocol. A generally accepted rule of thumb is that 1 hertz of CPU processing is required to send or receive 1 bit/s of TCP/IP. For example 5 Gbit/s of network traffic requires 5 GHz of CPU Processing. This implies that 2 entire cores of a 2.5 GHz multi-core processor will be required to handle the TCP/IP processing associated with 5 Gbit/s of TCP/IP traffic. Since Ethernet (10Ge) is bidirectional it is possible to send and receive 10 Gbit/s (for an aggregate throughput of 20 Gbit/s). Using the 1 Hz/ bit rule this equates to 8 of 2.5 GHz cores. Few if any current day end systems and servers have a requirement to move 10 Gbit/s in both directions [3].

The performance of the TCP/IP processing functions also depends on memory (memory access times) and data volume. It would need lots of memory access to read

data when computing checksums, which leads to an efficiency bottleneck. Therefore, it is necessary to accelerate network processing capability and reduce the CPU load by adding extra hardware.

Two challenges are observed:
The first challenge, the "**Need for Speed**," requires to design the time-critical TCP/IP processing function in FPGA, and to achieve the processing speed-up as much as possible.

The second challenge, the "**Need for Flexibility**," leads to the design of flexible functions in FPGA that can be updated whenever the protocol or standard changes or new features are introduced.

NETWORK DATA PROCESSING

Data must be processed at every layer it goes through while it is transmitted over the Internet. Different functions running on different layers takes charge of different kind of processing tasks. Generally, the TCP/IP functions can be divided into two parts [1]:

- Data plane, which refers to where network data passes through. Data plane functions, such as classification, table lookup, are performed over every packet passing through the system. Therefore, data plane has a large amount of data processing tasks. Since data are transmitted at high speeds, these processing tasks are also required to perform at high speeds. Most computational intensive operations are also performed on the data plane.
- Control plane, which takes charge of control and management tasks that coordinates the functions between the data plane and control plane within the system, and with the outside systems. It updates the tables on the data plane, performs signaling, interface management, and other complex actions that cannot be executed on the data plane.

Micro-Level Functions

Since the performance-critical operations are located in the data plane and are required to perform at transmission speeds, this paper focuses mainly on the functions on the data plane and refers to them as **micro-level functions**. The followings are typical micro-level functions: Encapsulation, Checksum, Data Parsing, Data Modification, Bitwise Comparison, Queuing/Scheduling, Cyclic Redundancy Checks, Fragmentation, Table Lookup and Classifications; see [1] for details of these micro-level functions. Furthermore, there are many network services running on the data plane which may consist of several micro-level functions. Some typical network services are: Address Resolution Protocol (ARP), Internet Protocol Security (IPSec), Firewall, and Network Address Translation (NAT) [1].

OBJECTIVE OF THE RESEARCH

The objective of this paper is to design the TCP/IP function (in this case, checksum) in a way that it can meet the discussed challenges. Specifically, the paper will:

- investigate the TCP/IP functions.
- design the performance critical micro-level function using FPGA.
- perform simulation to gain insight into the performance potential of the design.

LITERATURE REVIEW

From review of relevant literatures, the checksum calculation is observed to be one of the most computing intensive and time-critical functions that may impede the processing speeds of general purpose processors (GPPs). Checksum calculation is usually on the IP header, ICMP's entire message (including header and data), TCP and UDP's entire message.

Kay et al [8] categorized the processing overheads into several operations, including checksum, data move, data structure, errorcheck, mbuf, opsys, protspec, and others. They showed that checksum calculation is the main processing overhead, and the overhead grows as the size of the message increases.

In Tsai et al [10], a profile of the IPv4 forwarding on the Intel IXP1200 network processor with the 64-byte packets is illustrated. The Header Validation occupies

the highest processing time. The header validation mainly performs the following operations:

- Check version in the version field
- Check the header length field
- Calculate the header checksum

In these three steps, the checksum calculation is the most time consuming operation.

In the profiling of IPv4 packet forwarding by Kohler [17], the result shows that the checksum has the largest processing overheads. He achieved this using a click router and classified the IP forwarding functions into several elements such as: the CheckIPHeader which calculates the checksum, the LookupIPRoute which performs table lookup. The costs were measured by Pentium III cycle counters. The profiling result shows that the CheckIPHeader dramatically outranges over the other elements.

In Duke [18] it was observed that at (MTU = 1500 bytes) and data rate of 300Mb/s, copy and checksum functions take over 30% of the CPU processing time. Also at maximum transfer unit (MTU) of 8kb and data rate of 300Mb/s, the copy and checksum functions constitute over 30% of the CPU utilization. While over (45%) of CPU utilization was observed for copy and checksum function at (MTU = 56kb) and data rate of 400mb/s.

