

Third Prize

TCP/IP Offload Engine (TOE) for an SOC System

Institution: Institute of Computer & Communication Engineering, National Cheng Kung University

Participants: Zhan Bokai and Yu Chengye

Instructor: Chen Zhonghe

Design Introduction

Today, the Internet plays an important role in everyone's life. The 100-Mbps network system has become very popular in schools and offices, while the 1-Gbps network system is deployed in server host networks such as large portal websites and Storage Area Networks (SAN). Meanwhile, the Internet SCSI (iSCSI) protocol is recognized as the new standard for the emerging network storage technology—networking of storage component. However, the iSCSI protocol will place heavy demands on server processing when applications run on high-speed networks. According to recent studies, you need 100% efficiency from a Pentium III, 1-GHz processor or 30% efficiency from a Pentium 4 2.4 GHz processor to process the 1-Gbps TCP protocol. Therefore, it is necessary to accelerate network-processing capability and reduce the CPU load by adding extra hardware.

Accelerating system processing has a bearing on the built-in system featuring network functions. The network communication protocol performance depends on memory (memory access times) and data volume. For example, you need lots of memory access to read data when computing checksums, which leads to an efficiency bottleneck. Additionally, when the network interface controller (NIC) receives large volumes of data, the data is split into many different segments based on protocol restrictions. Therefore, subsequent interrupts affect the CPU efficiency. If protocols are processed through additional hardware and with an enhanced memory management mechanism, the above problems can be solved effectively.

The left side of Figure 1 shows a standard TCP/IP stack. To implement TCP/IP in an embedded system, we need two important blocks: the PHY and media access controller (MAC) that function in the physical layer and data-link layer.

These function blocks also operate under the local real-time operating system (RTOS) in the third layer (network layer, IP layer) and the fourth layer (transport layer, TCP layer). Thus, all TCP/IP protocols can be implemented in a software-only solution. We replaced the network protocol with our TCP/IP Offload Engine (TOE), which you can see in the system’s architecture on the right side of Figure 1. TOE uses hardware modules to implement protocols, and uses drivers to communicate with upper-layer space and operating systems.

Figure 1. Network Protocol Stacking (TCP/IP Stack) & TOE Engine

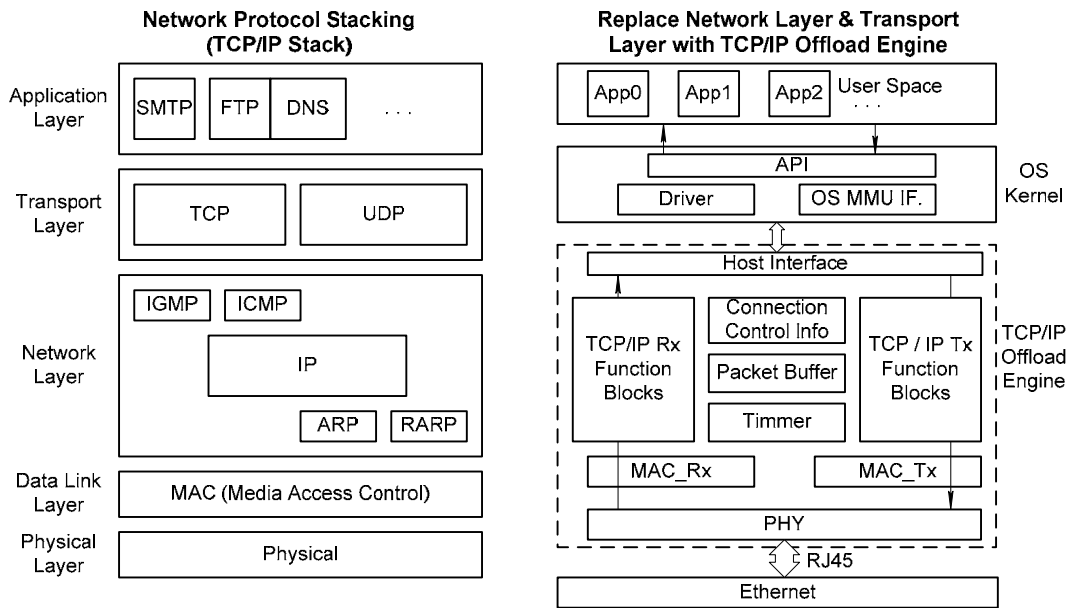
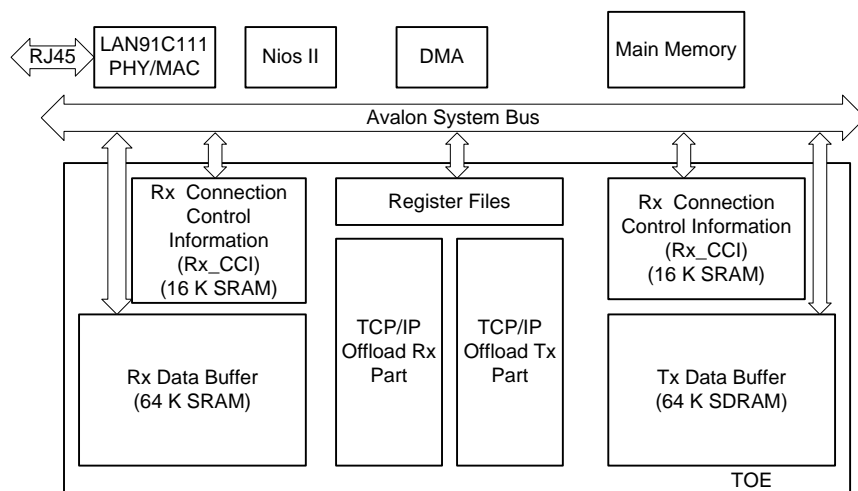


