TCP Offloading Engine Final Paper
CSEE 4840
May 14, 2014

Clementine Barbet (cb3022)
Christine Chen (cpc2143)
Qi Li (ql2163)
# Table of Contents

## I. Introduction

a. High Frequency Trading ................................................................. 4  
b. Transmission Control Protocol ....................................................... 4  
c. Motivation for FPGA Integration ...................................................... 5  

## II. C Programming

a. Motivation ....................................................................................... 6  
b. Structure ......................................................................................... 7  
c. Software API and Golden Reference .................................................. 8  
d. Testing (Wireshark) ........................................................................ 8  

## III. Modules

a. TCP Offloading Engine ................................................................. 9  
b. Block Diagrams .............................................................................. 10  
c. State Diagrams .............................................................................. 12  
d. Timing Diagrams ........................................................................... 16  
e. Modelsim ....................................................................................... 16  
f. RAM ............................................................................................... 19  
g. Packet Builder ............................................................................... 19  
h. Ethernet Packetizer ....................................................................... 20  

## IV. Control Between the Modules

a. Avalon MM Interface .................................................................... 21  
b. Avalon ST ...................................................................................... 21  
c. TCL ............................................................................................... 23  

## V. Lessons Learned ....................................................................... 24  

## VI. Conclusion ............................................................................... 25  

References ....................................................................................... 25  

Source Code ...................................................................................... 26
Description of Included Figures

Figure 1: Depiction of a TCP 3-way handshake
Figure 2: TCP stack with the top-level connected to a server or application
Figure 3: Throughput versus Latency in Software (red) and Hardware (green) [6]
Figure 4: Bypassing the Linux Kernel using Raw Sockets
Figure 5: Screenshot of Wireshark showing successful TCP transmission
Figure 6: High level diagram of TOE
Figure 7: Layout of data that is being stored in the RAM
Figure 8: RAM_searcher top-level state diagram
Figure 9: RAM_searcher checking for an existing connection while 9 (b) shows the insertion of a connection
Figure 10: Packetizer state diagram, where eth_ready is logic indicating incoming ethernet, to_send is the packet to be filled through a combinatorial loop, and next_last is also control logic, set high when the process is grabbing the last set of header data.
Figure 11: This is the conceptual diagram for the Packet_builder state machine. Top illustrates high level signals while bottom illustrates the states for retrieving relevant information from the RAM (with eight total states moving sequentially). Naming convention for each state follows directly with that described in the file. The design is similar to loading the data, although this time we’re retrieving it and loading it into our packet.
Figure 12: Timing diagram of the TOE when a connection is made successfully. This is a conceptual drawing for the hardware portion of the project startup, and we maintained fairly close to this timing.
Figure 13: Timing diagram of the TOE when a connection is made successfully. This is a conceptual drawing for the hardware portion of the project startup, and we maintained fairly close to this timing.
Figure 14: Result from running Testbench1.sv, which requests the creation of a new connection. We expect the connection to go through, returning ID 120. The state machine should be in the state WAIT_RQ, as can be seen above.
Figure 15: Modified Testbench1.sv, with states printed out
Figure 16: Resulting from running Testbench2.sv, which requests the creation of the same new connection twice in a row. We expect the first connection request to go through, returning ID 120, and the second to be rejected, with ERR_FOUND returned.
Figure 17: The timing diagram of Modelsim to go with the state diagram of Fig. 10. The testbench PB_testbench_c, which sets ram_in once and sets wren on for a given duration, is attached. Its functionality is the streaming in of data in the RAM after running the RAM_searcher and (b) zoomed in on the important locations, with relevant labels.
Figure 18: JTAG file for Packetizer.sv, showing that when a faulty connection is made, it parses, then exits
Figure 19: Interfaces between different modules that we implemented
Figure 20: High level diagram of the entire schematic and where the program fits in
Figure 21: More granular image of Qsys (Picture from Ref [11])
Figure 22: System console of running JTAG with a test whereby a connection is successfully made, and when another is requested, it is correctly denied.
Included Source Code

tcp_rawsck.c ................................................................. 26
arp.c ................................................................. 50
main.c ................................................................. 70

toe.sv ................................................................. 71
toe_init.sv ................................................................. 74
RAM2.v ................................................................. 76
RAM_searcher.sv ................................................................. 78
Packet_builder.sv ................................................................. 87
Packetizer.sv ................................................................. 93

testbench1.sv ................................................................. 97
testbench2.sv ................................................................. 101
PB_testbench.sv ................................................................. 107
syscon-test.tcl ................................................................. 109
syscon-test2.tcl ................................................................. 112
I. Introduction
   
   a. High Frequency Trading

   High frequency trading (HFT) refers to a type of algorithmic trading where sophisticated technological tools and computer programs are used to rapidly trade securities. Studies show that HFT accounted for more than 50% of all trade in 2010 on the US-equity market with a growth-rate of 70% in 2009 [6]. In electronic trading of stocks, orders are sent by electronic form to stock exchanges, and a large number of orders are injected to the market with a sub-millisecond round-trip execution time. Bids and ask orders are then matched by the exchange to execute a trade. In order to win the market, large amounts of data has to be handled with minimal amount of processing delay. In general, if the large amount of data is handled by software, the associated processing latency will no longer be competitive enough. Therefore, there has been a growing amount of interest in the finance industry where FPGA-based hardware is used to process the HFT data, utilizing the parallel and dedicated processing ability of FPGA.

   b. Transmission Control Protocol

   The Transmission Control Protocol (TCP)/IP protocol serves as an Internet suite used to transmit data through connections with the aim of transmitting data securely. Under its protocol, it is able to begin connections, accept connections, accept data, save data, and close connections. Most notably, it utilizes a 3-way handshake to connect before transmitting data, as is shown in Fig. 1.

![Figure 1: Depiction of a TCP 3-way handshake](image)

As mentioned in the previous section, in order to handle large amounts of network data rapidly, TCP offload engines are implemented, which use the gigabit Ethernet interface to process the data rapidly. This can only be done on a peripheral FPGA, separate from the CPU, so that it can handle this amount of data with lower latency.
The next step to transmitting packets through our own volition in hardware is by implementing raw sockets. There has been a significant amount of work done in this area, much of it described in the raw socket API, which we referenced extensively [1], as well as other resources and tutorials [2, 3, 4].

This was done in C code, first by implementing the Address Resolution Protocol (ARP) [5]. The file included is the tcp_rawsck.c file. In order to implement TCP packets, we needed to implement resolution at the link layer. The ARP is a request and reply protocol that encapsulates the IP, which encapsulates the TCP layer below it. This is the arp.c file. This description is shown through the TCP stack diagram of Fig. 2.

Underneath the ARP implementation, TCP is implemented through the board with Linux running on it. Then, this was broken down into separate modules so that when implementing the hardware portion of the code, it would conceptually be transferable.

![TCP stack diagram](image)

Figure 2: TCP stack with the top-level connected to a server or application

The program Wireshark is used to visualize the TCP packets being sent and the subsequent responses received. A snapshot of the golden references in the form of this pattern of data transmission and reception is essentially taken.

c. Motivation for FPGA Integration

As mentioned above, low latency is very important for HFT. The first few players to execute orders may be the only ones able to profit from a given opportunity. Also some HFT strategies such as latency arbitrage depend upon the ability to access market data and execute orders faster than other investors.
In software-based HFT platforms, the incoming traffic data is transferred to memory and then CPU is interrupted to handle the application processing. After the processing is complete, the data is transferred back over the network. The interrupt-driven software stack, unpredictable transfers and cache misses make the network latencies higher and less predictable in software implementations. In contrast, in FPGA implementations, the data is directly available within the same clock cycle of its arrival time, naturally achieving low latency with minimal jitter [6]. In addition, FPGA can achieve higher throughput using its natural parallelism advantage. Multiple pipelines can be instantiated in parallel when data dependencies do not exist.

II. C Programming

a. Motivation

The motivation behind this portion of the project was to understand how TCP/IP protocol enables reliable communication, implement a TCP stack that bypasses the Linux Kernel, and verify implementation through comparison with Golden Model. Then, the larger goal was to be able to replicate the implementation on the hardware side.

Doing this portion of the project required considerable start-up effort and time. First, we had to understand how the lower level Kernel connection worked. Then, we had to understand how to interpret the TCP headers we were receiving and whether the response we were getting was expected. At first, we connected with a large search engine server. Then, we created our own Telnet server, which enabled us to use command line to connect to a remote server [7], and were able to establish connection and collect informative responses that way.
b. Structure

Figure 4: Bypassing the Linux Kernel using Raw Sockets

In our C implementation, to be able to bypass the Linux kernel and process packet at the basic level, we use `sd = socket(PF_PACKET, SOCK_RAW, htons(ETH_P_ALL))`. We also had issues with Linux Kernel sending spontaneous RST, which we solved using the command `sudo iptables -A OUTPUT -o wlan0 -p tcp --dport 52000 --tcp-flags RST RST -j DROP`. The connection is initiated by performing 3-handshake (SYN,SYN-ACK, ACK). and our program is able to handle several connections at once. Reliable data transfer is performed with appropriate ACK sending. The connection is closed by performing 3-handshake protocol (FIN, FIN-ACK, ACK). The user can switch from hardware implementation to software implementation while using same function calls. What is not included in the TCP implementation is: window-size advertising, management of retransmissions.

In the main function, first `tcp_new()` allocates memory for a `tcp_ctrl` structure where all the TCP information is stored, including source/destination IP/MAC address, port number, sequence number, acknowledgement number etc. Then in the `tcp_bind()` function, a socket descriptor is created to get source MAC address and maximum transmission unit (MTU). Next, in the `tcp_connect()` function, a connection is made to a URL. The URL is first resolved using `getaddrinfo()` to get the destination IP address, then ARP protocol is used to resolve the destination MAC address. After that the 3-way handshake is performed by sending SYN first, receiving SYNACK from the destination and the sending ACK to the destination.

After the 3-way handshake is done, a "GET / HTTP/1.1\r\n\r\n" is first sent to the destination using the `tcp_write()` function. Then `tcp_rcv()` will receive the incoming TCP packet
and update the sequence number and acknowledgement number accordingly. A second round of transmission/receiving cycle is done to transmit a whole file to the destination.

c. Software API and Golden Reference

The structure that holds the connection data is tcp_ctrl. Then, the function tcp_rawsck(void) calls the software API, which consists of tcp_new (allocates memory for a new connection, ie. all the header space), tcp_bind (connects to the system interface), tcp_connect (sets the address information), tcp_listen (essentially a combination of tcp_write and tcp_receive; their functions are self-explanatory from their names), and tcp_close (sends the fin/ack). These are the aforementioned modules that allow the process to be modularized for convenient integration in hardware.

The golden reference is a Raw Capture file of making the handshake, request, transmission, and recording of this result. Then, a second request and transmission is made, followed by a recording of this result. The results of this are produced in the GoldenRef file that is done manually. This was done with the main.c file, also included in section VII.

d. Testing (Wireshark)

![Figure 5: Screenshot of Wireshark showing successful TCP transmission](image)

We extensively used Wireshark to track packet sent and received. The golden reference is generated by sending http request to Google using Telnet, and comparison to Golden reference has been done manually. It can be seen clearly from Fig. 5 of the Wireshark capture that the successful establishment of TCP between our host (160.39.212.242) and the remote destination (74.125.228.211). First three packets are the 3-way handshake (SYN, SYN-ACK, ACK). Then a HTTP GET request is sent to the destination, then the destination answers with an ACK and subsequent TCP packets. Our host acknowledges each successful receiving of the TCP packets, with ACK number increment shown.
III. Modules

a. TCP Offloading Engine

The top-level files is named TOE.sv. It instantiates the modules that were made for the purposes of transmitting a packet to establish a connection, which will be described in this section. Specifically, these include Packetizer, Packet_builder, TOE_init, and RAM2. For its i/o signals, it connects to both the Avalon MM slave interface [8] as well as the Avalon ST source interface for the purposes of outputting the Ethernet frame.

The TCP Offloading Engine (TOE) is implemented by the module TOE_init.sv. The RAM_searcher is instantiated within this module. It takes in network header information from two sides, both from the network side for reception, and from the software side for transmission by the user. This means that it is maintaining the state of each piece of data for a connection. The information that is relevant is the header information of the source MAC address, the destination MAC address, the source IP address, the destination IP address, the source port number, and the destination port number. These six pieces of data are necessary for creating and ending a connection. In order to ensure that only one relevant connection is made per destination, the TOE includes a piece of componentry that double checks that there is no existing connection before creating a new one.

The status information that we keep to maintain state includes: Return, Processing, and Wait Request. When Wait Request is enabled, this means that there has been a request for a new connection to be made. This signal being set high means that we should start processing all the steps within the TOE module. When reply is set high, this means that the Processing has been completed. Then, when req_code is set during the Return state, we can tell whether an error condition has occurred (if a connection had already been made), which is checked for by the Ram Searcher module. Otherwise, a signal 120 means all the steps needed to be taken inside the TOE have been completed for now.
Fig. 6 shows a high level diagram of the TOE module, the input/output of the TOE module are done through the Avalon MM interface and the Avalon ST interface. The input to the TOE are clk, rst, write_data, write, read, chipselect and address. Output of the TOE is readdata.

Inside the TOE_init module there is a RAM_searcher module, which takes care of the current connection information such as source/destination IP/MAC/port, seq number, ack number etc. Those information are stored in an instantiated RAM module.
The TOE needs to keep track of each of the connections currently involving the host. These connections have all been initiated by the local host since the TOE is not able to handle connections requests coming from remote hosts through TCP packets with flag SYN.

This information is stored in the Connections RAM implemented by module RAM2.v. This RAM was created using Altera MegaWizzard and has the following characteristics:
- Dual-ported
- 2 cycle delay for reading data into the RAM
- 32-bit words and hence 32-bit wide bus
- Size of 256 32-bit words.

Connection information is organized by “records” inspired by C structs. One record contains all the data necessary for keeping track and building packets concerning one connection. A record requires a total of 10 words, which means that simultaneous connections can be handled. The fields
composing a record are indicated in Figure 7. They can all be accessed with a constant offset from the base address of the record. Some fields such as the MAC addresses require more than one word, while others need less than a full word. Padding is inserted to keep the data aligned.

c. State Diagrams

![State Diagram](image)

Figure 8: RAM_searcher top-level state diagram

Fig. 8 shows the top-level state diagram of the RAM_search module, which is implemented by `RAM_searcher.sv`. Initially the RAM_searcher is in idle state. If it gets a request (01 or 11), it will go to the state where it searches for a connection (searching) or tries to delete a connection (deleting). If it is currently in searching state, and a connection record is found, it will go to the error_found state; if no record is found, then a new connection record is inserted to the RAM (inserting state). If currently in deleting state, when deletion is done, it will go to the return_success state. Finally all the states will go back to the idle state if req equals 00.
Figure 9 (a): RAM_searcher checking for an existing connection while 9 (b) shows the insertion of connection

This more granular image (Fig. 9) shows checking if there is already a connection. RAM addr is incremented if we are progressing to the next state. chk_equal means found existing connection, and it would return error. If an existing connection is not found, then the base_address is set as the address to write into. Finally, the inserting of connection is shown.
Figure 10: Packetizer state diagram, where eth_ready is logic indicating incoming ethernet, to_send is the packet to be filled through a combinatorial loop, and next_last is also control logic, set high when the process is grabbing the last set of header data.
Figure 11: This is the conceptual diagram for the Packet_builder state machine. Top illustrates high level signals while bottom illustrates the states for retrieving relevant information from the RAM (with eight total states moving sequentially). Naming convention for each state follows directly with that described in the file. The design is similar to loading the data, although this time we’re retrieving it and loading it into our packet.
d. Timing Diagrams

![Timing Diagram](image.png)

Figure 12: Timing diagram of the TOE when a connection is made successfully. This is a conceptual drawing for the hardware portion of the project startup, and we maintained fairly close to this timing.

Fig. 12 shows the timing diagram of the TOE module. At the clock rising edge, a req is set high meaning the IP user wants to request a new connection. Then open_con is set low meaning the TOE is currently checking if there is an existing connection and is not currently accepting new request. If there is no existing connection, after a few clock cycles the status would be set high meaning the check is done and open_con is high meaning the TOE is ready for req again.

e. Modelsim

Testbenches were run in order to ensure that the separate modules that we had written worked as we previously laid out. This was done on a step-by-step basis, with reference to tutorials [9], to ensure that we did not get bogged down in the coding. The following is the testbench that combines both the Initial TOE implementation and the RAM searcher implementation, which verifies that the connection request has not already been established before.