CHECKSUM CONCEPT AND CALCULATIONS

Checksum are used to ensure the integrity of data portions for data transmission or storage. Due to transmission errors, the transmitter calculates a checksum of the data and transmits the data together with the checksum. The receiver calculates the checksum of the received data with the same algorithm as the transmitter. If the received and calculated checksum don't match, a transmission error has occurred.

Traditionally, the Internet has been using a 16-bit checksum. The sender calculates the checksum by following these steps [4]:

1. The message is divided into 16-bit words.
2. The value of the checksum word is set to zero (0).
3. All words including the checksum are added using one's complement addition.
4. The sum is complemented and becomes the checksum.
5. The checksum is sent with the data.

The receiver uses the following steps for error detection:

1. The message (including checksum) is divided into 16-bit words.
2. All words are added using one's complement addition.
3. The sum is complemented and becomes the new checksum.
4. If the value of checksum is 0, the message is accepted; otherwise it is rejected.

Checksum Calculation for IP Packet Header

The calculation of the checksum is a sequence of 16-bit one's complement additions. Let's take the checksum calculation of the IP packet header as an example. The IP packet header is depicted in Figure 2.1. Each row of the header contains 32 bits and is divided into two 16-bit fields. To compute the checksum, the checksum field in the header is first set to all zeros. The 16-bit one's complement addition is performed over every half of the row, and the final result which is the complement of the final sum is put in the Header Checksum field [1, 5].

ANALYSIS AND DESIGN OF A TCP/IP CHECKSUM SYSTEM

The checksum system is used to maintain a running sum of data sent or received over a communication line. Normally, the data is sent as a packet of data bytes containing the computed checksum from the sender. The system sums all new data presented to it.

The architecture for the state machine for checksum calculation divides the machine into three parts: system, data subsystem, and control subsystem. The system connects the data and controller subsystems. The data subsystem maintains the values of data manipulated by the machine using registers to

hold the data value and multiplexers when more than one input to the register is possible. The controller computes the controls to the multiplexers and registers within the data subsystem. The architecture diagram is shown in Figure 2. The System has external inputs and outputs that connect to either of the two internal subsystems. The main purpose of the System is to connect the Data and Control subsystems [13].