Figure 2 shows the system overview of our design in the FPGA. The system adopts the Avalon[®] system bus. In addition to the TOE architecture, the design includes system main memory, a Nios[®] II processor, DMA, the LAN91C111, and a PHY/MAC chip controller. We will describe the TOE’s major functions and blocks in a later section.

In this system, the Nios II processor functions as the primary controller for data communication between the MAC, TOE, and main memory. The MAC receives a message box data and sends it to a data buffer space of TOE (TCP/IP Offload Engine) through firmware, which processes the data and sends it to the main memory. While transmitting, MAC first sends the data to the internal data buffer space of the TOE, which then adds headers and transfers it back to the MAC for transmission.

Figure 2. Hardware Overview



We used the Nios II processor because the Altera® PCI development board provides peripherals such as the PCI interface and SDRAM, which are required by the system. Although we were unable to achieve the final goal in this contest, that is to use SDRAM as a data buffer to support more connections and use the PCI interface to communicate with the upper-layer operating system (OS), we selected the above development board and related Altera integrated development environment (IDE) to facilitate easy development and scalability in the future.

Function Description

Although we designed the general TCP function block, we could not complete it due to time constraints. Therefore, the final product does not provide the TCP function. The paragraphs below will mark this part as “Unfinished”.

Major Functions & Blocks

Figure 1 shows a complete TCP/IP stack. However, because the whole network protocol is too complicated, we have only implemented the most used functions of TCP/IP protocols. The white shaded blocks (ARP, ICMP, IP, UDP, and TCP) in Figure 1 were implemented in our design. Three major functions are included in this design:

- Send pings and respond to echo request packets (ARP & ICMP).
- Provide UDP transmission capability of up to eight connections simultaneously.
- Establish and manage up to eight TCP connections simultaneously (unfinished).

In the hardware scheme shown in Figure 2, the TOE body includes modules as follows:

- Four SRAMs (alt_synram):
 - Two 64-KByte SRAMs are used as data buffers, which are temporary packet storage for receiving and transmitting (Rx and Tx data buffer).

- Two 16-KByte SRAMs are used as storage blocks for connection control information (CCI) of receiving end and transmitting end. The two blocks will record the status of data buffer, and the queue information of protocol processing sub-module. All data except packet data are stored in this CCI.
- Rx and Tx protocol processing blocks:
 - Rx protocol processing module.
 - Tx protocol processing module.

The two protocol processing modules consist of small modules, which are responsible for partial logic functions of specific protocol layers respectively.

- Register files (TOE internal buffer), which generally includes the following items:
 - CPU control bit.
 - TOE status.
 - Control buffer of item addition at transmitting end (CPU is used to initiate a buffer group that sends work instruction).
 - Control buffer of item addition at receiving end (CPU is used to initiate a buffer group that receives work instruction).
 - ARP table control buffer.
 - UDP control block control buffer.
 - TCP control block control buffer (unfinished).
 - Protocol modules control buffer.
 - Queue substrate of protocol processing sub-module and item number control buffer.

Refer to the “Design Architecture” section for the implementation of each module.

Performance Parameters

Because the TCP module is unfinished, the buffer access times and memory data volume are used for TCP performance evaluation.

Performance Evaluation

The performance can be evaluated as follows.

- *100-Mbps wire-speed*—In Ethernet applications, the TCP maximum segment size (MSS) is usually set to 1,460 bytes, while the IP maximum transfer unit (MTU) is set to 1,500 bytes. Therefore, the size of the message box is often 1,518 bytes (a MAC header is 14 bytes, and a CRC is 4 bytes). In addition, there are eight bytes of preamble when the MAC transmits a valid message box. Therefore, to receive a message box it needs $1526[x]8[x]10 \text{ ns} = 122080 \text{ ns} = 0.12179 \text{ ms}$, that is we need to process a frame within 0.12208 ms or 12208 clock cycles.

- *1-Gbps wire-speed*—The timing is 1/10 of the result above, meaning we need to process a frame within 12.2 μ s or 1220.8 clock cycles.