As such, you can see that the data is indeed being put into the RAM on the write and positive clock cycle. Then, the data is being read out. This illustrates a successful connection made.
Figure 13: ModelSim simulation of simplified TOE. On a Read, data is checked through the RAM_searcher, and with no overlapping connection, on a Write data is then written into the RAM, seen through writedata.

Figure 14: Result from running Testbench1.sv, which requests the creation of a new connection. We expect the connection to go through, returning ID 120. The state machine should be in the state WAIT_RQ, as can be seen above.
Figure 15: Modified testbench1.sv, with states printed out

Figure 16: Result from running Testbench2.sv, which requests the creation of the same new connection twice in a row. We expect the first connection request to go through, returning ID 120, and the second to be rejected, with ERR_FOUND returned.
Figure 17(a) The timing diagram of Modelsim to go with the state diagram of Fig. 10. The testbench PB_testbench.sv, which sets ram_in once and sets wren on for a given duration, is included in the Source Code section. Its functionality is the streaming in of data in the RAM after running the RAM_searcher and (b) zoomed in of the important locations, with relevant labels.

f. RAM

The RAM is created by the Megawizard Function. Most optimally, we would use a bidirectional RAM to be able to handle both incoming data, and read out data simultaneously. Any synchrony issues would be minimized by checking first the valid bit of each entry of the RAM before doing any reading/writing to it. The altera_mf_ver library needs to be included in the path in order to run this module in verification stage.

g. Packet Builder

The functionality of the packet builder module, which is implemented by Packet_builder.sv, is to create a TCP packet, inclusive of a payload and a header. The total length of the header is 54 bytes, since TCP options are not used. The way in which the Packet_builder is implemented is through both a high level and second level automata. The high level automata consists of four main states. These are: s_IDLE, s_REQ, s_CHECK, and s_WAIT. For s_IDLE, if wren is high from the RAM search, we go into this state. From here we continue to s_REQ, which grabs the valid bit from the incoming RAM. s_CHECK checks the valid bit. s_WAIT sets the valid bit and then sets the transfer valid bit as the valid bit.

Then, for the second level automata, there are fourteen states: o_IDLE, o_REQ, o_WAIT, o_CPY_SEQ, o_CPY_ACK, o_CPY_IP_SRC, o_CPY_IP_DST, o_CPY_MAC_SRC1, o_CPY_MAC_DST1, o_CPY_MAC_SRC2, o_CPY_MAC_DST2, o_CPY_PORTS, o_STALL, and o_DONE. When it is idle, we go into the case loop. When it is o_CPY_ACK to o_CPY_PORTS, data is grabbed from RAM_in and put into the header data storage. Then, when it is done, the valid-bit is 0 (a non-state).
The important triggers from high to second level automata is the valid_bit_high. When the current state is done, then RAM_stored_header_data is placed into packet. This means that all the header regions are properly filled out according to the spec. This packet is transmitted out through the Avalon-ST in top level file. It has the opposite functionality as the packetizer, described below, which is essentially a packet decomposer.

**h. Ethernet packetizer**

The function of the packetizer, implemented by `Packetizer.sv`, is to parse the incoming Ethernet headers [10], which include IP, functioning under IPV4, and TCP. This is performed through a state diagram. Since the Avalon MM bus only takes in 32 bits at a time, this is what we must work with on each state. Therefore, we iterate through every state of 32-bit parsings and perform perfunctory operations in the middle where the packet might be dropped. For instance, if the valid bit is not set high inside the IP header, then the pocket should be dropped. Immediately after this value is read in, the check is done within the next state such that if it is an invalid packet, no further unnecessary parsing is done.

The conceptual idea of the state diagram for the packetizer is shown in Fig. 10.

![State Diagram](figure18.jpg)

Figure 18: JTAG file for Packetizer.sv, showing that when a faulty connection is made, it parses, then exits.
IV. Control between the modules

Figure 19: Interfaces between different modules that we implemented

a. Avalon MM interface

The data that Avalon MM takes from its peripherals are usually six input/outputs of: the register writedata, its control logic write, clk, the register readdata, and its control logic read. The states that we store in the Avalon MM register that we create within writedata are req_code, id_in, reply, ip_src, ip_dst, mac_src, mac_dst, port_src, and port_dst. New request and kill request come from different sides of the isle; new request comes from the user and kill request comes from the destination. Likewise with new ID and kill ID, typically. Thus, the control bits that are required by the initial connection are as follows: new request (self explanatory), error (when there is a connection already made, other instances where the connection is not successfully established), done (when the connection process is completed), and new ID (to pass out as the address to where data for this connection is stored inside the RAM.

Specifically, according to our spec, the address consists of [7:0]. The reply itself only consists of [7:1], while the other bit is ID if a 0 is returned. Otherwise, if the 1 is appended, this indicates an error.
Figure 20: High level diagram of the entire schematic and where the program fits in
b. Avalon ST

Data that is being transmitted is accepted through Avalon ST wires. There are two kinds of data. First, there is the data that we are receiving. Then, there is the header information that we are accepting from the network. These come in through the Avalon ST wires.

The Avalon ST to MM file is done using the top level Qsys file, where inputs and outputs are instantiated.

c. TCL

The TCL file implementing JTAG goes one step beyond Modelsim to show the synthesized connections working, with the incoming request being processed, and when a same connection is requested a second time, it correctly returns ERR_FOUND. This second connection is then deleted successfully, attached in Fig. 22.
V. Lessons Learned

1. Doing the implementation of the TCP connection using C code required a considerable amount of implementation. There were various online tutorials that we were referencing, but it still required extensive amounts of understanding and interconnections.

   a. During the implementation of the C code, we tried for some time to get a TCP syn/ack. The issue lay in having every part of the header be the exact correct length. Otherwise, the checksum would not be verified.
   b. There as a reset command that would be sent in the connection. This could be solved by turning off the IP tables.
   c. Begin integration between modules as soon as possible. There were certain things that we laid out in our initial milestone which, when re-evaluated during the mid-milestone,
we realized was not relevant or reasonable to do. For instance, we decided to proceed with the Golden Reference over the PCAP file [12], because the PCAP file would be too stringent for the variability of TCP protocol.

d. A note on the variability of TCP: this took up much of our project- understanding the functionality of TCP in the software side.

e. For the packetbuilder, some issue ran into was that the valid_bit_high was being used as both a trigger for the second-level automata and as the initial state that causes you to go into the virtual loop in the high-level. This was solved by setting valid_bit_tx to valid_bit_high, and using that as the trigger for the second-level automata, while proceeding to do the same thing as before with valid_bit_high.

VI. Conclusion

We show the software implementation of successful request for connection, with raw socket API, and Wireshark verification, for hardware implementation of TCP processing. Additionally, we show hardware implementation of integrated modules for starting a connection to send a syn packet. Verification was done using ModelSim, and the relevant modules include: TOE_init, RAM_searcher, RAM, packet_builder, and packetizer, with the toplevel file as TOE. We include the testbenches and diagrams that we drew and implemented for checking purposes.

References

Source Code

This section includes a portion of the software code as well as the hardware code. The latter part of the hardware includes just a portion of the testbench that we had written.

#include "tcp_rawsck.h"
#include "arp.h"
#include <string.h>

#define SYN 0x02
#define ACK 0x10
#define SYNACK 0x12
#define FINACK 0x11

char *allocate_strmem (int);
uint8_t *allocate_ustrmem (int);
int *allocate_intmem (int);

uint16_t checksum (uint16_t *, int);
uint16_t tcp4_checksum(struct ip, struct tcphdr, uint8_t *, int);
uint16_t tcp2_checksum(struct ip, struct tcphdr);

// Filling packets
int fill_iphdr(struct tcp_ctrl *, int);
int fill_tcphdr(struct tcp_ctrl *, uint8_t flags, int len);
int fill_ethhdr(struct tcp_ctrl *, int len);

// Establishing connection
void sd_ARP_rq(struct tcp_ctrl *);
void rcv_ARP_asw(struct tcp_ctrl *);
void sd_SYN_pck(struct tcp_ctrl *);
int rcv_SYNACK_pck(struct tcp_ctrl *);
int rcv_ACK_pck(struct tcp_ctrl *);
int sd_FINACK_pck(struct tcp_ctrl *);
int rcv_FINACK_pck(struct tcp_ctrl *);
void sd_ACK_pck(struct tcp_ctrl *, int);

struct tcp_ctrl *tcp_new_rawsck(void) {
  printf("Entering : tcp_new()\n");

  struct tcp_ctrl *tcp_ctrl = malloc(sizeof(struct tcp_ctrl));
  if (tcp_ctrl == NULL) {
    perror("malloc() failed");
    exit(EXIT_FAILURE);
  }

  tcp_ctrl->seq = random();
  tcp_ctrl->rcv_ack = 0;
  tcp_ctrl->mtu = 0;
  tcp_ctrl->state = CLOSED;
  // Allocate memory for various arrays.
  tcp_ctrl->iphdr = (struct ip *)malloc(sizeof(struct ip));
  if (tcp_ctrl->iphdr == NULL) {
    perror("Memory Allocation for tcphdr failed");
    exit(EXIT_FAILURE);
  }
  tcp_ctrl->tcphdr = (struct tcphdr *)malloc(sizeof(struct tcphdr));
  if (tcp_ctrl->tcphdr == NULL) {
    perror("Memory Allocation for tcphdr failed");
    exit(EXIT_FAILURE);
  }
  tcp_ctrl->src_mac = allocate_ustrmem(6);
  tcp_ctrl->dst_mac = allocate_ustrmem(6);
  tcp_ctrl->ether_frame = allocate_ustrmem(IP_MAXPACKET);
  tcp_ctrl->sdbuffer = allocate_ustrmem(IP_MAXPACKET);
  tcp_ctrl->interface = allocate_strmem(40);
  tcp_ctrl->target = allocate_strmem(40);
  tcp_ctrl->src_ip = allocate_strmem(INET_ADDRSTRLEN);
  tcp_ctrl->dst_ip = allocate_strmem(INET_ADDRSTRLEN);
tcp_ctrl -> ip_flags = allocate_intmem (4);
tcp_ctrl -> tcp_flags = allocate_intmem (8);

if ((tcp_ctrl->sd = socket (PF_PACKET, SOCK_RAW, htons(ETH_P_ALL))) < 0) {
    perror ("socket() failed");
    exit (EXIT_FAILURE);
}

printf("Exiting : tcp_new()
");
return tcp_ctrl;
}

int tcp_bind_rawsck(struct tcp_ctrl* tcp_ctrl, char *ip_addr, uint16_t sport, char *interface)
{
    printf("Entering : tcp_bind\n");

    int sd;
    struct ifreq ifr;

    strcpy (tcp_ctrl->src_ip, ip_addr);
    strcpy (tcp_ctrl->interface, interface);
    tcp_ctrl -> sport = sport;

    if ((sd = socket (AF_INET, SOCK_RAW, IPPROTO_RAW)) < 0) {
        perror ("socket() failed to get socket descriptor for using ioctl()");
        exit (EXIT_FAILURE);
    }

    memset(&ifr, 0, sizeof(struct ifreq));
    snprintf(ifr.ifr_name, sizeof (ifr.ifr_name), "%s", interface);
    if ((ioctl (sd, SIOCGIFHWADDR, &ifr)) < 0) {
        perror ("ioctl() failed to get source MAC address ");
        return (EXIT_FAILURE);
    }
    memcpy(tcp_ctrl->src_mac, ifr.ifr_hwaddr.sa_data, 6 * sizeof(uint8_t));

    // Find interface index from interface name and store index in
    // struct sockaddr_ll device, which will be used as an argument of sendto().
memset (&(tcp_ctrl->device), 0, sizeof (struct sockaddr_ll));
if (((tcp_ctrl->device).sll_ifindex = if_nametoindex (tcp_ctrl->interface)) == 0) {
    perror ("if_nametoindex() failed to obtain interface index ");
    exit (EXIT_FAILURE);
}
printf ("Index for interface %s is %i\n", tcp_ctrl->interface, (tcp_ctrl->device).sll_ifindex);

// Use ioctl() to get interface maximum transmission unit (MTU).
memset (&ifr, 0, sizeof (ifr));
strcpy (ifr.ifr_name, interface);
if (ioctl (sd, SIOCGIFMTU, &ifr) < 0) {
    perror ("ioctl() failed to get MTU ");
    return (EXIT_FAILURE);
}
tcp_ctrl->mtu = ifr.ifr_mtu;
printf ("Current MTU of interface %s is: %i\n", interface, tcp_ctrl->mtu);

close(sd);

printf("Exiting : tcp_bind()\n");
return 0;
}

int tcp_connect_rawsck(struct tcp_ctrl *tcp_ctrl, char* url) {
    printf("Entering : tcp_connect()\n");
    
    int status;

    struct addrinfo hints, *res;
    struct sockaddr_in *ipv4;
    void *tmp;

    strcpy(tcp_ctrl->target, url);
tcp_ctrl->dport = 80;

    // Fill out hints for getaddrinfo()
    memset(&hints, 0, sizeof(struct addrinfo));
hints.ai_family = AF_INET;
hints.ai_socktype = SOCK_STREAM;
hints.ai_flags = hints.ai_flags | AI_CANONNAME;

// Resolve target using getaddrinfo()
if ((status = getaddrinfo(tcp_ctrl->target, NULL, &hints, &res)) != 0) {
    fprintf(stderr, "getaddrinfo() failed: %s
", gai_strerror(status));
    exit(EXIT_FAILURE);
}
ipv4 = (struct sockaddr_in *) res->ai_addr;
tmp = &(ipv4 -> sin_addr);
strcpy(tcp_ctrl->dst_ip, inet_ntoa(ipv4->sin_addr));
freeaddrinfo(res);

// Fill out sockaddr_ll.
(tcp_ctrl->device).sll_family = AF_PACKET;
memcpy(((tcp_ctrl->device).sll_addr, tcp_ctrl -> src_mac, 6 * sizeof (uint8_t));
(tcp_ctrl->device).sll_halen = htons(6);

sd_ARP_rq(tcp_ctrl);
rcv_ARP_asw(tcp_ctrl);
sd_SYN_pck(tcp_ctrl);
// Modify function synack
int ack = rcv_SYNACK_pck(tcp_ctrl);
tcp_ctrl -> rcv_ack = ack;
printf("ACK value after receiving SYNACK : %u\n", tcp_ctrl -> rcv_ack);
sd_ACK_pck(tcp_ctrl, tcp_ctrl -> rcv_ack);

printf("Exiting tcp_connect()\n");

return 0;
}

int tcp_close_rawsck(struct tcp_ctrl *tcp_ctrl) {
    sd_FINACK_pck(tcp_ctrl);
    rcv_FINACK_pck(tcp_ctrl);
    sd_FINACK_pck(tcp_ctrl);
tcp_ctrl->seq++;
tcp_ctrl->rcv_ack++;
sd_ACK_pck(tcp_ctrl, tcp_ctrl -> rcv_ack);
rcv_ACK_pck(tcp_ctrl);
sd_ACK_pck(tcp_ctrl, tcp_ctrl -> rcv_ack);
return 0;
}

int tcp_write_rawsck(struct tcp_ctrl *tcp_ctrl, void *data, int len) {

printf("Entering : tcp_write()\n");

int status, i, frame_length;
int max_payload = tcp_ctrl->mtu - TCP_HDRLEN - IP4_HDRLEN - ETH_HDRLEN;
printf("max_payload : %d\n", max_payload);
if (len > max_payload) {
    perror("Request too long");
    exit(EXIT_FAILURE);
}
memcpy(tcp_ctrl->sdbuffer, (uint8_t *) data, len);
fill_iphdr(tcp_ctrl, len);
fill_tcphdr(tcp_ctrl, ACK, len);
fill_ethhdr(tcp_ctrl, len);