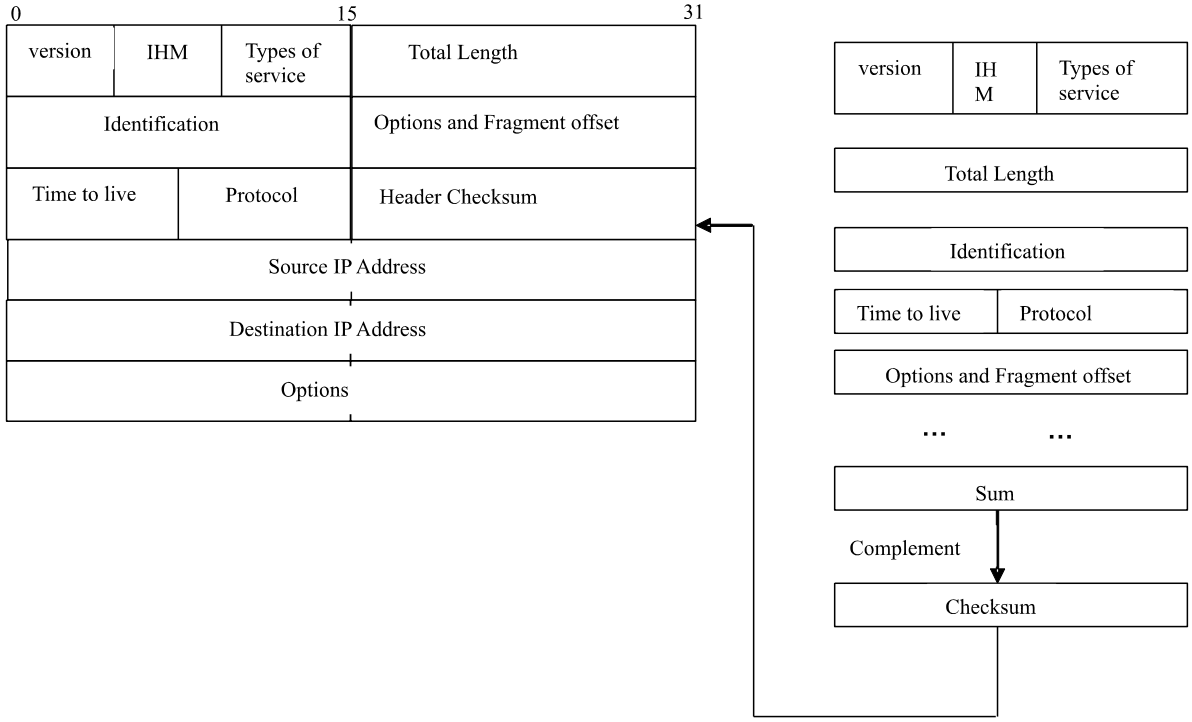

Figure 1: Checksum Calculation for IP Packet Header

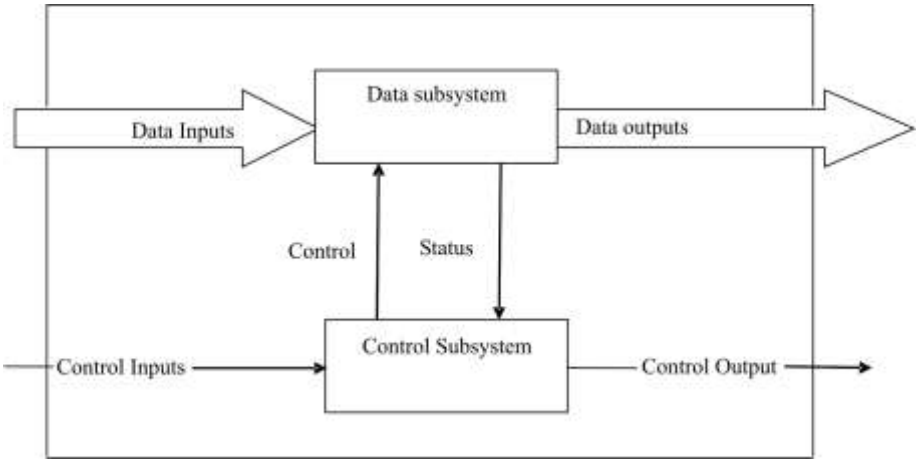

Figure 2: Architecture of Proposed Checksum System

To synchronize its operation with the outside world, the system is set to **run**, given a **data** byte, and notified that there is **newdata** to sum. For instance, given that the data packet contained:

03	06	04	02

The system will receive the data input of 03, 06, 04, 02 in hexadecimal. The sequence of external data and control given to the checksum system is shown in steps 1-8 of Table 1. When Run = 0 and newdata = 0 the system is at its initialization state with the initial sum equals to zero. When Run = 1, the system is notified of the presence of a packet of data (four bytes in this case). On being presented with a data byte from the packet, newdata changes from 0 to 1, thus the first data byte is fetched and added to the initialized sum and subsequently other data bytes are fetched consecutively and accumulated. The computed sum accumulates as 0, 3, 9, 13, 15 (00, 03, 09, 0D, 0F in hexadecimal). The complement of the final computed sum is regarded as the checksum (F0). When newdata is low, computation is put on hold, and the value of the result remains constant within that period. The checksum system developed in this research must indicate when the checksum has been computed (it is **done** at run = 0 and newdata = 0). Accordingly, step 9 begins a new data packet indicated by run changing from 0 back to 1. In actual use, the sender's packet would be recomputed to determine whether the data packet had been correctly received [13].

Table 1 Typical Simulation Illustration Table

	Data	run	New data	Additions Result	done
1.	03	0	0	00	1
2.	03	1	0	00	0
3.	03	1	1	03	0
4.	06	1	1	09	0
5.	04	1	1	0D	0
6.	02	1	1	0F	0
7.	02	1	0	0F	0
8.	02	0	0	0F	1
9.	07	1	0	0F	0
10.	07	1	1	07	0

METHODOLOGY

The following steps are used to achieve the above stated objectives:

I. Determining the inputs and outputs.
II. Defining states and transition conditions in a state diagram.
III. Defining the outputs of each state (the outputs include device outputs and state control outputs).
IV. Determining computational device required.
V. Diagram for data /control subsystems and connections.
VI. VHDL codes for the control subsystem, data subsystem and the system, for compilation and simulation.

Inputs and Outputs

The diagram below (Figure 3) represents a black box with the required inputs and outputs. Here **Data** and computed sum (**Result)** are n-bit inputs and outputs, while **run**, **newdata**, and **done** are single-bit control signals.

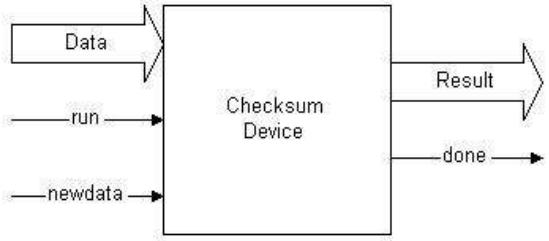

Figure 3: Checksum system with I/O ports

States and Transitions

Figure 4 illustrates first the high-level operations necessary (Figure 4a). These high-level operations are decomposed to control outputs of multiplexers and registers (Figure 4b).