Practical Performance Analysis

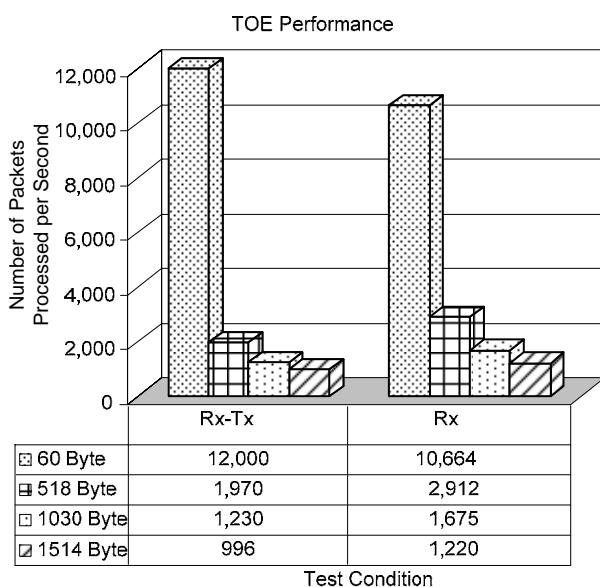
The test scenario is:

- TOE: operation frequency 40 MHz
- PC transmitting packet: 1 GHz Linux running UDP program
- Measuring packet volume: 2 GHz Windows XP running Ethereal

TOE Performance

Figure 3 shows the number of packets in different sizes that the TOE can process per second. Rx-Tx means that when receiving a packet, the TOE always transmits a responding packet in the same size, so the number of packets processed shall include Rx packet and Tx packet. Rx means a packet in response is transmitted upon receiving 1,000 packets. Therefore, the number of packets processed is the Rx packet number. Additionally, the Nios II CPU processes all data replications.

Figure 3. TOE Performance



In Rx-Tx testing, when the packet size is 60 bytes, the number of packets processed by the TOE per second can go up to 1514 bytes (packet size). This is because the TOE has three data replications from receiving to transmitting: the first from MAC buffer to TOE RxBuffer, the second from TOE RxBuffer to TOE TxBuffer, and the third from TOE TxBuffer to MAC.

In Rx testing, when the packet is 60 bytes, the number of packets processed by TOE per second can go up to 1514 bytes (packet size), which is 12 times the size when compared with Rx-Tx testing. This is because the replication from TOE RxBuffer to TOE TxBuffer is reduced.

In Rx-Tx testing, when the packet is 60 bytes, the number of packets processed by TOE per second is 12,000; that is, the packets received are 6,000. However, when provided with the same packet size, TOE only receives 10,664 packets per second in the Rx testing. This is because the CPU has to process data replications of Rx packets and Tx packets simultaneously.

Provided that the time for data replication is deducted, and if we assume a packet size of 1514 bytes, the data volume received per second is $6000[x]1514/1000000=9.084$ Mbytes.

In Rx testing, when the packet is 60 bytes, the number of packets processed by TOE per second is 10,664. Provided, time for data replication is deducted, and assuming packet size of 1514 bytes, the data volume received per second is: $10664[x]1514/1000000=16.145$ Mbytes.

As a result, the performance bottleneck of TOE lies in the data replication. If the time of data replication is reduced but the speed is accelerated, the TOE will be able to process 100 Mbps network speed. To process 1-Gbps data, we need to improve the processor frequency of up to $1\text{ Gbps}/16.145\text{ Mbytes}=7.8$ and $40\text{ MHz} \times 7.8 = 312\text{ MHz}$.

Design Architecture

This section describes the system’s architecture.

TOE Hardware System Design Concept

As shown in Figure 4, we divided stacks into two modules: Rx and Tx, which are both ASICs and are attached to the same system bus. Therefore, they can be controlled by the same embedded CPU and share a common memory. However, the TCP module needs to communicate with input modules in full-duplex mode (the size of the sliding window is influenced by ACK reply), and the two modules can operate independently. The advantage of this design is that the transmitting and receiving allows parallel processing as long as some appropriate firmware is deployed.

Figure 4. Parallel Processing Module Dividing (Tx & Rx)

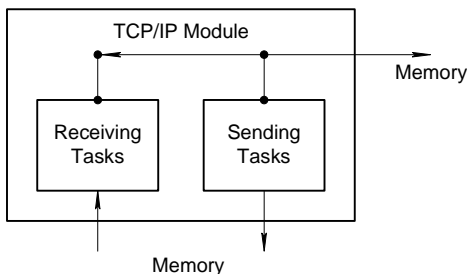
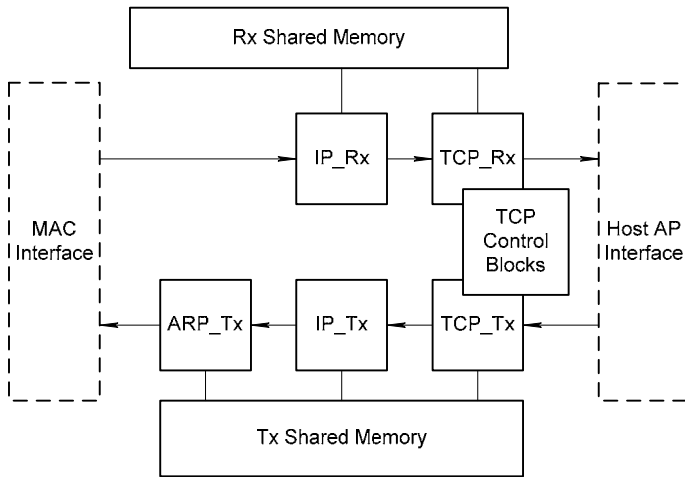