// Send ethernet frame to socket.
frame_length = ETH_HDRLEN + IP4_HDRLEN + TCP_HDRLEN + len;
int success;
if (((success = sendto (tcp_ctrl->sd, tcp_ctrl->ether_frame, frame_length, 0, (struct sockaddr *) &(tcp_ctrl->device), sizeof (struct sockaddr_ll))) <= 0) {
    perror ("sendto() failed");
    exit (EXIT_FAILURE);
}
tcp_ctrl -> seq += len;
//if (tcp_ctrl -> state == SYN_SENT) tcp_ctrl -> state = OPEN;

rcv_ACK_pck(tcp_ctrl);

printf("Exiting : tcp_write()\n");
int tcp_rcv_rawsock(struct tcp_ctrl *tcp_ctrl, uint8_t *data, int max_len){
    printf("Entering : tcp_rcv()\n");
    int len = 0;
    int bytes, payload;

    struct tcphdr *tcphdr;
    tcphdr = (struct tcphdr *) (tcp_ctrl -> ether_frame + ETH_HDRLEN + IP4_HDRLEN);
    while ( len < max_len ) {
        if ((bytes = recv(tcp_ctrl -> sd, tcp_ctrl -> ether_frame, IP_MAXPACKET, 0)) < 0) {
            printf("ERROR");
            perror("recv() failed");
            exit(EXIT_FAILURE);
        } else {
            // Filter TCP packets
            if (((tcp_ctrl->ether_frame[12]) << 8) + tcp_ctrl -> ether_frame[13]) == ETH_P_IP) {
                printf("th_seq : %u\n", ntohl(tcphdr -> th_seq));
                printf("rcv_ack : %u\n", tcp_ctrl -> rcv_ack);
                if (((ntohl(tcphdr -> th_seq)) == tcp_ctrl -> rcv_ack)) {
                    payload = bytes - TCP_HDRLEN - IP4_HDRLEN - ETH_HDRLEN;
                    printf("payload : %d\n", payload);
                    if( payload < 536 ) {
                        memcpy(data + len, (uint8_t *) tcphdr + TCP_HDRLEN, payload);
                        len += payload;
                        tcp_ctrl -> rcv_ack += (payload);
                        sd_ACK_pck(tcp_ctrl, tcp_ctrl -> rcv_ack);
                        printf("Breaking\n");
                        break;
                    }
                    memcpy(data + len, (uint8_t *) tcphdr + TCP_HDRLEN, payload);
                    printf("len : %d\n", len);
                    len += payload;
                    tcp_ctrl -> rcv_ack += payload;
                    sd_ACK_pck(tcp_ctrl, tcp_ctrl -> rcv_ack);
                }
            }
        }
    }
    return success;
}


```c
else {
    printf("DROP CONNECTION\n");
    return -1;
}
}
}
}
if (len == max_len) {
    perror("Buffer complet");
    exit(EXIT_FAILURE);
}
printf("Exiting : tcp_rcv()\n");
return len;

fill_iphdr(struct tcp_ctrl *tcp_ctrl, int len) {
    struct ip *iphdr = tcp_ctrl -> iphdr;
    int ip_flags[4];
    int status;
    // IPV4 header length (4 bits): Number of 32-bit words in header = 5
    iphdr -> ip_hl = IP4_HDRLEN / sizeof(uint32_t);
    // Internet Protocol version (4 bits): IPv4
    iphdr -> ip_v = 4;
    // Type of service (8 bits)
    iphdr -> ip_tos = 0;
    // Total length of datagram (16 bits): IP header + TCP header
    iphdr -> ip_len = htons (IP4_HDRLEN + TCP_HDRLEN + len);
    // ID sequence number (16 bits): unused, since single datagram
    iphdr -> ip_id = htons (0);
    // Flags, and Fragmentation offset (3, 13 bits): 0 since single datagram
    // Zero (1 bit)
    ip_flags[0] = 0;
    // Do not fragment flag (1 bit)
    ip_flags[1] = 0;
    // More fragments following flag (1 bit)
    ip_flags[2] = 0;
    // Fragmentation offset (13 bits)
    ip_flags[3] = 0;
```
iphdr -> ip_off = htons ((ip_flags[0] << 15)
    + (ip_flags[1] << 14)
    + (ip_flags[2] << 13)
    + ip_flags[3]);

    // Time-to-Live (8 bits): default to maximum value
    iphdr -> ip_ttl = 255;
    // Transport layer protocol (8 bits): 6 for TCP
    iphdr -> ip_p = IPPROTO_TCP;

    // Source IPv4 address (32 bits)
    if ((status = inet_pton(AF_INET, tcp_ctrl -> src_ip, &(iphdr -> ip_src))) != 1) {
        fprintf(stderr, "inet_pton() failed 1.\nError message: %s", strerror(status));
        exit(EXIT_FAILURE);
    }

    // Destination IPv4 address (32 bits)
    if ((status = inet_pton(AF_INET, tcp_ctrl -> dst_ip, &(iphdr -> ip_dst))) != 1) {
        fprintf(stderr, "inet_pton() failed 2.\nError message: %s", strerror(status));
        exit(EXIT_FAILURE);
    }

    // IPv4 header checksum (16 bits): set to 0 when calculating checksum
    iphdr -> ip_sum = 0;
    iphdr -> ip_sum = checksum ((uint16_t *) iphdr, IP4_HDRLEN);

    return 0;
}

int fill_tcphdr(struct tcp_ctrl *tcp_ctrl, uint8_t flags, int len) {

    // TCP header
    struct tcphdr *tcphdr = tcp_ctrl -> tcphdr;
    int tcp_flags[8];

    // Source port number (16 bits)
    tcphdr -> th_sport = htons (tcp_ctrl -> sport);
    // Destination port number (16 bits)
    tcphdr -> th_dport = htons (tcp_ctrl -> dport);
    // Sequence number (32 bits)
    tcphdr -> th_seq = htonl(tcp_ctrl -> seq);
// Acknowledgement number (32 bits): 0 in first packet of SYN/ACK process
// Isolate the ACK flag and put 0 instead if the ACK flag is not on
if (flags & 0x10) { tcphdr -> th_ack = htonl (tcp_ctrl -> rcv_ack); }
else { tcphdr -> th_ack = htonl (0); }
// Reserved (4 bits): should be 0
tcphdr -> th_x2 = 0;
// Data offset (4 bits): size of TCP header in 32-bit words
tcphdr -> th_off = TCP_HDRLEN / 4;
// Flags
tcphdr -> th_flags = flags;
// Window size (16 bits)
tcphdr -> th_win = htons (14600);
// Urgent pointer (16 bits): 0 (only valid if URG flag is set)
tcphdr -> th_Urp = htons (0);
// TCP checksum (16 bits)
tcphdr -> th_sum = 0;
tcphdr -> th_sum = tcp4_checksum (*(tcp_ctrl -> iphdr), *(tcp_ctrl -> tcp hdr), tcp_ctrl
-> sdbuf, len);

return 0;

int fill_ethhdr(struct tcp_ctrl *tcp_ctrl, int len) {

    // Fill out ethernet frame header.
    // Destination and Source MAC addresses
    memcpy (tcp_ctrl -> ether_frame, tcp_ctrl -> dst_mac, 6 * sizeof (uint8_t));
    memcpy (tcp_ctrl -> ether_frame + 6, tcp_ctrl -> src_mac, 6 * sizeof (uint8_t));
    // Next is ethernet type code (ETH_P_IP for IPv4).
    // http://www.iana.org/assignments/ethernet-numbers
    tcp_ctrl -> ether_frame[12] = ETH_P_IP / 256;
    tcp_ctrl -> ether_frame[13] = ETH_P_IP % 256;
    // Next is ethernet frame data (IPv4 header + TCP header).
    // IPv4 header
    memcpy (tcp_ctrl -> ether_frame + ETH_HDRLEN, tcp_ctrl -> iphdr, IP4_HDRLEN * sizeof
    (uint8_t));
    // TCP header
    memcpy (tcp_ctrl -> ether_frame + ETH_HDRLEN + IP4_HDRLEN, tcp_ctrl -> tcp hdr,
    TCP_HDRLEN * sizeof (uint8_t));
memcpy (tcp_ctrl->ether_frame + ETH_HDRLEN + IP4_HDRLEN + TCP_HDRLEN, tcp_ctrl->sdbuf, len * sizeof(uint8_t));

}  

int sd_FINACK_pck(struct tcp_ctrl *tcp_ctrl) {

    printf("Entering : sd_FINACK_pck()\n");

    int status, i, frame_length, bytes;

    fill_iphdr(tcp_ctrl, 0);
    fill_tcphdr(tcp_ctrl, FINACK, 0);
    fill_ethhdr(tcp_ctrl, 0);

    // Send ethernet frame to socket.
    frame_length = ETH_HDRLEN + IP4_HDRLEN + TCP_HDRLEN;
    if ((bytes = sendto((tcp_ctrl->sd, tcp_ctrl->ether_frame, frame_length, 0, (struct sockaddr *) &((tcp_ctrl->device), sizeof(struct sockaddr_ll))) <= 0) {
        perror("sendto() failed");
        exit(EXIT_FAILURE);
    }

    printf("Exiting : sd_FINACK_pck()\n");

}  

void sd_SYN_pck(struct tcp_ctrl *tcp_ctrl) {

    printf("Entering : sd_SYN_pck()\n");

    int status, i, frame_length, bytes;

    fill_iphdr(tcp_ctrl, 0);
    fill_tcphdr(tcp_ctrl, SYN, 0);
    fill_ethhdr(tcp_ctrl, 0);

    // Send ethernet frame to socket.
// Ethernet frame length = ethernet header (MAC + MAC + ethernet type) + ethernet data (IP header + TCP header)
frame_length = ETH_HDRLEN + IP4_HDRLEN + TCP_HDRLEN;
if ((bytes = sendto (tcp_ctrl->sd, tcp_ctrl->ether_frame, frame_length, 0, (struct sockaddr *) &tcp_ctrl->device), sizeof (struct sockaddr_ll)) <= 0) {
    perror ("sendto() failed");
    exit (EXIT_FAILURE);
}

//tcp_ctrl -> state = SYN_SENT;
printf("Exiting : sd_SYN_pck()\n");

int rcv_ACK_pck(struct tcp_ctrl *tcp_ctrl) {
    printf("Entering : rcv_ACK_pck()\n");

    int status;
    struct tcphdr *tcphdr;
    tcphdr= (struct tcphdr *) (tcp_ctrl -> ether_frame + ETH_HDRLEN + IP4_HDRLEN);
    struct ip *ip;
    ip = (struct ip *) (tcp_ctrl -> ether_frame + ETH_HDRLEN);
    do {
        if ((status = recv (tcp_ctrl -> sd, tcp_ctrl -> ether_frame, IP_MAXPACKET, 0)) < 0) {
            if (errno == EINTR) {
                memset (tcp_ctrl -> ether_frame, 0, IP_MAXPACKET * sizeof (uint8_t));
                continue; // Something weird happened, but let's try again.
            } else {
                perror ("recv() failed:");
                exit (EXIT_FAILURE);
            }
        }
    } while (((((tcp_ctrl -> ether_frame[12]) << 8) + tcp_ctrl -> ether_frame[13]) != ETH_P_IP)
    ||(strcmp(inet_ntoa(ip -> ip_src), tcp_ctrl -> dst_ip) != 0)
    ||(strcmp(inet_ntoa(ip -> ip_dst), tcp_ctrl -> src_ip) != 0) // Maybe we can remove this condition
    ||(memcmp(tcp_ctrl -> ether_frame, tcp_ctrl -> src_mac, 6) != 0) // In case we have several MAC (possible ?)
    ||(tcphdr->th_flags != ACK));
    printf("Exiting : rcv_ACK_pck()\n");
    return tcp_ctrl -> rcv_ack + status;
int rcv_FINACK_pck(struct tcp_ctrl *tcp_ctrl) {

  printf("Entering : rcv_FINACK_pck()\n");

  int status;
  struct tcphdr *tcphdr;
  tcphdr = (struct tcphdr *) (tcp_ctrl->ether_frame + ETH_HDRLEN + IP4_HDRLEN);
  struct ip *ip;
  ip = (struct ip *) (tcp_ctrl->ether_frame + ETH_HDRLEN);
  do {
    if ((status = recv(tcp_ctrl->sd, tcp_ctrl->ether_frame, IP_MAXPACKET, 0)) < 0) {
      if (errno == EINTR) {
        memset(tcp_ctrl->ether_frame, 0, IP_MAXPACKET * sizeof(uint8_t));
        continue; // something weird happened, but let's try again.
      } else {
        perror("recv() failed:");
        exit(EXIT_FAILURE);
      }
    }
  }

  printf("Exiting : rcv_FINACK_pck()\n");
  return 0;
}

int rcv_SYNACK_pck(struct tcp_ctrl *tcp_ctrl) {

  printf("Entering : rcv_SYNACK_pck()\n");
int status;
struct tcphdr *tcphdr;
tcphdr = (struct tcphdr *) (tcp_ctrl -> ether_frame + ETH_HDRLEN + IP4_HDRLEN);
struct ip *ip;
ip = (struct ip *) (tcp_ctrl -> ether_frame + ETH_HDRLEN);
while (((((tcp_ctrl->ether_frame[12]) << 8) + tcp_ctrl->ether_frame[13]) != ETH_P_IP) 
|| (strcmp(inet_ntoa(ip->ip_src), tcp_ctrl->dst_ip) != 0) 
|| (strcmp(inet_ntoa(ip->ip_dst), tcp_ctrl->src_ip) != 0))
// In case we have several IP on the same machine
|| (memcmp(tcp_ctrl->ether_frame, tcp_ctrl->src_mac, 6) != 0) 
// In case we have several MAC (possible ?)
|| (tcphdr -> th_flags != SYNACK)) {

    if ((status = recv (tcp_ctrl->sd, tcp_ctrl->ether_frame, IP_MAXPACKET, 0)) < 0) {
        if (errno == EINTR) {
            memset (tcp_ctrl->ether_frame, 0, IP_MAXPACKET * sizeof (uint8_t));
            continue; // Something weird happened, but let's try again.
        } else {
            perror ("recv() failed: ");
            exit (EXIT_FAILURE);
        }
    }
}
tcp_ctrl->seq++;
return ntohl(tcphdr->th_seq) + 1;

void sd_ACK_pck(struct tcp_ctrl *tcp_ctrl, int ack) {
    int status, i, frame_length, bytes;

    fill_iphdr(tcp_ctrl, 0);
    fill_tcphdr(tcp_ctrl, ACK, 0);
    fill_ethhdr(tcp_ctrl, 0);

    // Send ethernet frame to socket.
    frame_length = ETH_HDRLEN + IP4_HDRLEN + TCP_HDRLEN;
if ((bytes = sendto (tcp_ctrl -> sd, tcp_ctrl -> ether_frame, frame_length, 0, (struct sockaddr *) & (tcp_ctrl -> device), sizeof (struct sockaddr_ll)) <= 0) {
    perror ("sendto() failed");
    exit (EXIT_FAILURE);
}
}

void sd_ARP_rq(struct tcp_ctrl *tcp_ctrl) {

    printf("Entering : sd_ARP_rq()\n");
    int status;
    arp_hdr arphdr;

    // Set destination MAC address: broadcast address
    memset (tcp_ctrl -> dst_mac, 0xff, 6 * sizeof (uint8_t));

    // Fill ARP header
    // Hardware type (16 bits): 1 for ethernet
    arphdr.htype = htons (1);

    // Protocol type (16 bits): 2048 for IP
    arphdr.ptype = htons (ETH_P_IP);

    // Hardware address Length (8 bits): 6 bytes for MAC address
    arphdr.hlen = 6;

    // Protocol address Length (8 bits): 4 bytes for IPv4 address
    arphdr.plen = 4;

    // OpCode: 1 for ARP request
    arphdr.opcode = htons (ARPOP_REQUEST);

    // Sender hardware address (48 bits): MAC address
    memcpy (arp hdr .sender_mac, tcp_ctrl -> src_mac, 6 * sizeof (uint8_t));

    // Sender protocol address (32 bits)
    // See getaddrinfo() resolution of src_ip.
// Target hardware address (48 bits): zero, since we don't know it yet.
memset(arphdr.target_mac, 0, 6 * sizeof (uint8_t));

// Source IP address
if ((status = inet_pton(AF_INET, tcp_ctrl->src_ip, arphdr.sender_ip)) != 1) {
    fprintf(stderr, "inet_pton() source IP address.\nError message: %s", strerror(status));
    exit(EXIT_FAILURE);
}

// Fill Ethernet header
int frame_length, bytes;

// Ethernet frame length = ethernet header (MAC + MAC + ethernet type) + ethernet data
// (ARP header)
frame_length = ETH_HDRLEN + ARP_HDRLEN;

// Destination and Source MAC addresses
memcpy (tcp_ctrl->ether_frame, tcp_ctrl->dst_mac, 6 * sizeof (uint8_t));
memcpy (tcp_ctrl->ether_frame + 6, tcp_ctrl->src_mac, 6 * sizeof (uint8_t));

// Next is ethernet type code (ETH_P_ARP for ARP).
// http://www.iana.org/assignments/ethernet-numbers
tcp_ctrl->ether_frame[12] = ETH_P_ARP / 256;
tcp_ctrl->ether_frame[13] = ETH_P_ARP % 256;

// Next is ethernet frame data (ARP header).