Registers: Since a value must be maintained through multiple states, it can be made a register (the alternative is to set the value in each state). That includes any internal values and outputs. For the system in this research, there is **Sum, Result,** and **done** that are maintained. Sum and Result are in n-bit registers, **done** in a single-bit register (flipflop) or be set in each state (the method actually used). In this design, only the n-bit values (Sum and Result) are formally treated as registers and named as **rSum** and **rResult** registers respectively. When to assign a value

to the rSum register is controlled by the **rSumLoad** signal from the control subsystem. If rSumLoad = 1, the register value changes on the clock edge. The same is true for rResult register, when rResultLoad = 1. The purpose of rSum register is to hold the accumulation sum, while the rResult register is to hold the Result after the sum is computed and run = 0 (sum'ing is stopped), since Sum = 0 when run = 0.

Multiplexers: The need for a multiplexer is determined by whether a variable has multiple assignments. The checksum device has one variable **Sum** with two assignments (**Sum=0**, and **Sum=Sum+Data**). The multiplexer selects the input that the **rSum** register will receive, when the control signal **muxSum** is 0 the input of "00000000" is selected, when 1 the input **Sum+Data** is selected.

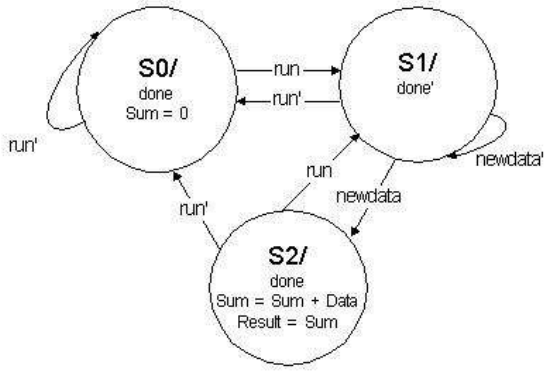

Figure 4(a): Moore State Diagram for Checksum Device- High-level

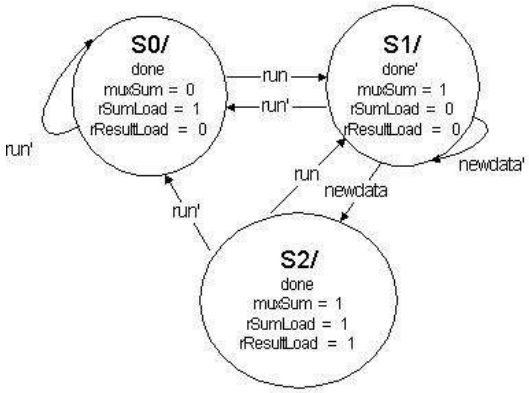

Figure 4(b): Multiplexer/Register-level

The Computational Device

The checksum requires the operation of **Sum+Data** which can be implemented using the **Carry lookahead adder**. The implementation of the checksum function can be reduced to the implementation of a 16-bit one's complement adder. Furthermore, one's complement addition can be performed by a two's complement adder by propagating the carry-out signal to the carry-in. Therefore, the first target is to design a fast 16-bit two's complement adder.

Data Subsystem

Steps 2-4 of the methodology produces the data registers needed to hold state information, multiplexers to select between data sources, and computational operations on the data. A diagram of the devices and connections as shown in Figure 5 can help visualize data subsystem architecture. The Data subsystem receives three Control subsystem inputs and the Data input, outputting the Result. The data subsystem component in VHDL would have an interface similar to [13]:

```
COMPONENT DataSubsystem
PORT (clk : IN STD_ LOGIC;
MuxSum, rSumLoad, rResultLoad: IN STD_ LOGIC;
Data: IN STD_LOGIC_VECTOR (7 DOWNTO 0);
Checksum: OUT STD_LOGIC_VECTOR (7 DOWNTO 0));
    END COMPONENT;
```

Control subsystem

The control subsystem of Figure 6 consists of the implementation of the FSM corresponding to the state diagram. The state diagram is implemented directly in a high level VHDL representation. The VHDL control subsystem component has an interface similar to [13]

```
COMPONENT ControlSubsystem
PORT (clk, run, newdata: IN STD_ LOGIC;
MuxSum, rSumLoad, rResultLoad, done: OUT STD_ LOGIC);
    END COMPONENT;
```