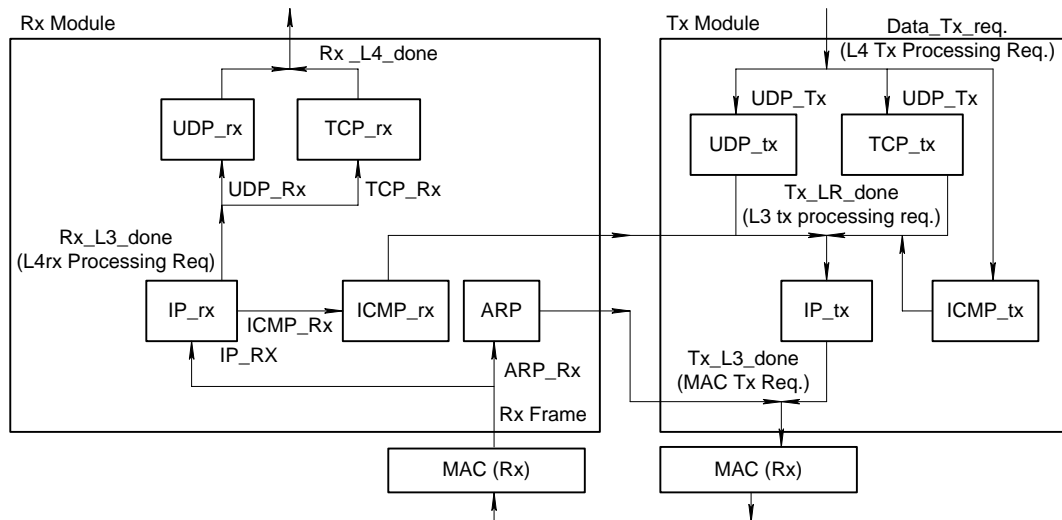
Figure 5 shows the relationship between the TCP packet processing module and the buffer. After the protocol processing is finished on the first layer, the processed data is discarded to the next layer in order to avoid too much memory-to-memory transfer, so that the processing module of each communication protocol will be able to read and write using a shared memory.

Figure 5. Protocol Processing Module Required for TCP Packet Processing & Shared Memory



When it finishes processing a protocol packet, the module notifies the next module of a pointer in the same shared memory so that the processing flow can obtain the data with minimum memory transfer volume. However, no memory exceeding three access ports is available for use. Therefore, we need to determine which memory will come first when reading and writing the buffer via the arbiter circuit when we connect these modules with the buffer. We need to research whether the different applications or connections are related in this arbitration mechanism.

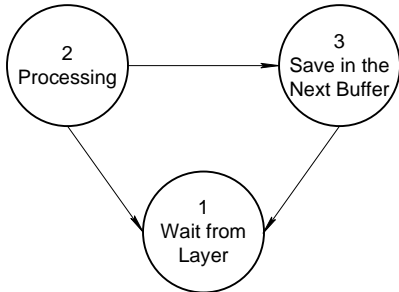
Figure 6. Protocol Module Division Design



The three major functions (TCP, UDP, and ping) in the document are realized based on the *TCP/IP Protocol Suite* by Behrouz A. Forouzan and Sophia Chung Fegan. We have utilized hardware circuits to complete the logic function of each software module (using multi-cycle based finite state machine to realize function of single module by stages), and then connect the logic circuit to shared memory for communication (see Figure 5). To realize parallel processing, the whole TCP/IP stack is divided into several modules as shown in Figure 6. As mentioned above, such a design approach enables it to receive

packets while transmitting. Additionally, when a packet is processing TCP, it can accept another packet to occupy the IP processing module for implementation of protocol-related work on the IP layer. Each protocol module runs the state machine as shown in Figure 7.

Figure 7. State Machine Structure of Protocol Processing Module



In short, these sub-modules use multi-cycle logic circuits to work as software modules. The module group responsible for receiving analyzes the header fields of the message box stored in the data buffer according to sequence, and then processes this data. The transmitting module clusters add network headers to the data segment that needs to be transmitted according to user instructions (for example, driver and system firmware).

Process Communication Queue Among Modules: Buffer Tables, Connection Control Information (CCI)

The communication among the processing modules is shown in Figure 8 and is realized through queues. In our design, we implemented it using a buffer table. The buffer table consists of pointers and information required by packet processing and is stored in the connection control information (CCI) RAM, pointing at a memory block to store complete packets into the data buffer. Each processing module is related to at least two tables; one indicates the data pointer where the data is processed, and the other holds the data pointer required to notify the module on the next layer when the module processing is finished.

In Figure 8, the lines connecting the blocks indicate table names. Each buffer table has several items, with data pointers pointing to the packet located in the data buffer. The ARP module captures the data block required for ARP protocol processing in the ARP_Rx table. It then performs the operation and store the finished data pointer in the Tx_L3_done table. Finally, it waits for the MAC_Tx module to read the pointer and output the data. Figure 8 shows the logic-buffer table and the data buffer. In our design, each data pointer has four fields as shown in Figure 9. Field description is explained following the data buffer design description.