// ARP header
memcpy (tcp_ctrl->ether_frame + ETH_HDRLEN, &arp hdr, ARP_HDRLEN * sizeof (uint8_t));

// Send ethernet frame to socket.
if ((bytes = sendto(tcp_ctrl->sd, tcp_ctrl->ether_frame, frame_length, 0, (struct sockaddr *) &tcp_ctrl->device), sizeof (struct sockaddr_ll)) <= 0) {
    perror("sendto() failed");
    exit(EXIT_FAILURE);
}
printf("Exiting : sd_ARP_rq()\n");
}

void rcv_ARP_asw(struct tcp_ctrl* tcp_ctrl) {

  // Listen for incoming ethernet frame from socket sd.
  // We expect an ARP ethernet frame of the form:
  //   MAC (6 bytes) + MAC (6 bytes) + ethernet type (2 bytes)
  //     + ethernet data (ARP header) (28 bytes)
  // Keep at it until we get an ARP reply.

printf("Entering : rcv_ARP_asw()\n");

int status, i;
arp_hdr *arphdr;

arphdr = (arp_hdr *) (tcp_ctrl->ether_frame + ETH_HDRLEN);

while ((((tcp_ctrl->ether_frame[12] << 8) + tcp_ctrl->ether_frame[13]) != ETH_P_ARP) || (ntohs(arphdr->opcode) != ARPOP_REPLY)) {
  if ((status = recv (tcp_ctrl->sd, tcp_ctrl->ether_frame, IP_MAXPACKET, 0)) < 0) {
    if (errno == EINTR) {
      memset (tcp_ctrl->ether_frame, 0, IP_MAXPACKET * sizeof (uint8_t));
      continue; // Something weird happened, but let's try again.
    } else {
      perror ("recv() failed: ");
      exit (EXIT_FAILURE);
    }
  }
}

for (i = 0; i < 6; i++) tcp_ctrl->dst_mac[i]=arphdr->sender_mac[i];
for (i = 0; i < 6; i++) printf("%02x:", tcp_ctrl->dst_mac[i]);
printf("\n");
// Allocate memory for an array of chars.
char *allocate_strmem (int len)
{
  void *tmp;

  if (len <= 0) {
    fprintf(stderr, "ERROR: Cannot allocate memory because len = %i in allocate_strmem().\n", len);
    exit(EXIT_FAILURE);
  }

  tmp = (char *) malloc (len * sizeof (char));
  if (tmp != NULL) {
    memset(tmp, 0, len * sizeof (char));
    return (tmp);
  } else {
    fprintf(stderr, "ERROR: Cannot allocate memory for array allocate_strmem().\n");
    exit(EXIT_FAILURE);
  }
}

// Allocate memory for an array of unsigned chars.
uint8_t *allocate_ustrmem (int len)
{
  void *tmp;

  if (len <= 0) {
    fprintf(stderr, "ERROR: Cannot allocate memory because len = %i in allocate_ustrmem().\n", len);
    exit(EXIT_FAILURE);
  }

  tmp = (uint8_t *) malloc (len * sizeof (uint8_t));
  if (tmp != NULL) {
    memset(tmp, 0, len * sizeof (uint8_t));
    return (tmp);
  } else {
    fprintf(stderr, "ERROR: Cannot allocate memory for array allocate_ustrmem().\n");
  }
exit (EXIT_FAILURE);

// Allocate memory for an array of ints.
int *allocate_intmem (int len)
{
    void *tmp;

    if (len <= 0) {
        fprintf (stderr, "ERROR: Cannot allocate memory because len = %i in allocate_intmem().\n", len);
        exit (EXIT_FAILURE);
    }

    tmp = (int *) malloc (len * sizeof (int));
    if (tmp != NULL) {
        memset (tmp, 0, len * sizeof (int));
        return (tmp);
    } else {
        fprintf (stderr, "ERROR: Cannot allocate memory for array allocate_intmem().\n");
        exit (EXIT_FAILURE);
    }
}

// Checksum function
uint16_t checksum (uint16_t *addr, int len)
{
    int nleft = len;
    int sum = 0;
    uint16_t *w = addr;
    uint16_t answer = 0;

    while (nleft > 1) {
        sum += *w++;
        nleft -= sizeof (uint16_t);
    }

    if (nleft == 1) {
```c
*(uint8_t *)&answer = *(uint8_t *)w;
sum += answer;
}

sum = (sum >> 16) + (sum & 0xFFFF);
sum += (sum >> 16);
answer = ~sum;
return (answer);
}

// Build IPv4 TCP pseudo-header and call checksum function.
uint16_t tcp4_checksum (struct ip iphdr, struct tcphdr tcphdr, uint8_t *payload, int payloadlen)
{
    uint16_t svalue;
    char buf[IP_MAXPACKET], cvalue;
    char *ptr;
    int i, chksumlen = 0;

    memset (buf, 0, IP_MAXPACKET);

    ptr = &buf[0]; // ptr points to beginning of buffer buf

    // Copy source IP address into buf (32 bits)
    memcpy (ptr, &iphdr.ip_src.s_addr, sizeof (iphdr.ip_src.s_addr));
    ptr += sizeof (iphdr.ip_src.s_addr);
    chksumlen += sizeof (iphdr.ip_src.s_addr);

    // Copy destination IP address into buf (32 bits)
    memcpy (ptr, &iphdr.ip_dst.s_addr, sizeof (iphdr.ip_dst.s_addr));
    ptr += sizeof (iphdr.ip_dst.s_addr);
    chksumlen += sizeof (iphdr.ip_dst.s_addr);

    // Copy zero field to buf (8 bits)
    *ptr = 0; ptr++;
    chksumlen += 1;

    // Copy transport layer protocol to buf (8 bits)
    memcpy (ptr, &iphdr.ip_p, sizeof (iphdr.ip_p));
```
ptr += sizeof (iphdr.ip_p);
chksumlen += sizeof (iphdr.ip_p);

// Copy TCP length to buf (16 bits)
svalue = htons (sizeof (tcphdr) + payloadlen);
memcpy (ptr, &svalue, sizeof (svalue));
ptr += sizeof (svalue);
chksumlen += sizeof (svalue);

// Copy TCP source port to buf (16 bits)
memcpy (ptr, &tcphdr.th_sport, sizeof (tcphdr.th_sport));
ptr += sizeof (tcphdr.th_sport);
chksumlen += sizeof (tcphdr.th_sport);

// Copy TCP destination port to buf (16 bits)
memcpy (ptr, &tcphdr.th_dport, sizeof (tcphdr.th_dport));
ptr += sizeof (tcphdr.th_dport);
chksumlen += sizeof (tcphdr.th_dport);

// Copy sequence number to buf (32 bits)
memcpy (ptr, &tcphdr.th_seq, sizeof (tcphdr.th_seq));
ptr += sizeof (tcphdr.th_seq);
chksumlen += sizeof (tcphdr.th_seq);

// Copy acknowledgement number to buf (32 bits)
memcpy (ptr, &tcphdr.th_ack, sizeof (tcphdr.th_ack));
ptr += sizeof (tcphdr.th_ack);
chksumlen += sizeof (tcphdr.th_ack);

// Copy data offset to buf (4 bits) and
// copy reserved bits to buf (4 bits)
cvalue = (tcphdr.th_off << 4) + tcphdr.th_x2;
memcpy (ptr, &cvalue, sizeof (cvalue));
ptr += sizeof (cvalue);
chksumlen += sizeof (cvalue);

// Copy TCP flags to buf (8 bits)
memcpy (ptr, &tcphdr.th_flags, sizeof (tcphdr.th_flags));
ptr += sizeof (tcphdr.th_flags);
chksumlen += sizeof(tcphdr.th_flags);

// Copy TCP window size to buf (16 bits)
memcpy (ptr, &tcphdr.th_win, sizeof(tcphdr.th_win));
ptr += sizeof(tcphdr.th_win);
chksumlen += sizeof(tcphdr.th_win);

// Copy TCP checksum to buf (16 bits)
// Zero, since we don't know it yet
*ptr = 0; ptr++;
*ptr = 0; ptr++;
chksumlen += 2;

// Copy urgent pointer to buf (16 bits)
memcpy (ptr, &tcphdr.th_urp, sizeof(tcphdr.th_urp));
ptr += sizeof(tcphdr.th_urp);
chksumlen += sizeof(tcphdr.th_urp);

// Copy payload to buf
memcpy (ptr, payload, payloadlen);
ptr += payloadlen;
chksumlen += payloadlen;

// Pad to the next 16-bit boundary
i = 0;
while (((payloadlen+i)%2) != 0) {
    i++;
    chksumlen++;
    ptr++;
}

return checksum ((uint16_t *) buf, chksumlen);
}

// Build IPv4 TCP pseudo-header and call checksum function.
uint16_t tcp2_checksum (struct ip iphdr, struct tcphdr tcphdr)
{
    uint16_t svalue;


char buf[IP_MAXPACKET], cvalue;
char *ptr;
int chksumlen = 0;

ptr = &buf[0]; // ptr points to beginning of buffer buf

// Copy source IP address into buf (32 bits)
memcpy (ptr, &iphdr.ip_src.s_addr, sizeof (iphdr.ip_src.s_addr));
ptr += sizeof (iphdr.ip_src.s_addr);
chksumlen += sizeof (iphdr.ip_src.s_addr);

// Copy destination IP address into buf (32 bits)
memcpy (ptr, &iphdr.ip_dst.s_addr, sizeof (iphdr.ip_dst.s_addr));
ptr += sizeof (iphdr.ip_dst.s_addr);
chksumlen += sizeof (iphdr.ip_dst.s_addr);

// Copy zero field to buf (8 bits)
*ptr = 0; ptr++;
chksumlen += 1;

// Copy transport layer protocol to buf (8 bits)
memcpy (ptr, &iphdr.ip_p, sizeof (iphdr.ip_p));
ptr += sizeof (iphdr.ip_p);
chksumlen += sizeof (iphdr.ip_p);

// Copy TCP length to buf (16 bits)
svalue = htons (sizeof (tcphdr));
memcpy (ptr, &svalue, sizeof (svalue));
ptr += sizeof (svalue);
chksumlen += sizeof (svalue);

// Copy TCP source port to buf (16 bits)
memcpy (ptr, &tcphdr.th_sport, sizeof (tcphdr.th_sport));
ptr += sizeof (tcphdr.th_sport);
chksumlen += sizeof (tcphdr.th_sport);

// Copy TCP destination port to buf (16 bits)
memcpy (ptr, &tcphdr.th_dport, sizeof (tcphdr.th_dport));
ptr += sizeof (tcphdr.th_dport);
chksumlen += sizeof (tcphdr.th_dport);

// Copy sequence number to buf (32 bits)
mempy (ptr, &tcphdr.th_seq, sizeof (tcphdr.th_seq));
ptr += sizeof (tcphdr.th_seq);
chksumlen += sizeof (tcphdr.th_seq);

// Copy acknowledgement number to buf (32 bits)
mempy (ptr, &tcphdr.th_ack, sizeof (tcphdr.th_ack));
ptr += sizeof (tcphdr.th_ack);
chksumlen += sizeof (tcphdr.th_ack);

// Copy data offset to buf (4 bits) and
// copy reserved bits to buf (4 bits)
cvalue = (tcphdr.th_off << 4) + tcphdr.th_x2;
mempy (ptr, &cvalue, sizeof (cvalue));
ptr += sizeof (cvalue);
chksumlen += sizeof (cvalue);

// Copy TCP flags to buf (8 bits)
mempy (ptr, &tcphdr.th_flags, sizeof (tcphdr.th_flags));
ptr += sizeof (tcphdr.th_flags);
chksumlen += sizeof (tcphdr.th_flags);

// Copy TCP window size to buf (16 bits)
mempy (ptr, &tcphdr.th_win, sizeof (tcphdr.th_win));
ptr += sizeof (tcphdr.th_win);
chksumlen += sizeof (tcphdr.th_win);

// Copy TCP checksum to buf (16 bits)
// Zero, since we don't know it yet
*ptr = 0; ptr++;
*ptr = 0; ptr++;
chksumlen += 2;

// Copy urgent pointer to buf (16 bits)
mempy (ptr, &tcphdr.th_urp, sizeof (tcphdr.th_urp));
ptr += sizeof (tcphdr.th_urp);
chksumlen += sizeof (tcphdr.th_urp);
return checksum ((uint16_t *) buf, chksumlen);

***********************ARP.c**********************************

// Send an IPv4 ARP packet via raw socket at the link layer (ethernet frame).
// Values set for ARP request.

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <netdb.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netinet/ip.h>
#include <netinet/tcp.h>
#include <arpa/inet.h>
#include <sys/ioctl.h>
#include <net/if.h>
#include <linux/if_ether.h>
#include <errno.h>

typedef struct _arp_hdr arp_hdr;

struct _arp_hdr {
    uint16_t htype;
    uint16_t ptype;
    uint8_t hlen;
    uint8_t plen;
    uint16_t opcode;
    uint8_t sender_mac[6];
    uint8_t sender_ip[4];
    uint8_t target_mac[6];
uint8_t target_ip[4];
};

#define ETH_HDRLEN 14   // Ethernet header length
#define TCP_HDRLEN 20
#define IP4_HDRLEN 20    // IPv4 header length
#define ARP_HDRLEN 28    // ARP header length
#define ARPOP_REQUEST 1  // Taken from <linux/if_arp.h>
#define ARPOP_REPLY 2

char *allocate_strmem (int);
uint8_t *allocate_ustrmem (int);
int *allocate_intmem (int);

int interface_lookup(char*, char*, struct ifreq*, uint8_t *, struct sockaddr_ll*);
in t listen_ARP(int, uint8_t *, arp_hdr *, uint8_t *);
in t fill_ARPhdr(arp_hdr *, uint8_t *);

uint16_t checksum (uint16_t *, int);
uint16_t tcp4_checksum (struct ip, struct tcphdr);

int main (int argc, char **argv)
{

printf("Starting\n");

int sd;
ar p_hdr ar phdr_out;
uint8_t *src_mac, *dst_mac, *ether_frame;
struct addrinfo hints, *res;
struct sockaddr_ll device;
struct ifreq ifr;

struct ip iphdr;
int *ip_flags;
int status;

struct tcphdr tcphdr;
int *tcp_flags;
int i;
int frame_length, bytes;

// Allocate memory for various arrays.
src_mac = allocate_ustrmem(6);
dst_mac = allocate_ustrmem(6);
ether_frame = allocate_ustrmem(IP_MAXPACKET);
interface = allocate_strmem(40);
target = allocate_strmem(40);
src_ip = allocate_strmem(INET_ADDRSTRLEN);
dst_ip = allocate_strmem(INET_ADDRSTRLEN);
ip_flags = allocate_intmem (4);
tcp_flags = allocate_intmem (8);

// Look-up interface
interface_lookup(interface, "wlan0", &ifr, src_mac, &device);

// Resolve ipv4 url if needed
config_ipv4(src_ip, "160.39.10.135", target, "www.google.com", src_mac, &hints, res, &arphdr_out, &device, dst_ip);

// Set destination MAC address: broadcast address
memset (dst_mac, 0xff, 6 * sizeof (uint8_t));

// Fill out ARP packet
fill_ARPhdr(&arphdr_out, src_mac);

sd = fill_send_ETHhdr(ether_frame, dst_mac, src_mac, &arphdr_out, &device);

listen_ARP(sd, ether_frame, &arphdr_out, dst_mac);

//IPV4 header
// IPv4 header Length (4 bits): Number of 32-bit words in header = 5
iphdr.ip_hl = IP4_HDRLEN / sizeof (uint32_t);

// Internet Protocol version (4 bits): IPv4
iphdr.ip_v = 4;
// Type of service (8 bits)
iphdr.ip_tos = 0;

// Total length of datagram (16 bits): IP header + TCP header
iphdr.ip_len = htons (IP4_HDRLEN + TCP_HDRLEN);

// ID sequence number (16 bits): unused, since single datagram
iphdr.ip_id = htons (0);

// Flags, and Fragmentation offset (3, 13 bits): 0 since single datagram

// Zero (1 bit)
ip Flags[0] = 0;

// Do not fragment flag (1 bit)
ip Flags[1] = 0;

// More fragments following flag (1 bit)
ip Flags[2] = 0;

// Fragmentation offset (13 bits)
ip Flags[3] = 0;

iphdr.ip_off = htons ((ip Flags[0] << 15)
                      + (ip Flags[1] << 14)
                      + (ip Flags[2] << 13)
                      + ip Flags[3]);

// Time-to-Live (8 bits): default to maximum value
iphdr.ip_ttl = 255;

// Transport layer protocol (8 bits): 6 for TCP
iphdr.ip_p = IPPROTO_TCP;

// Source IPv4 address (32 bits)
if ((status = inet_pton (AF_INET, src_ip, &(iphdr.ip_src))) != 1) {
    fprintf (stderr, "inet_pton() failed 1.\nError message: %s", strerror (status));
exit (EXIT_FAILURE);