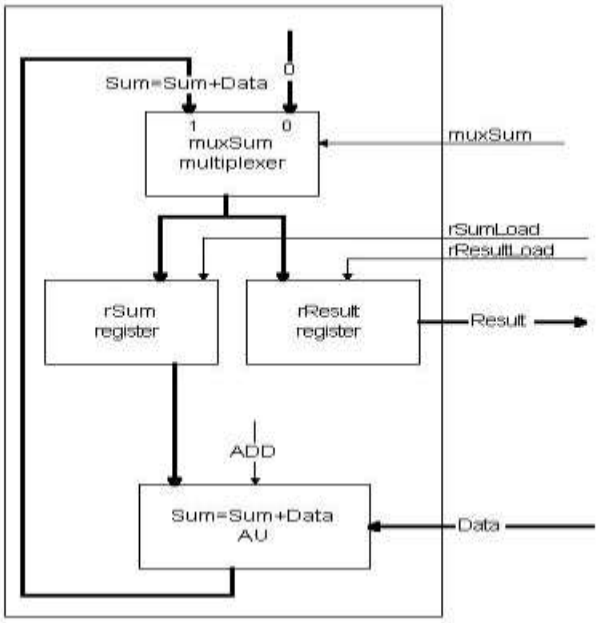

Figure 5: View of Data subsystem details

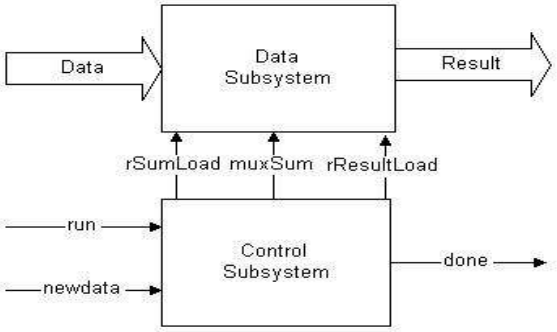

Figure 6: View of Data and Control subsystems

The Checksum Computation System

The design takes 64 bits per clock cycle. This is a modification of the control and data subsystems of Figures 5 and 6. To achieve this, four adders are used to perform the 64-bit calculations and accumulation of the addition results (see Figure 7). The first two adders are placed in parallel. Each has two 16-bit inputs, thus making 32-bit word. Both adder inputs are combined to give a 64-bit word input. When data is applied at the inputs pins, calculation is effected by the two parallel adders (CLA1 and CLA2) and their respective addition results are sent to the input pins of the third adder (CLA3) for further additions. The addition result from the third adder is then forwarded to the fourth adder (CLA4) to perform accumulation. The result from the fourth adder is transferred through multiplexer1 to the result register's input and from the register the result is forwarded through multiplexer2 to an inverter which receives and complements the input value to produce the checksum output.

The computer processor, on the prompting by the **done** signal from the system, fetches the Checksum value from the inverter output for placement in the checksum space of the TCP or IP header, if the host is the sender; or would compare the checksum value with the value (0000) to determine its validity, if the host is the receiver. However, if the checksum value is not the same as (0000), then the packet is discarded.

IMPLEMENTATIONS AND RESULTS

Quartus II software (9.1 web edition) is deployed for the compilation and timing simulation of the designed system.

TIMING SIMULATION

The designed checksum function is used to maintain a running sum of data received or sent over a communication line. Normally, the data is sent or received as a packet of data bytes with the computed checksum. The checksum calculation in the IP header is used as an example. A typical IP packet header contains the data as shown in Figure 8.