Figure 8. Buffer Table Scheme

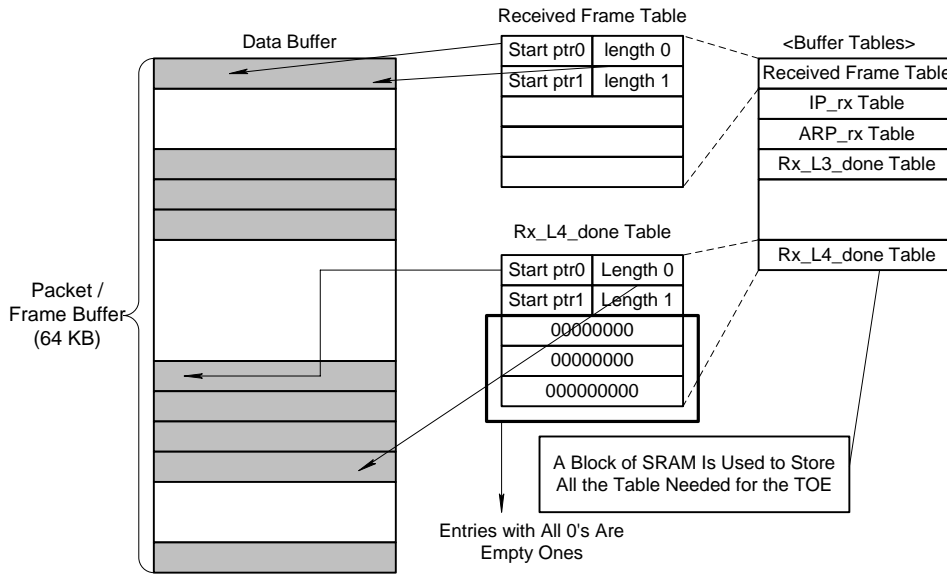
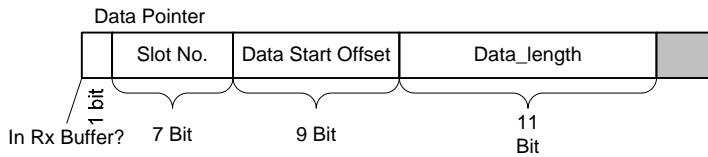


Figure 9. Data Pointers



Data Buffer Area Design

To facilitate buffer management, the 64-Kbyte data buffer area is divided into several slots, each occupied by a message box. The larger message boxes may occupy more slots. In this design, the number of slots is set in the compiled hardware circuit. We have set 128 slots in the current trial version, where each slot takes up 512 bytes. In this way, the buffer management is less complex and requires less CCI storage.

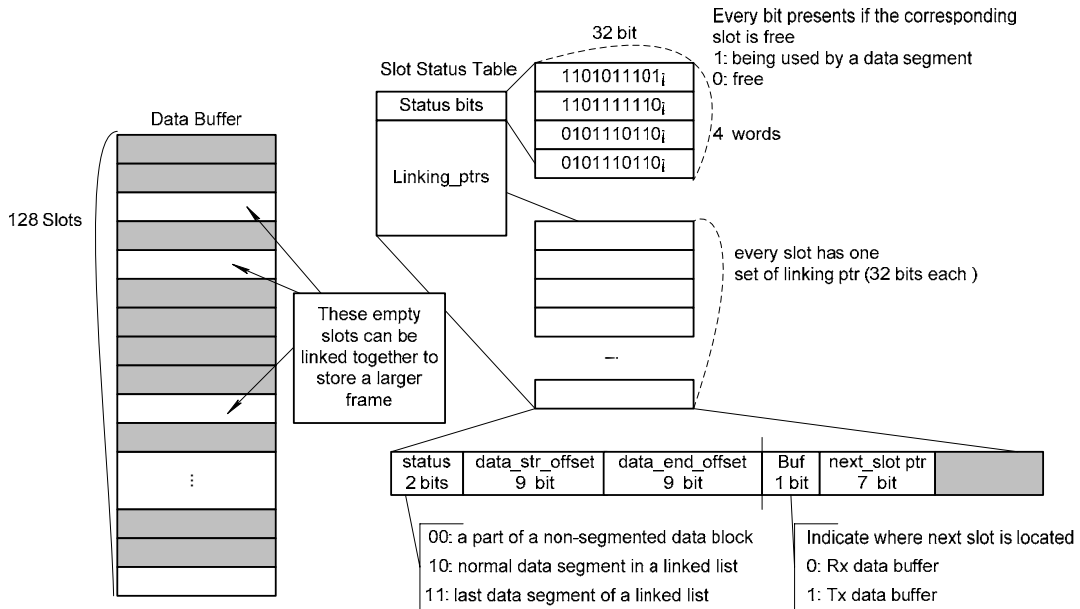
Data pointer fields in Figure 9 can be explained as follows:

- In the Rx buffer: 1 bit, shows whether data is at the receiving end buffer or the transmitting end buffer.
- Slot No.: Number of the first slot occupied by the data.
- Data start offset: Indicates the start offset of the valid data.
- Data length: Length of the data.

The second and third bit can be combined to form a 16-bit data buffer address, pointing to the start point of the data.