// Destination IPv4 address (32 bits)
if ((status = inet_pton (AF_INET, dst_ip, &iphdr.ip_dst)) != 1) {
    fprintf (stderr, "inet_pton() failed 2.\nError message: %s", strerror (status));
    exit (EXIT_FAILURE);
}

// IPv4 header checksum (16 bits): set to 0 when calculating checksum
iphdr.ip_sum = 0;
iphdr.ip_sum = checksum ((uint16_t *)iphdr, IP_HDRLEN);

// TCP header

// Source port number (16 bits)
tcphdr.th_sport = htons (52946);

// Destination port number (16 bits)
tcphdr.th_dport = htons (80);

// Sequence number (32 bits)
tcphdr.th_seq = htonl (random());

// Acknowledgement number (32 bits): 0 in first packet of SYN/ACK process
tcphdr.th_ack = htonl (0);

// Reserved (4 bits): should be 0
tcphdr.th_x2 = 0;

// Data offset (4 bits): size of TCP header in 32-bit words
tcphdr.th_off = TCP_HDRLEN / 4;

// Flags (8 bits)

// FIN flag (1 bit)
tcp_flags[0] = 0;
// SYN flag (1 bit): set to 1
tcp_flags[1] = 1;

// RST flag (1 bit)
tcp_flags[2] = 0;

// PSH flag (1 bit)
tcp_flags[3] = 0;

// ACK flag (1 bit)
tcp_flags[4] = 0;

// URG flag (1 bit)
tcp_flags[5] = 0;

// ECE flag (1 bit)
tcp_flags[6] = 0;

// CWR flag (1 bit)
tcp_flags[7] = 0;

tcphdr.th_flags = 0;
for (i=0; i<8; i++) {
    tcphdr.th_flags += (tcp_flags[i] << i);
}

// Window size (16 bits)
tcphdr.th_win = htons (14600);

// Urgent pointer (16 bits): 0 (only valid if URG flag is set)
tcphdr.th_urp = htons (0);

// TCP checksum (16 bits)
tcphdr.th_sum = 0;
tcphdr.th_sum = tcp4_checksum (iphdr, tcphdr);

// Fill out ethernet frame header.
// Ethernet frame length = ethernet header (MAC + MAC + ethernet type) + ethernet data (IP header + TCP header)
frame_length = 6 + 6 + 2 + IP4_HDRLEN + TCP_HDRLEN;

// Destination and Source MAC addresses
memcpy (ether_frame, dst_mac, 6 * sizeof (uint8_t));
membcpy (ether_frame + 6, src_mac, 6 * sizeof (uint8_t));

// Next is ethernet type code (ETH_P_IP for IPv4).
// http://www.iana.org/assignments/ethernet-numbers
ether_frame[12] = ETH_P_IP / 256;
ether_frame[13] = ETH_P_IP % 256;

// Next is ethernet frame data (IPv4 header + TCP header).

// IPv4 header
memcpy (ether_frame + ETH_HDRLEN, &iphdr, IP4_HDRLEN * sizeof (uint8_t));

// TCP header
memcpy (ether_frame + ETH_HDRLEN + IP4_HDRLEN, &tcphdr, TCP_HDRLEN * sizeof (uint8_t));

// Send ethernet frame to socket.
if ((bytes = sendto (sd, ether_frame, frame_length, 0, (struct sockaddr *) &device, sizeof (device))) <= 0) {
    perror ("sendto() failed");
    exit (EXIT_FAILURE);
}

// Put more checks to verify that the packet received is an ACK
printf("Receiving TCP...

struct tcphdr *tcp_in;
tcp_in = (struct tcphdr *) (ether_frame + 6 + 6 + 2 + IP4_HDRLEN);
struct ip *ip_in;
ip_in = (struct ip *) (ether_frame + 6 + 6 + 2);
while (((ether_frame[12] << 8) + ether_frame[13]) != ETH_P_IP) || (*inet_ntoa(ip_in->ip_src)) != *dst_ip) {
    if ((status = recv(sd, ether_frame, IP_MAXPACKET, 0)) < 0) {
        if (errno == EINTR) {
            memset(ether_frame, 0, IP_MAXPACKET * sizeof(uint8_t));
            continue; // Something weird happened, but let's try again.
        } else {
            perror("recv() failed: ");
            exit(EXIT_FAILURE);
        }
    }
}

tcphdr.th_seq = htonl(1 + htonl(tcphdr.th_seq));
tcphdr.th_ack = htonl(1 + ntohl(tcp_in->th_seq));

// Flags (8 bits)

// FIN flag (1 bit)
tcp_flags[0] = 0;

// SYN flag (1 bit): set to 1
tcp_flags[1] = 0;

// RST flag (1 bit)
tcp_flags[2] = 0;

// PSH flag (1 bit)
tcp_flags[3] = 0;

// ACK flag (1 bit)
tcp_flags[4] = 1;

// URG flag (1 bit)
tcp_flags[5] = 0;

// ECE flag (1 bit)
tcp_flags[6] = 0;
// CWR flag (1 bit)
tcp_flags[7] = 0;

tcphdr.th_flags = 0;
for (i = 0; i < 8; i++) {
    tcphdr.th_flags += (tcp_flags[i] << i);
}

// TCP checksum (16 bits)
tcphdr.th_sum = 0;
tcphdr.th_sum = tcp4_checksum (iphdr, tcphdr);

// Destination and Source MAC addresses
memcpy (ether_frame, dst_mac, 6 * sizeof (uint8_t));
memcpy (ether_frame + 6, src_mac, 6 * sizeof (uint8_t));

// Next is ethernet type code (ETH_P_IP for IPv4).
// http://www.iana.org/assignments/ethernet-numbers
ether_frame[12] = ETH_P_IP / 256;
ether_frame[13] = ETH_P_IP % 256;

// Next is ethernet frame data (IPv4 header + TCP header).

// IPv4 header
memcpy (ether_frame + ETH_HDRLEN, &iphdr, IP4_HDRLEN * sizeof (uint8_t));

// TCP header
memcpy (ether_frame + ETH_HDRLEN + IP4_HDRLEN, &tcphdr, TCP_HDRLEN * sizeof (uint8_t));

// Send ethernet frame to socket.
if ((bytes = sendto (sd, ether_frame, frame_length, 0, (struct sockaddr *)&device, sizeof (device))) <= 0) {
    perror ("sendto() failed");
    exit (EXIT_FAILURE);
}
close (sd);

// Free allocated memory.
free (src_mac);
free (dst_mac);
free (ether_frame);
free (interface);
free (target);
free (src_ip);
free (dst_ip);
free (ip_flags);

return (EXIT_SUCCESS);
}

int fill_ARPhdr(arp_hdr *arphdr_out, uint8_t *src_mac)
{
    // Hardware type (16 bits): 1 for ethernet
    arphdr_out->htype = htons (1);

    // Protocol type (16 bits): 2048 for IP
    arphdr_out->ptype = htons (ETH_P_IP);

    // Hardware address length (8 bits): 6 bytes for MAC address
    arphdr_out->hlen = 6;

    // Protocol address length (8 bits): 4 bytes for IPv4 address
    arphdr_out->plen = 4;

    // OpCode: 1 for ARP request
    arphdr_out->opcode = htons (ARPOP_REQUEST);

    // Sender hardware address (48 bits): MAC address
    memcpy (&arphdr_out->sender_mac, src_mac, 6 * sizeof (uint8_t));

    // Sender protocol address (32 bits)
    // See getaddrinfo() resolution of src_ip.
// Target hardware address (48 bits): zero, since we don't know it yet.
memset (&arphdr_out->target_mac, 0, 6 * sizeof (uint8_t));

// Target protocol address (32 bits)
// See getaddrinfo() resolution of target.
return 0;
}

int fill_send_ETHhdr(uint8_t *ether_frame, uint8_t *dst_mac, uint8_t *src_mac, arp_hdr *arphdr_out, struct sockaddr_ll *device)
{
    int sd, frame_length, bytes;
    // Fill out ethernet frame header.
    // Ethernet frame Length = ethernet header (MAC + MAC + ethernet type) + ethernet data (ARP header)
    frame_length = 6 + 6 + 2 + ARP_HDRLEN;

    // Destination and Source MAC addresses
    memcpy (ether_frame, dst_mac, 6 * sizeof (uint8_t));
    memcpy (ether_frame + 6, src_mac, 6 * sizeof (uint8_t));

    // Next is ethernet type code (ETH_P_ARP for ARP).
    // http://www.iana.org/assignments/ethernet-numbers
    ether_frame[12] = ETH_P_ARP / 256;
    ether_frame[13] = ETH_P_ARP % 256;

    // Next is ethernet frame data (ARP header).
    // ARP header
    memcpy (ether_frame + ETH_HDRLEN, arphdr_out, ARP_HDRLEN * sizeof (uint8_t));

    // Submit request for a raw socket descriptor.
    if ((sd = socket (PF_PACKET, SOCK_RAW, htons (ETH_P_ALL))) < 0) {
        perror ("socket() failed ");
        exit (EXIT_FAILURE);
    }
}
// Send ethernet frame to socket.
if ((bytes = sendto (sd, ether_frame, frame_length, 0, (struct sockaddr *) device, sizeof (struct sockaddr_ll))) <= 0) {
    perror ("sendto() failed");
    exit (EXIT_FAILURE);
}

return sd;
}

int config_ipv4(char* src_ip, char* src_ip_addr, char* target, char* trg_ip_addr, uint8_t *src_mac, struct addrinfo *hints, struct addrinfo *res, arp_hdr *arphdr_out, struct sockaddr_ll *device, char* dst_ip)
{
    int status;
    struct sockaddr_in *ipv4;
    void *tmp;

    // Source IPv4 address: you need to fill this out
    strcpy (src_ip, src_ip_addr);

    // Destination URL or IPv4 address (must be a link-local node): you need to fill this out
    strcpy (target, trg_ip_addr);

    // Fill out hints for getaddrinfo().
    memset (hints, 0, sizeof (struct addrinfo));
    hints->ai_family = AF_INET;
    hints->ai_socktype = SOCK_STREAM;
    hints->ai_flags = hints->ai_flags | AI_CANONNAME;

    // Source IP address
    if ((status = inet_pton (AF_INET, src_ip, arphdr_out->sender_ip)) != 1) {
        fprintf (stderr, "inet_pton() source IP address.\nError message: %s", strerror (status));
        exit (EXIT_FAILURE);
    }

    // Resolve target using getaddrinfo().
if ((status = getaddrinfo (target, NULL, hints, &res)) != 0) {
    fprintf (stderr, "getaddrinfo() failed: %s\n", gai_strerror (status));
    exit (EXIT_FAILURE);
}
ipv4 = (struct sockaddr_in *) res->ai_addr;
tmp = &(ipv4->sin_addr);
memcpy (arphdr_out->target_ip, tmp, 4 * sizeof (uint8_t));
if (inet_ntop (AF_INET, tmp, dst_ip, INET_ADDRSTRLEN) == NULL) {
    status = errno;
    fprintf (stderr, "inet_ntop() failed.\n Error message: %s", strerror(status));
    exit(EXIT_FAILURE);
}
freeaddrinfo (res);

// Fill out sockaddr_ll.
device->sll_family = AF_PACKET;
memcpy (device->sll_addr, src_mac, 6 * sizeof (uint8_t));
device->sll_halen = htons (6);

return 0;
}

int listen_ARP(int sd, uint8_t *ether_frame, arp_hdr *arphrd_out, uint8_t *dst_mac)
{
    // Listen for incoming ethernet frame from socket sd.
    // We expect an ARP ethernet frame of the form:
    //   MAC (6 bytes) + MAC (6 bytes) + ethernet type (2 bytes)
    //   + ethernet data (ARP header) (28 bytes)
    // Keep at it until we get an ARP reply.
    printf("Receiving ... \n");

    int status, i;
    arp_hdr *arp_pt_in;
    arp_pt_in = (arp_hdr *) (ether_frame + 6 + 6 + 2);
    while (((((((ether_frame[12]) << 8) + ether_frame[13]) != ETH_P_ARP) || (ntohs (arp_pt_in->opcode) != ARP0P_REPLY)) {
        if ((status = recv (sd, ether_frame, IP_MAXPACKET, 0)) < 0) {

}
if (errno == EINTR) {
    memset (ether_frame, 0, IP_MAXPACKET * sizeof (uint8_t));
    continue;  // Something weird happened, but let's try again.
} else {
    perror ("recv() failed: ");
    exit (EXIT_FAILURE);
}

// DEBBUG - TO BE COMMENTED
// Print out contents of received ethernet frame.
printf ("nEthernet frame header:n");
printf ("Destination MAC (this node): ");
for (i=0; i<5; i++) {
    printf ("%02x:", ether_frame[i]);
}
printf ("%02x\n", ether_frame[5]);
printf ("Source MAC: ");
for (i=0; i<5; i++) {
    printf ("%02x:", ether_frame[i+6]);
}
printf ("%02x\n", ether_frame[11]);
// Next is ethernet type code (ETH_P_ARP for ARP).
// http://www.iana.org/assignments/ethernet-numbers
printf ("Ethernet type code (2054 = ARP): \n", ((ether_frame[12]) << 8) + ether_frame[13]);
printf ("nEthernet data (ARP header):\n");
printf ("Hardware type (1 = ethernet (10 Mb)): %u\n", ntohs (arp_pt_in->htype));
printf ("Protocol type (2048 for IPv4 addresses): %u\n", ntohs (arp_pt_in->ptype));
printf ("Hardware (MAC) address length (bytes): %u\n", arp_pt_in->hlen);
printf ("Protocol (IPv4) address length (bytes): %u\n", arp_pt_in->plen);
printf ("Opcode (2 = ARP reply): %u\n", ntohs (arp_pt_in->opcode));
printf ("Sender hardware (MAC) address: ");
for (i=0; i<5; i++) {
    printf ("%02x:", arp_pt_in->sender_mac[i]);
}
printf ("%02x\n", arp_pt_in->sender_mac[5]);
printf ("Sender protocol (IPv4) address: %u.%u.%u.%u\n",

arp_pt_in->sender_ip[0], arp_pt_in->sender_ip[1], arp_pt_in->sender_ip[2], arp_pt_in->sender_ip[3]);

printf("Target (this node) hardware (MAC) address: ");
for (i=0; i<5; i++) {
    printf("%02x:", arp_pt_in->target_mac[i]);
}
printf("%02x\n", arp_pt_in->target_mac[5]);
printf("Target (this node) protocol (IPv4) address: %u.%u.%u.%u\n",
    arp_pt_in->target_ip[0], arp_pt_in->target_ip[1], arp_pt_in->target_ip[2], arp_pt_in->target_ip[3]);

for (i = 0; i < 6; i++) dst_mac[i] = arp_pt_in->sender_mac[i];
printf("dst_mac :");
for (i = 0; i < 6; i++) printf("%02x:", dst_mac[i]);
printf("\n");
return 0;
}

// Allocate memory for an array of chars.
char *allocate_strmem (int len)
{
    void *tmp;

    if (len <= 0) {
        fprintf(stderr, "ERROR: Cannot allocate memory because len = %i in allocate_strmem().\n", len);
        exit(EXIT_FAILURE);
    }

tmp = (char *) malloc(len * sizeof(char));
if (tmp != NULL) {
    memset(tmp, \0, len * sizeof(char));
    return (tmp);
} else {
    fprintf(stderr, "ERROR: Cannot allocate memory for array allocate_strmem().\n");
    exit(EXIT_FAILURE);
}
}
// Allocate memory for an array of unsigned chars.

uint8_t *allocate_ustrmem (int len)
{
    void *tmp;

    if (len <= 0) {
        fprintf (stderr, "ERROR: Cannot allocate memory because len = %i in allocate_ustrmem().\n", len);
        exit (EXIT_FAILURE);
    }

    tmp = (uint8_t *) malloc (len * sizeof (uint8_t));
    if (tmp != NULL) {
        memset (tmp, 0, len * sizeof (uint8_t));
        return (tmp);
    } else {
        fprintf (stderr, "ERROR: Cannot allocate memory for array allocate_ustrmem().\n");
        exit (EXIT_FAILURE);
    }
}

int interface_lookup(char *interface, char *name, struct ifreq *ifr, uint8_t *src_mac, struct sockaddr_ll *device)
{
    int sd;
    printf("Looking up interface\n");
    strcpy(interface, name);

    // Submit request for a socket descriptor to look up interface.
    if ((sd = socket (AF_INET, SOCK_RAW, IPPROTO_RAW)) < 0) {
        perror ("socket() failed to get socket descriptor for using ioctl() ");
        exit (EXIT_FAILURE);
    }