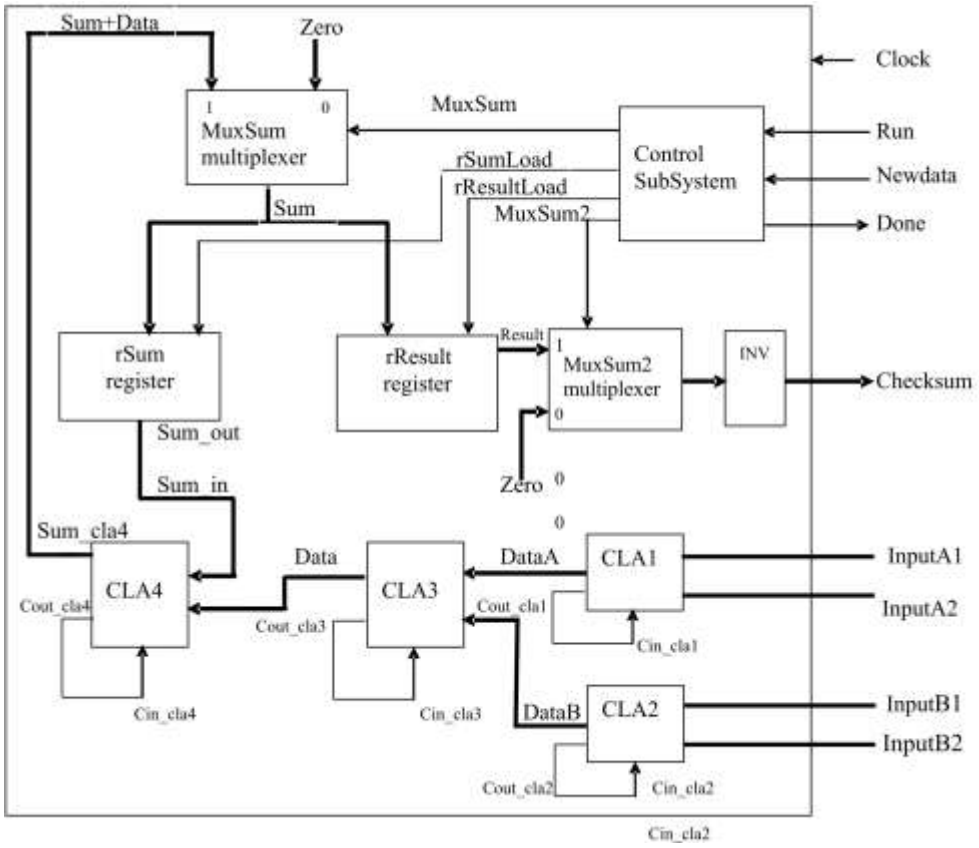

Figure 7: Modified Checksum System

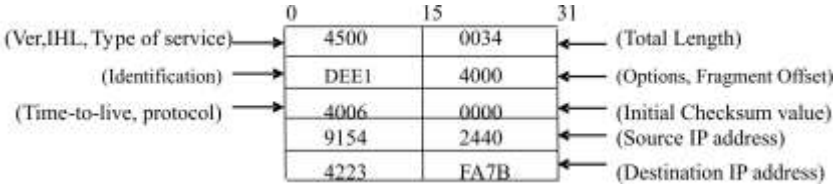

Figure 8: IP Packet in 64-bit words

The IP header contains 20 bytes, and structured into five 32-bit word data. Consequently, there are three 64-bit word inputs for the 64-bit circuit as shown in figure 9.0.

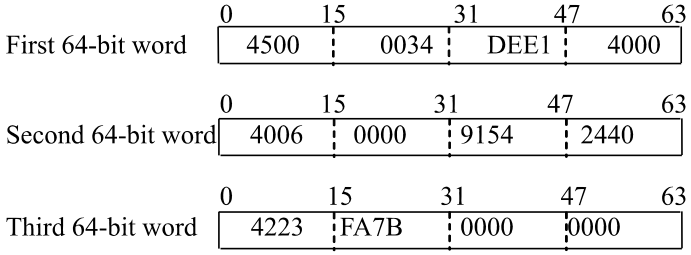

Figure 9: 64-bit words of the IP header

The other half of the third 64-bit input data word is padded with zeros to make up for a complete word. The waveform for the timing simulation is shown in Figure 10.