Logically, the data buffer puts the contents of the message box into memory. However, because the packets are not processed in sequence, a linked list mechanism changes the small usable area into serial memory blocks, improving the working efficiency of the data buffer. If a frame can obtain a whole list of serial data while requiring the data buffer, the linked list mechanism is not started; instead, the serial idle segments are distributed into the frame.

Figure 10. Data Buffer & Linked List

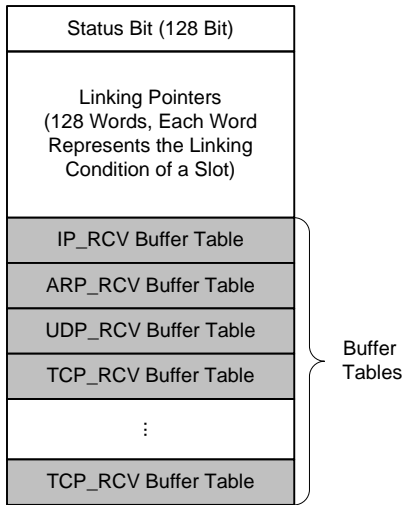


As shown in Figure 10, the idle spaces in the data buffer are not in series because of the non-sequential processing. The best way to deal with this mechanism is to connect the idle segments to a linked list. Except when transferring whole packets (for example, to transfer a message box from MAC to TOE or from TOE to main memory), other packet headers do not need to support a linked list, because the slot size is set to far exceed the overall length of all protocol headers. This means that the linked list does not span the slot while it is processing the packet header.

In Figure 10, we can see that the data buffer and linked list mechanism needs two memory blocks to support it. These memory spaces are put into CCI, and the resulting CCI memory allocation is shown in Figure 11. The first memory area is called status bits, where there are a total of 128 data bits. Each one represents the status of a slot in the data buffer—1 for occupied and 0 for idle. The second memory area is called linking ptrs (linking pointers), which records the interconnection status of occupied data slots. Here there are 128 total units, each with a 32-bit data, and each corresponds to one slot. As shown in the figure, the highest 2-bit indicates the slot status—‘00’ means that the slot is a part of the data that has not be segmented; ‘10’ means the slot is a node of linked list; ‘11’ means the slot is the last node of linked list. Next to this are two 7-bit fields, respectively, indicating the starting and ending shifts of valid data; the last 9 bits is a data pointer to the next data slot.

The linked-list mechanism relies on the Nios II processor for management. So when the hardware module processes headers, the mechanism is not used for one slot. When it needs to send data into TOE, the Nios II processor detects the idle slot by referring to status bits in the TOE; and if necessary, initiates a linked-list mechanism to store the data in the TOE. When the data is moved out of the TOE, it determines whether there is a linked list, then reconnects the processed data and transmits it to the destination.

Figure 11. CCI Internal Memory Allocation



Processing Flow

Figure 12 shows the transmit packet flow.

When the TCP_Tx module is transmitted to the IP_Tx module, you need the data pointer and packet type (see below).



Referring back to the data buffer design in Figures 10 and 11, the data buffers of both the receiving and transmitting ends have two fields in CCI to record the status of the data buffer. These are status bits and linking pointers: status bits use 0 and 1 to record if the slot is idle; the linking pointers record the interconnection status among each slot. When data is transferred into TOE, CPU needs to control the DMA engine for data transfer and modify the data structure that records the data buffer status. This operation ensures correct control of the processing module operation.

Figure 12 explains the process flow for packet transmission.

- When the packet that is going to be transmitted is in the main memory, the CPU checks whether the status bits in the slot status table denote enough space. There may be three outcomes:
 - *Sufficient serial space*—Set the status bit of the corresponding space to be used to 1. Set the status bit of the corresponding Linking_ptrs of slot to be 00.
 - *Sufficient non-serial space*—Set the status bit of the corresponding space to be 1. Each field of the corresponding Linking_ptrs shall be set.
 - *Insufficient space*—The CPU checks continuously until there is sufficient space.
- The CPU calls the DMA to transfer the data from the main memory to the Tx_data_buffer.
- Add an entry into the Tx_data_req and notify the TCP/IP offload engine to transmit some data.
- In the TCP/IP offload engine, each module has a corresponding table, and the data is processed as follows: the upper-layer module finishes data processing and adds the pointer to the data that is put into the table of next layer. In this way, the module knows which data to process by looking at the table.
- After the last layer arp_tx in the TCP/IP offload engine finishes processing, it sends out an interrupt signal. The CPU calls the DMA to transfer the data from the Tx_data_buffer to the MAC buffer to complete the packet transmission.

Figure 13 shows the receive packet flow.