    // Use ioctl() to look up interface name and get its MAC address.
    memset (ifr, 0, sizeof (*ifr));
    snprintf (ifr->ifr_name, sizeof (ifr->ifr_name), "%s", interface);
    if (ioctl (sd, SIOCGIFHWADDR, ifr) < 0) {
        perror ("ioctl() failed to get MAC address for interface ");
        exit (EXIT_FAILURE);
    }

    // Set the source MAC address for the network interface.
    memcpy (ifr->ifr_hwaddr, src_mac, sizeof (ifr->ifr_hwaddr));

    return sd;
}
perror ("ioctl() failed to get source MAC address ");
return (EXIT_FAILURE);
}
close (sd);

// Copy source MAC address.
memcpy (src_mac, ifr->ifr_hwaddr.sa_data, 6 * sizeof (uint8_t));

// Report source MAC address to stdout.
int i;
printf ("MAC address for interface %s is ", interface);
for (i=0; i<5; i++) {
    printf ("%02x:", src_mac[i]);
}
printf ("%02x
", src_mac[5]);

// Find interface index from interface name and store index in
// struct sockaddr_ll device, which will be used as an argument of sendto().
memset (device, 0, sizeof (device));
if ((device->sll_ifindex = if_nametoindex (interface)) == 0) {
    perror ("if_nametoindex() failed to obtain interface index ");
    exit (EXIT_FAILURE);
}
printf ("Index for interface %s is %i
", interface, device->sll_ifindex);

return 0;
}

// Allocate memory for an array of ints.
int *allocate_intmem (int len)
{
    void *tmp;

    if (len <= 0) {
        fprintf (stderr, "ERROR: Cannot allocate memory because len = %i in allocate_intmem().\n", len);
        exit (EXIT_FAILURE);
    }
}
tmp = (int *) malloc (len * sizeof (int));
if (tmp != NULL) {
    memset (tmp, 0, len * sizeof (int));
    return (tmp);
} else {
    fprintf (stderr, "ERROR: Cannot allocate memory for array allocate_intmem().\n");
    exit (EXIT_FAILURE);
}

// Checksum function
uint16_t checksum (uint16_t *addr, int len)
{
    int nleft = len;
    int sum = 0;
    uint16_t *w = addr;
    uint16_t answer = 0;

    while (nleft > 1) {
        sum += *w++;
        nleft -= sizeof (uint16_t);
    }

    if (nleft == 1) {
        *(uint8_t *) &(answer) = *(uint8_t *) w;
        sum += answer;
    }

    sum = (sum >> 16) + (sum & 0xFFFF);
    sum += (sum >> 16);
    answer = ~sum;
    return (answer);
}

// Build IPv4 TCP pseudo-header and call checksum function.
uint16_t tcp4_checksum (struct ip iphdr, struct tcphdr tcphdr)
{
    uint16_t svalue;
char buf[IP_MAXPACKET], cvalue;
char *ptr;
int chksumlen = 0;

ptr = &buf[0]; // ptr points to beginning of buffer buf

// Copy source IP address into buf (32 bits)
memcpy(ptr, &iphdr.ip_src.s_addr, sizeof (iphdr.ip_src.s_addr));
ptr += sizeof (iphdr.ip_src.s_addr);
chksumlen += sizeof (iphdr.ip_src.s_addr);

// Copy destination IP address into buf (32 bits)
memcpy(ptr, &iphdr.ip_dst.s_addr, sizeof (iphdr.ip_dst.s_addr));
ptr += sizeof (iphdr.ip_dst.s_addr);
chksumlen += sizeof (iphdr.ip_dst.s_addr);

// Copy zero field to buf (8 bits)
*ptr = 0; ptr++;
chksumlen += 1;

// Copy transport layer protocol to buf (8 bits)
memcpy(ptr, &iphdr.ip_p, sizeof (iphdr.ip_p));
ptr += sizeof (iphdr.ip_p);
chksumlen += sizeof (iphdr.ip_p);

// Copy TCP length to buf (16 bits)
svalue = htons (sizeof (tcphdr));
memcpy(ptr, &svalue, sizeof (svalue));
ptr += sizeof (svalue);
chksumlen += sizeof (svalue);

// Copy TCP source port to buf (16 bits)
memcpy(ptr, &tcphdr.th_sport, sizeof (tcphdr.th_sport));
ptr += sizeof (tcphdr.th_sport);
chksumlen += sizeof (tcphdr.th_sport);

// Copy TCP destination port to buf (16 bits)
memcpy(ptr, &tcphdr.th_dport, sizeof (tcphdr.th_dport));
ptr += sizeof (tcphdr.th_dport);
chksumlen += sizeof(tcphdr.th_dport);

// Copy sequence number to buf (32 bits)
memcpy(ptr, &tcphdr.th_seq, sizeof(tcphdr.th_seq));
ptr += sizeof(tcphdr.th_seq);
chksumlen += sizeof(tcphdr.th_seq);

// Copy acknowledgement number to buf (32 bits)
memcpy(ptr, &tcphdr.th_ack, sizeof(tcphdr.th_ack));
ptr += sizeof(tcphdr.th_ack);
chksumlen += sizeof(tcphdr.th_ack);

// Copy data offset to buf (4 bits) and
// copy reserved bits to buf (4 bits)
cvalue = (tcphdr.th_off << 4) + tcphdr.th_x2;
memcpy(ptr, &cvalue, sizeof(cvalue));
ptr += sizeof(cvalue);
chksumlen += sizeof(cvalue);

// Copy TCP flags to buf (8 bits)
memcpy(ptr, &tcphdr.th_flags, sizeof(tcphdr.th_flags));
ptr += sizeof(tcphdr.th_flags);
chksumlen += sizeof(tcphdr.th_flags);

// Copy TCP window size to buf (16 bits)
memcpy(ptr, &tcphdr.th_win, sizeof(tcphdr.th_win));
ptr += sizeof(tcphdr.th_win);
chksumlen += sizeof(tcphdr.th_win);

// Copy TCP checksum to buf (16 bits)
// Zero, since we don't know it yet
*ptr = 0; ptr++;
*ptr = 0; ptr++;
chksumlen += 2;

// Copy urgent pointer to buf (16 bits)
memcpy(ptr, &tcphdr.th_urp, sizeof(tcphdr.th_urp));
ptr += sizeof(tcphdr.th_urp);
chksumlen += sizeof(tcphdr.th_urp);
return checksum ((uint16_t *) buf, chksumlen);
}

/**************************** MAIN.c*****************************/

int main(int argc, char **argv) {
    int status;
    tcp_set_rawsck();

    uint8_t *rcv_data = (uint8_t *) malloc(RCP_BUFFER*sizeof(uint8_t));
    if (rcv_data == NULL) {
        perror("Error allocating reception buffer\n");
        exit(EXIT_FAILURE);
    }

    struct tcp_ctrl *tcp_ctrl = tcp_new();
    if ((status = tcp_bind(tcp_ctrl, "209.2.233.44", 52000, "wlan0")) < 0) {
        perror("Couldn't bind socket to port\n");
        exit(EXIT_FAILURE);
    }
    if ((status = tcp_connect(tcp_ctrl, "www.google.com")) < 0) {
        perror("Couldn't connect to server\n");
        exit(EXIT_FAILURE);
    }
    printf("*************** HANDSHAKE COMPLETED ***************\n");

    char *sd_data1 = "GET / HTTP/1.1\r\n\r"
    tcp_write(tcp_ctrl, sd_data1, strlen(sd_data1));
    printf("*************** FIRST REQUEST COMPLETED ***************\n");

    int len = tcp_rcv(tcp_ctrl, rcv_data, RCP_BUFFER);
    printf("*************** FIRST TRANSMISSION COMPLETED ***************\n");

    FILE *f1 = fopen("result1", "ab");
    fwrite(rcv_data, 1, len, f1);
    printf("*************** RECORDING FIRST PHASE RESULTS COMPLETED ***************\n");

    if (len >= 0) {

71

```c
    tcp_write(tcp_ctrl, sd_data1, strlen(sd_data1));
    printf("*************** SECOND REQUEST COMPLETED ***************\n");

    len = tcp_rcv(tcp_ctrl, rcv_data, RCP_BUFFER);
    printf("*************** SECOND TRANSMISSION COMPLETED ***************\n");

    FILE *f2 = fopen("result2", "ab+");
fwrite(rcv_data, 1, len, f2);
    printf("*************** RECORDING SECOND PHASE RESULTS COMPLETED
");
}
```

```verilog
module TOE (input logic clk,
           input logic rst,

             // Avalon MM slave interface
           input logic [31:0] writedata,
           input logic write,
           output logic [31:0] readdata,
           input logic read,
           input logic chipselect,
           input logic [3:0] address,

             // Avalon ST interface (source) for output ethernet frame
           output logic [31:0] eth_out_data,
           output logic eth_out_startofpacket,
           output logic eth_out_endofpacket,
           output logic [1:0] eth_out_empty,
           input logic eth_out_ready,

           ...) {...
```

71
output logic    eth_out_valid
);
wire [7:0]    addr_a;
wire [31:0]   q_a;
wire [31:0]   data_a;
wire         wren_a;
wire [7:0]    addr_b;
wire [31:0]   q_b;
wire [31:0]   data_b;
wire         wren_b;
wire         busy;
wire         ready;
wire [431:0]  header;
logic [1:0]   req_code;
logic [7:0]   id_in;
logic [7:0]   reply;
logic [31:0]  ip_src;
logic [31:0]  ip_dst;
logic [47:0]  mac_src;
logic [47:0]  mac_dst;
logic [15:0]  port_src;
logic [15:0]  port_dst;
always_ff @(posedge clk)
    begin
        if (rst)
            begin
                req_code <= 2'd0;
                id_in    <= 8'd0;
                ip_src   <= 32'd0;
                ip_dst   <= 32'd0;
                mac_src  <= 47'd0;
                mac_dst  <= 47'd0;
                port_src <= 16'd0;
                port_dst <= 16'd0;
                readdata <= 32'd0;
            end
else if (write && chipselect)
    case (address)
    4'h0 : req_code <= writedata[31:30];
    4'h1 : id_in <= writedata[31:24];
    // Address 2 is for reply and shouldn't be used
    4'h3 : ip_src <= writedata;
    4'h4 : ip_dst <= writedata;
    4'h5 : mac_src[47:16] <= writedata;
    4'h6 : mac_src[15:0] <= writedata[31:16];
    4'h7 : mac_dst[47:16] <= writedata;
    4'h8 : mac_dst[15:0] <= writedata[31:16];
    4'h9 : port_src <= writedata[31:16];
    4'ha : port_dst <= writedata[31:16];
    endcase
else if(read && chipselect)
    begin
    case (address)
    4'h0 : readdata <= {req_code, 30'd0};
    4'h1 : readdata <= {id_in, 24'd0};
    4'h2 : readdata <= {reply, 24'd0};
    4'h3 : readdata <= ip_src;
    4'h4 : readdata <= ip_dst;
    4'h5 : readdata <= mac_src[47:16];
    4'h6 : readdata <= {mac_src[15:0], 16'd0};
    4'h7 : readdata <= mac_dst[47:16];
    4'h8 : readdata <= {mac_dst[15:0], 16'd0};
    4'h9 : readdata <= {port_src, 16'd0};
    4'ha : readdata <= {port_dst, 16'd0};
    default : readdata <= 32'd0;
    endcase
    end
end

Packetizer p0 (.clk(clk),
      .rst(rst),
      .done(busy),
      .ready(ready),
      .header(header),
      .data(eth_out_data),
.p_start(eth_out_startofpacket),
.p_end(eth_out_endofpacket),
.empty(eth_out_empty),
.eth_ready(eth_out_ready),
.eth_valid(eth_out_valid));

TOE_init t0 (.data(data_a),
  .wren(wren_a),
  .addr(addr_a),
  .q(q_a),
  .*);

Packet_builder pb0 (.rst(rst),
  .clk(clk),
  .addr(addr_b),
  .ram_in(data_b),
  .ram_out(q_b),
  .wren(wren_b),
  .busy(busy),
  .ready(ready),
  .header(header));

RAM2 Connections_RAM (.clock(clk),
  .address_a(addr_a),
  .address_b(addr_b),
  .q_a(q_a),
  .q_b(q_b),
  .data_a(data_a),
  .data_b(data_b),
  .wren_a(wren_a),
  .wren_b(wren_b));

endmodule

/**********************************TOE_init.sv*********************************************/

module TOE_init( input logic clk,
  input logic rst,
input logic [1:0]  req_code,
input logic [7:0]  id_in,
output logic [7:0]  reply,
input logic [31:0]  ip_src,
input logic [31:0]  ip_dst,
input logic [47:0]  mac_src,
input logic [47:0]  mac_dst,
input logic [15:0]  port_src,
input logic [15:0]  port_dst,
output logic [31:0]  data,
output logic     wren,
output logic [7:0]  addr,
input logic [31:0]  q
);

enum logic [1:0] {WAIT_RQ, PROCESSING, RETURN} state;

logic [1:0] latch_rq;

RAM_searcher rs0 (.clk(clk),
    .rst(rst),
    .req(req_code),
    .reply(reply),
    .id_in(id_in),
    .ip_src(ip_src),
    .ip_dst(ip_dst),
    .mac_src(mac_src),
    .mac_dst(mac_dst),
    .port_src(port_src),
    .port_dst(port_dst),
    .addr(addr),
    .data(data),
    .q(q),
    .wren(wren)
);

always_ff @ (posedge clk)
begin
if (rst)
  begin
    state <= WAIT_RQ;
  end
else if ((state == WAIT_RQ) && (req_code != 2'b00)) state <= PROCESSING;
else if ((state == PROCESSING) && (reply != 8'd0)) state <= RETURN;
else if ((state == RETURN) && (req_code == 2'b00)) state <= WAIT_RQ;
end

endmodule

/**********************************RAM2.v**********************************/

// megafunction wizard: %RAM: 2-PORT%
// GENERATION: STANDARD
// VERSION: WM1.0
// MODULE: altsyncram

// synopsys translate_off
`timescale 1 ps / 1 ps
// synopsys translate_on
module RAM2 (  
    address_a,  
    address_b,  
    clock,  
    data_a,  
    data_b,  
    wren_a,  
    wren_b,  
    q_a,  
    q_b);

    input [7:0] address_a;
    input [7:0] address_b;
    input clock;
    input [31:0] data_a;
    input [31:0] data_b;
    input wren_a;
    input wren_b;
```verilog
output [31:0] q_a;
output [31:0] q_b;
`ifndef ALTERA_RESERVED_QIS
// synopsys translate_off
`endif
tri1 clock;
tri0 wren_a;
tri0 wren_b;
`ifndef ALTERA_RESERVED_QIS
// synopsys translate_on
`endif
wire [31:0] sub_wire0;
wire [31:0] sub_wire1;
wire [31:0] q_a = sub_wire0[31:0];
wire [31:0] q_b = sub_wire1[31:0];

altsyncram altsyncram_component (  
  .clock0 (clock),
  .wren_a (wren_a),
  .address_b (address_b),
  .data_b (data_b),
  .wren_b (wren_b),
  .address_a (address_a),
  .data_a (data_a),
  .q_a (sub_wire0),
  .q_b (sub_wire1),
  .aclr0 (1'b0),
  .aclr1 (1'b0),
  .addressstall_a (1'b0),
  .addressstall_b (1'b0),
  .byteena_a (1'b1),
  .byteena_b (1'b1),
  .clock1 (1'b1),
  .clocken0 (1'b1),
  .clocken1 (1'b1),
  .clocken2 (1'b1),
  .clocken3 (1'b1),
  .eccstatus (),
  .rden_a (1'b1),
```
.rden_b (1'b1));

defparam
  altsyncram_component.address_reg_b = "CLOCK0",
  altsyncram_component.clock_enable_input_a = "BYPASS",
  altsyncram_component.clock_enable_input_b = "BYPASS",
  altsyncram_component.clock_enable_output_a = "BYPASS",
  altsyncram_component.clock_enable_output_b = "BYPASS",
  altsyncram_component.indata_reg_b = "CLOCK0",
  altsyncram_component.intended_device_family = "Cyclone IV GX",
  altsyncram_component.lpm_type = "altsyncram",
  altsyncram_component.numwords_a = 256,
  altsyncram_component.numwords_b = 256,
  altsyncram_component.operation_mode = "BIDIR_DUAL_PORT",
  altsyncram_component.outdata_aclr_a = "NONE",
  altsyncram_component.outdata_aclr_b = "NONE",
  altsyncram_component.outdata_reg_a = "CLOCK0",
  altsyncram_component.outdata_reg_b = "CLOCK0",
  altsyncram_component.power_up_uninitialized = "FALSE",
  altsyncram_component.read_during_write_mode_mixed_ports = "OLD_DATA",
  altsyncram_component.read_during_write_mode_port_a = "NEW_DATA_NO_NBE_READ",
  altsyncram_component.read_during_write_mode_port_b = "NEW_DATA_NO_NBE_READ",
  altsyncram_component.widthad_a = 8,
  altsyncram_component.widthad_b = 8,
  altsyncram_component.width_a = 32,
  altsyncram_component.width_b = 32,
  altsyncram_component.width_byteena_a = 1,
  altsyncram_component.width_byteena_b = 1,
  altsyncram_component.wrcontrol_wraddress_reg_b = "CLOCK0";

endmodule

/*******************************/

module RAM_searcher( input logic clk, input logic rst, input logic [1:0] req,
output logic [7:0] reply,
input logic [7:0] id_in,
input logic [31:0] ip_src,
input logic [31:0] ip_dst,
input logic [47:0] mac_src,
input logic [47:0] mac_dst,
input logic [15:0] port_src,
input logic [15:0] port_dst,
output logic [7:0] addr,
output logic [31:0] data,
input logic [31:0] q,
output logic wren
);

parameter LAST_RECORD = 8'd240;
parameter TCP_CLOSED = 31'd1;
parameter REQ_CONN = 2'b01;
parameter REQ_DEL = 2'b00;
parameter ERR_FOUND = 8'b1000_0000;
parameter ERR_FULL = 8'b1100_0000;

logic [8:0] empty;
logic [7:0] base_addr;
logic s_done;
logic ins_done;
logic found;
logic last_record;