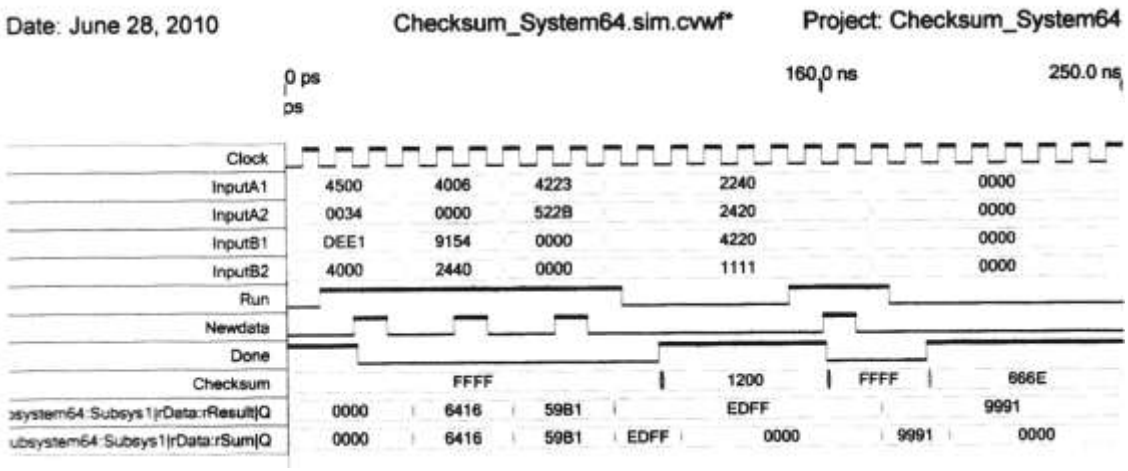

Figure 10: The waveform for the timing simulation

The waveform depicts the required behavior with the use of test vectors, such as the IP **data, run, newdata, rsum, rResult** and **done**. The **run** signal synchronizes it operations with the outside world. When it changes from low to high, it notifies the system that a new data packet is available, and the system initializes **Sum** to zero (rSum = 0000), and starts a new addition. The **newdata** signal, when at low puts the system on a hold state and when high, notifies the system of the presence of a new data bytes or word from the data packet to add and accumulate. The final sum is held in rResult register and transferred to an inverter which complements it and produce the checksum value. The purpose of **rResult** register is to hold the current result of **rSum** register after the accumulation is ended when **run = 0.** The result is then inverted by an inverter and sent to the output as the checksum value. The system indicates with **done** when the checksum is computed**.**

In the waveform diagram, the **Sum** and **Result** values are treated as **rSum** and **rResult** registers respectively. Also, at times 10ns and 150ns a new data packet begins, indicated by **run** changing from 0 to 1. Throughout the period of additions and accumulations of data, the checksum value remains as FFFF (complement of 0000), until after checksum'ing is concluded and stopped, before the actual checksum value is presented at the output. In order words, the system always initializes before the commencement of a new checksum'ing process. The accumulation uses the expression:

$$Sum = Sum + Data \quad (1)$$

The (+) symbol here represents the summing functionality of the carry-lookahead adder CLA4, (Figure 7.0).

Expression (1) is the same as:

$$rSum = rSum + Data \quad (2)$$

Where

$$Data = (InputA1 + InputA2) + (InputB1 + InputB2) \quad (3)$$

The (+) symbol in between the brackets here represents the summing functionality of the carry-lookahead adder CLA3, (Figure 7), and that within the brackets represent CLA1 and CLA2 adders respectively.

The system is first initialized to zero (Run = 0, rSum = 0000). At the rising edge of the clock pulse at 25ns (above **newdata** pulse) when (Run = 1), the first set of data is loaded into the inputs (inputA, inputA2, inputB1, inputB2) of adders (CLA1 and

CLA2). Addition of the data is effected and their outputs forwarded as inputs to CLA3 adder to produce the value (6416). This becomes an input to adder CLA4. At the same time the initialized value in rSum register is clocked to the other input port of adder CLA4, where the accumulation of the values is effected to produce the sum (0000 + 6416 = 6416), which is subsequently transferred to the output of rsum register at 37.522ns to replace the former value. The propagation delay due to the loadings through the input pins and additions of data through the adders, and the accumulation that produces the sum and eventual transfer to the rSum register output is (37.522 – 25.00 = 12.522ns). At other instances of the positive edge of the clock, (55ns and 85ns) the process of additions and accumulations are repeated on the next set of data within the data packet. When run go from high to low, computation is ended, and the system goes back to the initialization state, but the rResult register retains the final value (EDFF) of the concluded computation. The complement of this value gives the checksum value (1200). Stratix III, EP3SE50F484C2 is the chosen target FPGA device. The timing analyzer's report contains the following information:

F_{max} -- 204.75 Mhz (performance for the slow 1100mv 85c Stratix III model)
F_{max} -- 222.17 Mhz (performance for the slow 1100mv Stratix III model)

For the 1100mv 85c Stratix III model, since 64 bits are processed per clock cycle, the throughput is calculated as follows:
Throughput = 64 bits * F_{max} mhz = 64 * 204.75 * 10^6 = **13.104 Gbps**

For the 1100mv Stratix III model, the throughput is:
Throughput = 64 bits * F_{max} mhz = 64 * 222.17 * 10^6 = **14.218 Gbps**

The throughput is 13.104 Gbps and 14.218 Gbps respectively, which satisfies the requirement for the design. For the fields that change in the course of transmissions, like the TTL of the IP hearder, the checksum calculation could be replaced by the checksum incremental update, which recognizes the changed fields in the data to be calculated, and perform checksum update using the following equation [1, 15]:

$$HC` = \sim (\sim HC + (- m) + m`)$$
$$= \sim (\sim HC + \sim m + m`) \quad (4)$$