Figure 13. Process Flow of Receiving Packet

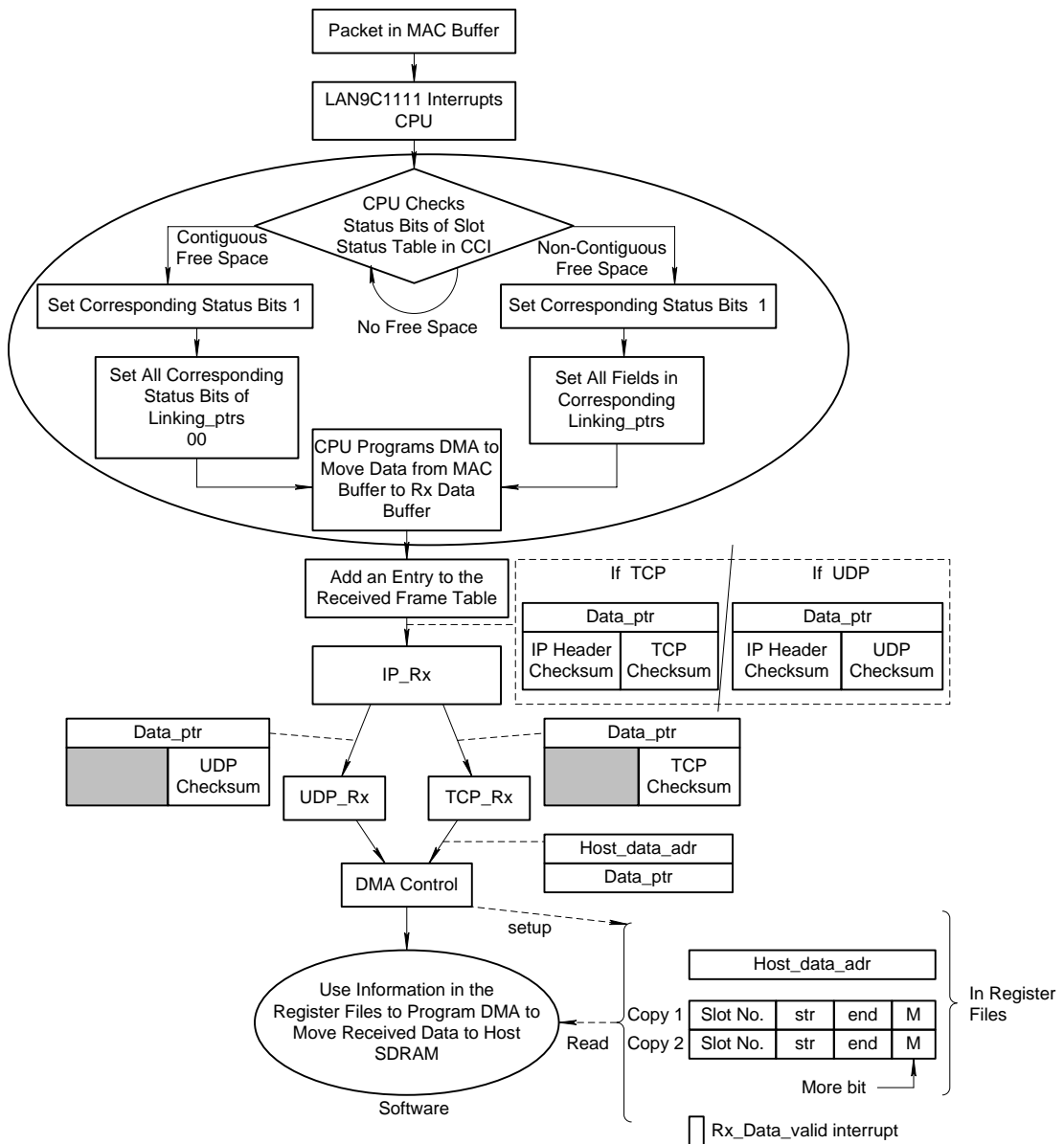


Figure 13 shows the process flow for packet receiving.

- When the packet that is going to be transmitted is in the main memory, the CPU checks whether the status bits in the slot status table indicate enough space. There can be three results:
 - *Sufficient serial space*—Set the status bit of the corresponding space to 1, and the corresponding Linking_ptrs status bit to 00.
 - *Sufficient non-serial space*—Set the status bit of the corresponding space to 1. Each field of the corresponding Linking_ptrs shall be set.

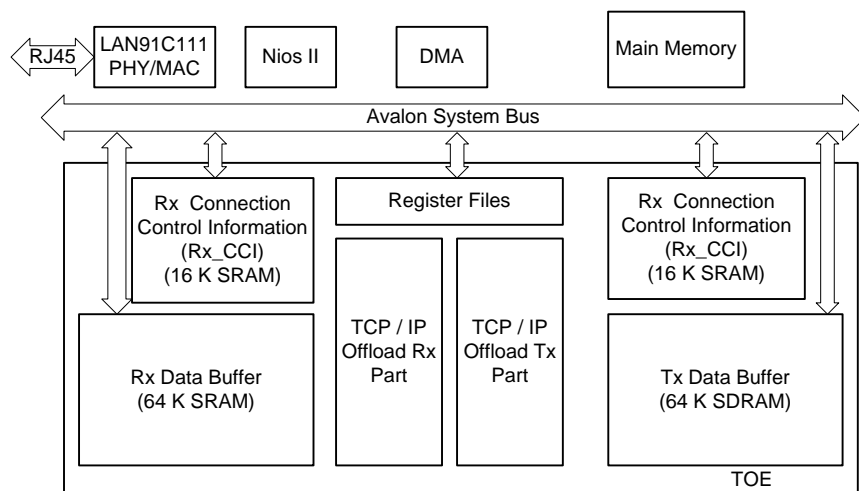
- *Insufficient space*—CPU will check continuously until there is sufficient space.
- The CPU calls the DMA to transfer the data from the main memory to the Rx_data_buffer.
- Add an entry into frame_table and notify the TCP/IP offload engine that the data has been received.
- In the TCP/IP offload engine, each module has a corresponding table, and the data is processed as follows: the upper-level layer finishes the data processing and adds the pointer pointing to the data that is put into the table onto the next layer. In this way, the module knows which data to process by looking at the table.
- After the last layer Tcp_rx and udp_rx in the TCP/IP offload engine finish processing, it sends out an interrupt signal. The CPU calls the DMA to transfer the data from the Rx_data_buffer to the main memory to complete receiving the packets.