/* States of the high-level automata */
enum logic [2:0] {hl_IDLE, hl_SEARCHING, hl_DELETING, hl_INSERTING, hl_RETURN_SUCCESS, hl_ERROR_FOUND, hl_ERROR_FULL} hl_state;

/* States of the second level automata for searching a matching connection in the RAM */
enum logic [1:0] {s_IDLE, s_FETCH_VALID, s_CHECK, s_FOUND} s_state;

/* States of the second level automata for searching a matching connection in the RAM */
enum logic [3:0] {chk_IDLE, chk_DONE, chk_VALID, chk_IP_SRC, chk_IP_DST, chk_MAC_SRC1, chk_MAC_SRC2, chk_MAC_DST1, chk_MAC_DST2, chk_PORTS, chk_EQUAL, chk_WAIT} chk_state;
/* States for INSERTION of new record */
enum logic [3:0] {ins_IDLE, ins_VALID_STATE, ins_SEQ, ins_ACK, ins_IP_SRC, ins_IP_DST, 
  ins_MAC_SRC1, ins_MAC_SRC2, ins_MAC_DST1, ins_MAC_DST2, ins_PORTS} ins_state;

assign last_record = (base_addr == LAST_RECORD);
assign found = (chk_state == chk_EQUAL);

/* High Level State Machine */
always_ff @(posedge clk)
  begin
    if(rst)
      begin
        hl_state <= hl_IDLE;
        reply <= 8'b0;
      end
    else
      case (hl_state)
        hl_IDLE :
          begin
            case (req)
              REQ_CONN : hl_state <= hl_SEARCHING;
              REQ_DEL  : hl_state <= hl_DELETING;
              default  : hl_state <= hl_IDLE;
            endcase
            reply <= 8'b0;
          end
        hl_DELETING :
          begin
            hl_state <= hl_RETURN_SUCCESS;
          end
        hl_SEARCHING :
          begin
            if (s_done)  hl_state <= hl_INSERTING;
            else if (found)
              begin
                hl_state <= hl_ERROR_FOUND;
                reply <= ERR_FOUND;
              end
            else
              hl_state <= hl_SEARCHING;
          end
  end
hl_INSERTING:

begin

if (ins_done)
begin
hl_state <= hl_RETURN_SUCCESS;
reply <= {1'b0, empty[7:1]};  // Return the id of the connection1
end
else if (empty[8] == 1)
begin
hl_state <= hl_ERROR_FULL;
reply <= ERR_FULL;
end
else hl_state <= hl_INSERTING;
end

hl_RETURN_SUCCESS:

begin

if (req == 2'b00) hl_state <= hl_IDLE;
else    hl_state <= hl_RETURN_SUCCESS;
end

hl_ERROR_FOUND:

begin

if (req == 2'b00) hl_state <= hl_IDLE;
else    hl_state <= hl_ERROR_FOUND;
end

hl_ERROR_FULL:

begin

if (req == 2'b00) hl_state <= hl_IDLE;
else    hl_state <= hl_ERROR_FULL;
end

default: hl_state <= hl_state;
endcase

end

logic chk_done;

/* Search automata */
always_ff @ (posedge clk)
begin
if (rst)
    begin
        s_state <= s_IDLE;
        base_addr <= 8’d0;
    end
else if (hl_state == hl_SEARCHING)
    case (s_state)
        s_IDLE : begin
            if (s_done)
                begin
                    s_state <= s_IDLE; // No overlay of fetching and checking
during first
                    base_addr <= 8’d0;
                    s_done <= 1’b0;
                end
            end
        else s_state <= s_CHECK;
    end
s_CHECK :
    begin
        if(found)
            begin
                s_state <= s_IDLE;
                base_addr <= 8’d0;
            end
        else if (chk_done && last_record)
            begin
                s_done <= 1;
                s_state <= s_IDLE;
            end
        else
            begin
                s_state <= s_CHECK;
                if (chk_done) base_addr <= base_addr + 8’d10;
            end
        endcase
    end
// There could be some endianess issue when placing data in the RAM

always_comb
begin
  if (hl_state == hl_SEARCHING)
  begin
    case (chk_state)
      chk_IDLE : addr = base_addr;
      chk_WAIT : addr = base_addr + 8'd3;
      chk_VALID : addr = base_addr + 8'd4;
      chk_IP_SRC : addr = base_addr + 8'd5;
      chk_IP_DST : addr = base_addr + 8'd6;
      chk_MAC_SRC1 : addr = base_addr + 8'd7;
      chk_MAC_SRC2 : addr = base_addr + 8'd8;
      chk_MAC_DST1 : addr = base_addr + 8'd9;
      chk_MAC_DST2 : addr = base_addr;
      default : addr = base_addr;
    endcase
  end
  else if (hl_state == hl_INSERTING)
  begin
    case (ins_state)
      ins_IDLE : addr = empty[7:0];
      ins_VALID_STATE : addr = empty[7:0];
      ins_SEQ : addr = empty[7:0] + 8'd1;
      ins_ACK : addr = empty[7:0] + 8'd2;
      ins_IP_SRC : addr = empty[7:0] + 8'd3;
      ins_IP_DST : addr = empty[7:0] + 8'd4;
      ins_MAC_SRC1 : addr = empty[7:0] + 8'd5;
      ins_MAC_SRC2 : addr = empty[7:0] + 8'd6;
      ins_MAC_DST1 : addr = empty[7:0] + 8'd7;
      ins_MAC_DST2 : addr = empty[7:0] + 8'd8;
      ins_PORTS : addr = empty[7:0] + 8'd9;
      default : addr = empty[7:0];
    endcase
  end
  else if (s_state == hl_DELETING) addr = {id_in[6:0], 1'b0};
  else addr = empty[7:0];
  end

always_comb
begin
  if (hl_state == hl_INSERTING)
case(ins_state)
  ins_IDLE : data = 32'd0;
  ins_VALID_STATE : data = {1'b1, TCP_CLOSED};
  ins_SEQ : data = 32'd0;
  ins_ACK : data = 32'd0;
  ins_IP_SRC : data = ip_src;
  ins_IP_DST : data = ip_dst;
  ins_MAC_SRC1 : data = mac_src[47:16];
  ins_MAC_SRC2 : data = {mac_src[15:0], 16'd0};
  ins_MAC_DST1 : data = mac_dst[47:16];
  ins_MAC_DST2 : data = {mac_dst[15:0], 16'd0};
  ins_PORTS : data = {port_src, port_dst};
  default : data = 32'd0;
endcase
else if (hl_state == hl_DELETING) data = 32'd0;
else data = 32'd0;
end

always_comb
begin
  if (hl_state == hl_INSERTING)
    case (ins_state)
      ins_IDLE : wren = 1'b0;
      ins_VALID_STATE : wren = 1'b1;
      ins_SEQ : wren = 1'b1;
      ins_ACK : wren = 1'b1;
      ins_IP_SRC : wren = 1'b1;
      ins_IP_DST : wren = 1'b1;
      ins_MAC_SRC1 : wren = 1'b1;
      ins_MAC_SRC2 : wren = 1'b1;
      ins_MAC_DST1 : wren = 1'b1;
      ins_MAC_DST2 : wren = 1'b1;
      ins_PORTS : wren = 1'b1;
      default : wren = 1'b0;
    endcase
  else if (hl_state == hl_DELETING) wren = 1'b1;
  else wren = 1'b0;
end
always_comb
    begin
        if (chk_state == chk_DONE) chk_done = 1'b1;
        else chk_done = 1'b0;
    end

always_ff @ (posedge clk)
    begin
        if(rst)
            begin
                chk_state <= chk_IDLE;
            end
        if(s_state == s_CHECK)
            case (chk_state)
                chk_DONE : chk_state <= chk_IDLE;
                chk_IDLE  : chk_state <= chk_WAIT;
                chk_WAIT  :
                    begin
                        chk_state <= chk_VALID;
                    end
                chk_VALID :
                    begin
                        if(q[31] == 1'b1) chk_state <= chk_IP_SRC;
                        else
                            begin
                                empty <= {1'b0, base_addr};
                                chk_state <= chk_DONE;
                            end
                    end
                chk_IP_SRC :
                    begin
                        if(q == ip_src) chk_state <= chk_IP_DST;
                        else    chk_state <= chk_DONE;
                    end
                chk_IP_DST :
                    begin
                        if(q == ip_dst) chk_state <= chk_MAC_SRC1;
                    end
            endcase
    end
else    chk_state <= chk_DONE;
end
chk_MAC_SRC1 :
begin
if(q == mac_src[47:16]) chk_state <= chk_MAC_SRC2;
else    chk_state <= chk_DONE;
end
chk_MAC_SRC2 :
begin
if(q[31:16] == mac_src[15:0]) chk_state <= chk_MAC_DST1;
else    chk_state <= chk_DONE;
end
chk_MAC_DST1 :
begin
if(q == mac_dst[47:16]) chk_state <= chk_MAC_DST2;
else    chk_state <= chk_DONE;
end
chk_MAC_DST2 :
begin
if(q[31:16] == mac_dst[15:0]) chk_state <= chk_PORTS;
else    chk_state <= chk_DONE;
end
chk_PORTS :
begin
if(q == {port_src, port_dst}) chk_state <= chk_EQUAL;
else    chk_state <= chk_DONE;
end
chk_EQUAL : chk_state <= chk_IDLE;
endcase
else if (hl_state == hl_IDLE) empty <= 8'd0;
end
always_ff @(posedge clk)
begin
if(rst)
begin
ins_state <= ins_IDLE;
ins_done <= 1'b0;
end
if(hl_state == hl_INSERTING)
    case (ins_state)
        ins_IDLE : 
            begin
                if (ins_done)
                    begin
                        ins_done <= 1'b0;
                        ins_state <= ins_IDLE;
                    end
                else ins_state <= ins_VALID_STATE;
            end
        ins_VALID_STATE : ins_state <= ins_SEQ;
        ins_SEQ : ins_state <= ins_ACK;
        ins_ACK : ins_state <= ins_IP_SRC;
        ins_IP_SRC : ins_state <= ins_IP_DST;
        ins_IP_DST : ins_state <= ins_MAC_SRC1;
        ins_MAC_SRC1 : ins_state <= ins_MAC_SRC2;
        ins_MAC_SRC2 : ins_state <= ins_MAC_DST1;
        ins_MAC_DST1 : ins_state <= ins_MAC_DST2;
        ins_MAC_DST2 : ins_state <= ins_PORTS;
        ins_PORTS :
            begin
                ins_state <= ins_IDLE;
                ins_done <= 1'b1;
            end
    endcase
end

module Packet_builder(
    input logic clk,
    input logic rst,
    output logic [7:0] addr,
    output logic [31:0] ram_in,
    input logic [31:0] ram_out,
    output logic wren,
    output logic [431:0] header, //add payload
    output logic ready,
    input logic busy);

87
parameter TCP_SENT_SYN = 31'd2;
parameter TCP_CLOSED = 31'd1;

enum logic [1:0] {s_IDLE, s_REQ, s_WAIT, s_CHECK} s_state;
enum logic [4:0] {o_IDLE, o_REQ, o_WAIT, o_CPY_SEQ, o_CPY_ACK, o_CPY_IP_SRC, o_CPY_IP_DST, o_CPY_MAC_DST1, o_CPY_MAC_DST2, o_CPY_MAC_SRC1, o_CPY_MAC_SRC2, o_CPY_PORTS, oSTALL, o_DONE} o_state;

logic found_closed;
enum logic {g_SEARCH, g_OPENING_CONNECTION} g_state;
logic [7:0] base_addr;
logic last_record;

assign ready = (o_state == o_DONE);
assign last_record = (base_addr == 8'd240);

always_ff@ (posedge clk)
begin
if (rst) g_state <= g_SEARCH;
else
  case (g_state)
    g_SEARCH :
      begin
        if (found_closed) g_state <= g_OPENING_CONNECTION;
        else g_state <= g_SEARCH;
      end
    g_OPENING_CONNECTION :
      begin
        if (ready) g_state <= g_SEARCH;
        else g_state <= g_OPENING_CONNECTION;
      end
  endcase
end

always_ff@ (posedge clk)

begin
if (rst)
begin
  s_state <= s_REQ;
  base_addr <= 8’d0;
  found_closed <= 1’b1;
end
if (g_state == g_SEARCH)
case (s_state)
s_IDLE :
  begin
    s_state <= s_REQ;
    found_closed <= 1’b0;
  end
s_REQ :
  begin
    s_state <= s_WAIT;
  end
s_WAIT :
  s_state <= s_CHECK;
s_CHECK :
  begin
    if ((ram_out[31] == 1’b1) && (ram_out[30:0] == TCP_CLOSED))
      begin
        found_closed <= 1’b1;
        s_state <= s_IDLE;
      end
    else
      begin
        s_state <= s_REQ;
        if (last_record) base_addr <= 8’d0;
        else base_addr <= base_addr + 8’d10;
      end
  end
endcase
else s_state <= s_IDLE;
end
always_ff@ (posedge clk)
  begin
    if (rst)
      begin
        o_state <= o_IDLE;
        header <= 432'd0;
      end
    else if (g_state == g_OPENING_CONNECTION)
      case (o_state)
        o_IDLE    : o_state <= o_REQ;
        o_REQ     : o_state <= o_WAIT;
        o_WAIT    :
          begin
            o_state <= o_CPY_SEQ;
            header[335:320] <= 16'h0080; //eth_type : IP
          end
        o_CPY_SEQ :
          begin
            header[311:304] <= 8'h10; //ip_tos
            header[303:288] <= 16'h0000; //ip_len FILLED AFTERWARD
            header[287:272] <= 16'h0000; //ip_id
            header[271:269] <= 3'h02;   //ip_flag = don't fragment
            header[268:256] <= 13'h0000; //frag_off
            header[255:248] <= 8'h00;   //ip_ttl
            header[247:240] <= 8'h06;   //ip_protocol : TCP (06)
            header[239:224] <= 16'h0000; //ip_checksum FILLED
          end
      endcase
  end
// [127:96] is for sequence number
// [95:64] is used for ACK when flag ACK is set
header[63:60] <= 4'h8;  //tcp_data_offset
header[59:57] <= 3'b000;  //tcp_reserved
header[56:48] <= 9'h002;  //tcp_flag
header[47:32] <= 16'h0000;  //tcp_checksum
header[31:16] <= 16'h3908;  //tcp_windowsize
header[15:0] <= 16'h0000;  //tcp_urgent pointer

end

o_CPY_SEQ :
begin
    o_state <= o_CPY_ACK;
    header[127:96] <= ram_out;
end

o_CPY_ACK :
begin
    o_state <= o_CPY_IP_SRC;
    if (header[52] == 1'b1) header[95:64] <= ram_out; //ACK is enabled
    else header[95:64] <= 32'b0;
end

o_CPY_IP_SRC :
begin
    o_state <= o_CPY_IP_DST;
    header[223:192] <= ram_out;
end

o_CPY_IP_DST :
begin
    o_state <= o_CPY_MAC_SRC1;
    header[191:160] <= ram_out;
end

o_CPY_MAC_SRC1 :
begin
    o_state <= o_CPY_MAC_SRC2;
    header[383:352] <= ram_out;
end

o_CPY_MAC_SRC2 :
begin
    o_state <= o_CPY_MAC_DST1;
    header[351:336] <= ram_out[31:16];
end
do_CPY_MAC_DST1 : begin
  o_state <= o_CPY_MAC_DST2;
  header[431:400] <= ram_out;
end
do_CPY_MAC_DST2 : begin
  o_state <= o_CPY_PORTS;
  header[399:384] <= ram_out[31:16];
end
do_CPY_PORTS : begin
  if (busy) o_state <= o_STALL;
  else o_state <= oDONE;
  header[159:128] <= ram_out;
end
o_STALL : begin
  if (busy) o_state <= o_STALL;
  else o_state <= oDONE;
end
o_DONE : o_state <= o_IDLE;
default: o_state <= o_IDLE;
endcase
end