Where HC` is the new checksum, HC is the old checksum, m represents the old value of a 16-bit field and m` represents the updated value of a 16-bit field. The (~) symbol denotes (complement of). Consequently, the total number of additions is reduced. For example, the IP header has 20 bytes, if checksum calculation is performed over the whole IP header using the 64-bit machine, 5 additions will be performed. However, since there are only two changed fields (TTL and Checksum), only 1 addition is needed. Consequently, the throughput of checksum calculation speed will triple. For instance, to compute the new checksum in the next hop, the following data and computations are required:
HC (old checksum value) = 1200 (from Figure 10)
M (old TTL value) = 4006 (from Figure 8)
M` (new TTL value) = 3006
Complement of HC = EDFF
Complement of M = BFF9
HC` (new checksum value) = ~ (EDFF + BFF9 + 3006 + 0000)
= ~ (inputA1+inputA2+inputB1+inputB2) – using checksum device.
= ~ (DDFF) = **2200** (5)

Three clock pulses are required to compute the entire IP packet header in the former host, but only one is needed here, thus the computational speed of the checksum is tripled.

CONCLUSIONS

The processing functions in the TCP/IP stack are investigated and focus is on the data plane which requires processing at transmission speeds. Micro-level functions are reviewed, and network services built upon these micro-level functions are identified. Based on

profiling results, the checksum function is selected as a performance-critical function and then implementation in an FPGA. The implementation details of the selected TCP/IP function is discussed - a 64-bit checksum system was designed, compiled and simulated successfully with throughputs of 13.104 Gbps and 14.218 Gbps.

The simulation result shows that the research objective is successfully achieved. Test results and performance confirms that the checksum function in FPGA is a viable option for offloading the checksum calculation from the GPPs at transmission speeds of the order of 10 Gbps and above. Consequently, this eliminates the latency of the read/write operations between the processor and memory, the latency due to the buffer/memory copy operations, and that due to I/O bus speeds limit. The system can also be used with the new IPv6 protocol.

REFERENCES

1. W. Lu, "Designing TCP/IP functions in FPGAs", M.Sc thesis, Delft University of Technology, China, 2003. [online]. Available:http://ce.et.tudelft.nl [Accessed: April 14, 2008].
2. Chen Zhonghe, "TCP/IP offload engine (TOE) for an Soc system", National Cheng King University. [online]. Available: http://www.altera.com/literature/de/3.3-2005_Taiwan_3rd_chengkuregu-web.pdf
3. T. Henriksson, etal, "VLSI implementation of internet checksum calculation for 10 gigabit Ethernet", Linkopings University, Linkoping. [online].Available: http://www.tomhe.isy.liu.se. [Accessed: April 4, 2009].
4. B. Forouzan, Data Communications and Networking, 4th ed. Singapore: McGraw-Hill, 2007. [online]. Available: http://www.mhhe.com/forouzan
5. Ack. William, "TCP/IP checksum calculations", [online]. Available: http://www.codemiles.com/post524.html [Accessed: March 3, 2009].
6. S. Brown and Z. Vranesic, Fundamentals of Digital Logic with VHDL Design, 2nd ed. Toronto: McGraw-Hill, 2005.
7. EE201-Homework 4, "System – level optimization of the embedded webserver". [online]. Available: http://www.ee.ucla.edu/~schaum/ee201/
8. J. Kay and J. Pasquade, "Profiling and reducing processing overheads in TCP/IP", University of California, San Diego. [online]. Available: http://www.jkay.cs.ucsd.edu. [Accessed: March 3rd, 2009.]
9. Mel Tsai, etal, "A benchmarking methodology for network processors", University of California, Berkeley. [online]. Available: mtsai@eecs.berkeley.edu
10. V. A. Pedroni and E. Kaufmann, "Digital electronics and design with VHDL", 2008. [online]. Available: http://www.ebyte.it/library/refs/Refs_EE_Books.html
11. Prof. Grishman, "Lecture 9-carry lookahead", [online]. Available: http://www.cs.nyu.edu/courses/fall08/v22.0436-001/lecture9.html
12. Tim Pagden, "Carry look ahead blocks", DOULOS, 1996. [online]. Available: http://www.doulos.com/knowhow/vhdl_designers_guide/models/carry_look_ahead_blocks/ [Accessed: Dec. 12, 2009]
13. J. F. Doyle, "Laboratory10 and Homework10 Register Transfer Logic", [online]. Available: http://homepages.ius.edu/JFDOYLE/c421/html/hw10.htm
14. Altera Corporation, Quartus II 9.1sp1 web edition, August 2009.
15. T. Mallory and A. Kullberg, "Incremental updating of the internet checksum", BBN Communications, January 1990. [online]. Available: http://tools.ietf.org/html/rfc1141
16. http://www.mbdowney.com/ee14713_vhdl_code.html
17. Eddie Kohler, "The click modular router", Ph.D thesis, Massachusetts Institute of Technology, 2001.
18. http://www.cs.duke.edu/ari/publications/tcpgig.pdf