Design Methodology

The architecture of the whole system is shown in Figure 14, detailing several main components:

- Nios II CPU.
- DMA engine.
- Main memory constructed by Altera synchronous SRAM (in the future, main memory could be constructed by the development board DDR SDRAM and SDRAM controller).
- Tri-state bridge for Lan91c111 PHY/MAC.
- Custom ASIC TCP/IP Offload Engine.

Figure 14. Hardware Overview



The interconnection of these components was configured using the SOPC Builder tool, and the data was transferred into each component through Avalon system bus. Our TOE features the following five external channels:

- Access channel of CCI system at receiving end.
- Access channel of CCI system at transmitting end.
- Access channel of data buffer system at receiving end.
- Access channel of data buffer system at transmitting end.
- Register file access.

These channels connect with the Avalon system bus through the interface to user logic in SOPC Builder, which is set in slave mode; meanwhile, based on an address mechanism, the Nios II processor can access five channels of TOE through this interface.

The system is controlled by the Nios II CPU, which executes the program located in the main memory. The program initializes the lan91c111 and TOE, waits for interrupts, and then executes the corresponding interrupt service routing (ISR).

The system features the ISR described as follows.

Lan91c111 PHY/MAC ISR

When the processed packet is put into the MAC buffer and the subsequent interrupt is generated, the CPU shall do the following:

- Check status bits in the slot status table to find if the rx_buffer has enough space for the packets.
- Packets are transferred from the MAC buffer to the rx_buffer by calling the DMA function.
- Add an entry to the Rx_frame.

TOE ISR

The CPU checks for the type of interrupt served by the TOE_status_reg:

- *Interrupt receiving packet*—The CPU calls the DMA to transfer the processed packets to the main memory.
- *Interrupt transmitting packet*—The CPU calls the DMA to transfer the processed packets to the MAC buffer.

DMA ISR

The CPU checks status bits in the slot status table to find out whether the Tx_buffer has enough space for packets.

- Packets are transferred from the Rx_buffer to the Tx_buffer by calling the DMA.
- Add an entry to the Tx_frame.

Design Features

The main features of our system design are as follows:

- If we used only the Nios II CPU to process the TCP/IP network protocol, it would be impossible to reach 100 Mbps. However, by deploying both the Nios II CPU and the TOE architecture to process the TCP/IP network protocol, we can reach 100 Mbps.
- Our architecture modularizes each protocol. Each module performs according to its own rules table. Therefore, we could easily add new protocols using the flexible TOE architecture.
- The modules for sending and receiving are separated, and this scheme enables message-box processing in parallel.
- Support for eight UDP connections.
- Hardware ARP table.
- Hardware UDP connection-management mechanism.
- When the processing modules handle the message box, they only transfer the pointers to the message-box, reducing data transfers in memory.
- Due to the network function created in hardware, we were able to reduce host CPU resources occupied by network tasks in embedded applications.
- We can efficiently manage the data buffer by dividing the storage memory into numerous slots, which simplifies the mechanism of data link list. Also, you can easily change the slot size configuration to meet different requirements by simply altering the design parameters.
- The system focuses on network storage devices, so the parameter setting in the current design is set to optimize storage-network applications.
- If you take into account the time from the MAC operation to receiving and storing this data onto destination main memory, our system can process the data packets at 100-Mbps of network speed.

Conclusion

In our opinion, the toughest challenge for the system designer is to have the system verified by application hardware. Fortunately, Altera's PCI development board provides an environment that enables testing of the system and interconnection bus that features CPU, DMA, main memory, PC bridge, and interconnecting peripherals. You can easily construct the system by using the SOPC Builder tool, which connects its own IP with the peripherals. For example, our system uses MAC and PHY as network devices, and we developed firmware using the C language.

We constructed the test environment using SOPC Builder, emphasizing the development of the hardware processing core. If the IP is implemented, its performance can be measured by using the SOPC Builder, which will greatly reduce the IP development time. Although this board provides enough I/O, IP is generally too expensive and we need to devote more effort for developing submodules in case there are not enough resources in the lab. For instance, we could not achieve the expected results in the design because there was no proper bridge connecting the DDR SDRAM Controller and the PCI bus. If we had IP of the trial version or the simulation environment of this bridge peripheral, it would have helped us to dramatically reduce the system development time.