// Define behaviour of addr
always_comb begin
  if (g_state == g_OPENING_CONNECTION) case (o_state)
    o_IDLE : addr = base_addr;
    o_REQ : addr = base_addr;
    o_WAIT : addr = base_addr + 8'd1;
    o_CPY_SEQ : addr = base_addr + 8'd2;
    o_CPY_ACK : addr = base_addr + 8'd3;
    o_CPY_IP_SRC : addr = base_addr + 8'd4;
    o_CPY_IP_DST : addr = base_addr + 8'd5;
    o_CPY_MAC_SRC1 : addr = base_addr + 8'd6;
  endcase
end
module Packetizer (input logic clk, input logic rst, input logic [431:0] header, input logic ready, output logic done, output logic [31:0] data, output logic p_start, output logic p_end, output logic [1:0] empty, input logic eth_ready, output logic eth_valid);

logic [431:0] to_send;
logic next_last;
logic [3:0] batch;
assign next_last = (batch == 4'd13);

enum logic [1:0] {s_IDLE, s_FIRST, s_BODY, s_END} s_state;

always_ff @ (posedge clk)
  begin
    if (rst) to_send <= 432'd0;
    else if (ready) to_send <= header;
  end
assign wren = (o_state == o_DONE);
assign ram_in = {1'b1, TCP_SENT_SYN};

endmodule
end

always_ff @(posedge clk)
begin
if (s_state == s_IDLE) batch <= 4'd0;
else if (eth_ready) batch <= batch + 4'd1;
end

always_ff @(posedge clk)
begin
if (rst)
begin
s_state <= s_IDLE;
end
else
case (s_state)


s_IDLE :
begin
if (eth_ready && (to_send != 432'd0)) s_state <= s_FIRST;
else s_state <= s_IDLE;
end

s_FIRST :
begin
if (eth_ready) s_state <= s_BODY;
else s_state <= s_FIRST;
end

s_BODY :
begin
if (eth_ready && next_last) s_state <= s_END;
else s_state <= s_BODY;
end

s_END :
begin
if (eth_ready) s_state <= s_IDLE;
else s_state <= s_END;
end
case endcase
end

assign p_start = (s_state == s_FIRST);
assign p_end = (s_state == s_END);
assign done = p_end;
assign empty = (batch == 4'd14) ? 2'd2 : 2'd0;
assign eth_valid = (s_state != s_IDLE);

always_comb
    case (batch)
        4'd0 :
            begin
                data[31:24] = to_send[7:0];
                data[23:16] = to_send[15:8];
                data[15:8] = to_send[23:16];
                data[7:0] = to_send[31:24];
            end
        4'd1:
            begin
                data[31:24] = to_send[39:32];
                data[23:16] = to_send[47:40];
                data[15:8] = to_send[55:48];
                data[7:0] = to_send[63:56];
            end
        4'd2:
            begin
                data[31:24] = to_send[71:64];
                data[23:16] = to_send[79:72];
                data[15:8] = to_send[87:80];
                data[7:0] = to_send[95:88];
            end
        4'd3:
            begin
                data[31:24] = to_send[103:96];
                data[23:16] = to_send[111:104];
                data[15:8] = to_send[119:112];
                data[7:0] = to_send[127:120];
            end
        4'd4:
            begin
                data[31:24] = to_send[135:128];
                data[23:16] = to_send[143:136];
                data[15:8] = to_send[151:144];
            end
data[7:0] = to_send[159:152];
end

4'd5:
begin
data[31:24] = to_send[167:160];
data[23:16] = to_send[175:168];
data[15:8] = to_send[183:176];
data[7:0] = to_send[191:184];
end

4'd6:
begin
data[31:24] = to_send[199:192];
data[23:16] = to_send[207:200];
data[15:8] = to_send[215:208];
data[7:0] = to_send[223:216];
end

4'd7:
begin
data[31:24] = to_send[231:224];
data[23:16] = to_send[239:232];
data[15:8] = to_send[247:240];
data[7:0] = to_send[255:248];
end

4'd8:
begin
data[31:24] = to_send[263:256];
data[23:16] = to_send[271:264];
data[15:8] = to_send[279:272];
data[7:0] = to_send[287:280];
end

4'd9:
begin
data[31:24] = to_send[295:288];
data[23:16] = to_send[303:296];
data[15:8] = to_send[311:304];
data[7:0] = to_send[319:312];
end

4'd10:
begin
data[31:24] = to_send[327:320];
data[23:16] = to_send[335:328];
data[15:8] = to_send[343:336];
data[7:0] = to_send[351:344];
end

4'd11:
begin
  data[31:24] = to_send[359:352];
data[23:16] = to_send[367:360];
data[15:8] = to_send[375:368];
data[7:0] = to_send[383:376];
end

4'd12:
begin
  data[31:24] = to_send[391:384];
data[23:16] = to_send[399:392];
data[15:8] = to_send[407:400];
data[7:0] = to_send[415:408];
end

4'd13:
begin
  data[31:24] = to_send[423:416];
data[23:16] = to_send[431:424];
data[15:8] = 8'd0;
data[7:0] = 8'd0;
end

default:
begin
  data[31:24] = 8'd0;
data[23:16] = 8'd0;
data[15:8] = 8'd0;
data[7:0] = 8'd0;
end
endcase

endmodule

module testbench1();
logic     clk;
logic     rst;
logic [31:0]    writedata;
logic     write;
wire [31:0]    readdata;
logic     read;
logic     chipselect;
logic [3:0]    address;

TOE t0 (.*);

initial
begin
writedata = 32'd0;
write = 1'd0;
read = 1'd0;
chipselect = 1'd0;
address = 4'd0;
end

initial clk = 1'b0;
always #20 clk = ~clk;

initial
begin
// Reset
rst = 0;
@ (posedge clk);
rst = 1;
@ (posedge clk);
rst = 0;

@ (posedge clk);
chipselect = 1'b1;
write = 1'b1;
address = 4'd3; //ip_src
writedata = 32'h11111111;
@ (posedge clk);
chipselect = 1'b0;
write = 1'b0;

@ (posedge clk);
chipselect = 1'b1;
write = 1'b1;
address = 4'd4; //ip_dst
writedata = 32'h22222222;

@ (posedge clk);
chipselect = 1'b0;
write = 1'b0;

@ (posedge clk);
chipselect = 1'b1;
write = 1'b1;
address = 4'd5; //mac_src1
writedata = 32'h33333333;

@ (posedge clk);
chipselect = 1'b0;
write = 1'b0;

@ (posedge clk);
chipselect = 1'b1;
write = 1'b1;
address = 4'd6; //mac_src2
writedata = 32'h33330000;

@ (posedge clk);
chipselect = 1'b0;
write = 1'b0;

@ (posedge clk);
chipselect = 1'b1;
write = 1'b1;
address = 4'd7; //mac_dst1
writedata = 32'h44444444;
@ (posedge clk);
chipselect = 1'b0;
write = 1'b0;

@ (posedge clk);
chipselect = 1'b1;
write = 1'b1;
address = 4'd7; //mac_dst2
writedata = 32'h44440000;

@ (posedge clk);
chipselect = 1'b0;
write = 1'b0;

@ (posedge clk);
chipselect = 1'b1;
write = 1'b1;
address = 4'd7; //port_src
writedata = 32'h5555;

@ (posedge clk);
chipselect = 1'b0;
write = 1'b0;

@ (posedge clk);
chipselect = 1'b1;
write = 1'b1;
address = 4'd8; //port_dst
writedata = 32'h6666_0000;

@ (posedge clk);
chipselect = 1'b0;
write = 1'b0;

@ (posedge clk);
chipselect = 1'b1;
write = 1'b1;
address = 4'd0; //req
writedata = 32'h4000_0000;

@ (posedge clk);
write = 1'b0;
address = 4'd2;
read = 1'b1;

@ (posedge clk);
wait(readdata != 32'h240);

@ (posedge clk);
chipselect = 1'b1;
write = 1'b1;
read = 1'b0;
address = 4'd0; //req
writedata = 32'd0;

end

endmodule

/*******************testbench2.sv******************************/
module testbench2();

logic        clk;
logic        rst;
logic [31:0]  writedata;
logic         write;
wire [31:0]   readdata;
logic         read;
logic         chipselect;
logic [3:0]   address;

TOE t0 (.*);
initial
begin
    writedata = 32'd0;
    write = 1'd0;
    read = 1'd0;
    chipselect = 1'd0;
    address = 4'd0;
end

initial clk = 1'b0;
always #20 clk = ~clk;

initial
begin
    // Reset
    rst = 0;
    @ (posedge clk);
    rst = 1;
    @ (posedge clk);
    rst = 0;

    // FIRST REQUEST
    @ (posedge clk);
    chipselect = 1'b1;
    write = 1'b1;
    address = 4'd3; //ip_src
    writedata = 32'h11111111;

    @ (posedge clk);
    chipselect = 1'b0;
    write = 1'b0;

    @ (posedge clk);
    chipselect = 1'b1;
    write = 1'b1;
    address = 4'd4; //ip_dst
    writedata = 32'h22222222;
@ (posedge clk);
chipselect = 1'b0;
write = 1'b0;

@ (posedge clk);
chipselect = 1'b1;
write = 1'b1;
address = 4'd5; //mac_src1
writedata = 32'h33333333;

@ (posedge clk);
chipselect = 1'b0;
write = 1'b0;

@ (posedge clk);
chipselect = 1'b1;
write = 1'b1;
address = 4'd6; //mac_src2
writedata = 32'h33330000;

@ (posedge clk);
chipselect = 1'b0;
write = 1'b0;

@ (posedge clk);
chipselect = 1'b1;
write = 1'b1;
address = 4'd7; //mac_dst1
writedata = 32'h44444444;

@ (posedge clk);
chipselect = 1'b0;
write = 1'b0;

@ (posedge clk);
chipselect = 1'b1;
write = 1'b1;
address = 4'd7; //mac_dst2
writedata = 32'h44440000;
@ (posedge clk);
chipselect = 1'b0;
write = 1'b0;

@ (posedge clk);
chipselect = 1'b1;
write = 1'b1;
address = 4'd7; //port_src
writedata = 32'h5555;

@ (posedge clk);
chipselect = 1'b0;
write = 1'b0;

@ (posedge clk);
chipselect = 1'b1;
write = 1'b1;
address = 4'd8; //port_dst
writedata = 32'h6666_0000;

@ (posedge clk);
chipselect = 1'b0;
write = 1'b0;

@ (posedge clk);
chipselect = 1'b1;
write = 1'b1;
address = 4'd0; //req
writedata = 32'h4000_0000;

@ (posedge clk);
write = 1'b0;
address = 4'd2;
read = 1'b1;

@ (posedge clk);
wait (readdata != 32'd0);
@ (posedge clk);
chipselect = 1'b1;
write = 1'b1;
read = 1'b0;
address = 4'd0;  //req
writedata = 32'd0;

// SECOND IDENTICAL REQUEST
@ (posedge clk);
chipselect = 1'b1;
write = 1'b1;
address = 4'd3;  //ip_src
writedata = 32'h11111111;

@ (posedge clk);
chipselect = 1'b0;
write = 1'b0;

@ (posedge clk);
chipselect = 1'b1;
write = 1'b1;
address = 4'd4;  //ip_dst
writedata = 32'h22222222;

@ (posedge clk);
chipselect = 1'b0;
write = 1'b0;

@ (posedge clk);
chipselect = 1'b1;
write = 1'b1;
address = 4'd5;  //mac_src1
writedata = 32'h33333333;

@ (posedge clk);
chipselect = 1'b0;
write = 1'b0;
@ (posedge clk);
chipselect = 1'b1;
write = 1'b1;
address = 4'd6; //mac_src2
writedata = 32'h33330000;

@ (posedge clk);
chipselect = 1'b0;
write =1'b0;

@ (posedge clk);
chipselect = 1'b1;
write = 1'b1;
address = 4'd7; //mac_dst1
writedata = 32'h44444444;

@ (posedge clk);
chipselect = 1'b0;
write = 1'b0;

@ (posedge clk);
chipselect = 1'b1;
write = 1'b1;
address = 4'd7; //mac_dst2
writedata = 32'h44440000;

@ (posedge clk);
chipselect = 1'b0;
write = 1'b0;

@ (posedge clk);
chipselect = 1'b1;
write = 1'b1;
address = 4'd7; //port_src
writedata = 32'h5555;

@ (posedge clk);
chipselect = 1'b0;
write = 1'b0;
@ (posedge clk);
chipselect = 1'b1;
write = 1'b1;
address = 4'd8; //port_dst
writedata = 32'h6666_0000;

@ (posedge clk);
chipselect = 1'b0;
write = 1'b0;

@ (posedge clk);
chipselect = 1'b1;
write = 1'b1;
address = 4'd0; //req
writedata = 32'h4000_0000;

@ (posedge clk);
write = 1'b0;
address = 4'd2;
read = 1'b1;

@ (posedge clk);
@ (posedge clk);
wait(readdata != 32'd0);

@ (posedge clk);
chipselect = 1'b1;
write = 1'b1;
read = 1'b0;
address = 4'd0; //req
writedata = 32'd0;

end

dendmodule
module PB_testbench;

/*Packet_builder*/
logic clk;
logic reset;
reg [8:0] address;
reg [31:0] ram_in;
wire [31:0] ram_out;
logic wren;
reg [8:0] empty;

initial
begin
wren=1;
address=4'b0;
end

initial
begin
clk=1'b0;
forever
#20 clk= ~clk;
end

initial
begin
#40 ram_in=00000000000000000000000000000001; //valid-bit-high
#80 ram_in=32'b1;
#120 ram_in=32'b0;
#160 ram_in=32'b1;
#200 ram_in=32'b0;
#240 ram_in = 32'b1;
#260 ram_in=32'b0;
#300 ram_in=32'b1;
#340 wren=~wren;
end

Packet_builder_c pb0(.);
endmodule
# A Tcl script for the Qsys system console

# Start Qsys, open your soc_system.qsys file, run File->System Console,
# then execute this script by selecting it with Ctrl-E

# The System Console is described in Chapter 10 of Volume III of
# the Quartus II Handbook

# Alternately,
# system-console --project_dir=. --script=syscon-test.tcl

# system-console --project_dir=. -cli
# and then "source syscon-test.tcl"

# Base addresses of the peripherals: take from Qsys
set toe_init 0x0
puts "Started system-console-test-script"

# Using the JTAG chain, check the clock and reset

set j [lindex [get_service_paths jtag_debug] 0]
open_service jtag_debug $j
puts "Opened jtag_debug"

puts "Checking the JTAG chain loopback: [jtag_debug_loop $j {1 2 3 4 5 6}]"
jtag_debug_reset_system $j

puts -nonewline "Sampling the clock: "
foreach i {1 1 1 1 1 1 1 1 1 1} {
    puts -nonewline [jtag_debug_sample_clock $j]
}
puts ""

puts "Checking reset state: [jtag_debug_sample_reset $j]"
close_service jtag_debug $j
puts "Closed jtag_debug"

# Perform bus reads and writes

set m [lindex [get_service_paths master] 0]
open_service master $m
puts "Opened master"

puts "Request connection #1"
# Write a test pattern to the various registers
foreach {r v} {4 0x00000000 8 0x00000000 12 0x11111111 16 0x22222222 20 0x33333333 24 0x33330000 28 0x44444444 32 0x44440000 36 0x55550000 40 0x66660000 0x40000000} {
    master_write_32 $m [expr $toe_init + $r] $v
}
puts "Waiting for the request to be processed"
# Wait until the connection has been added to the RAM
puts "Reply #1 : [master_read_32 $m [expr $toe_init + 8] 0x1]"
puts "Request has been processed successfully"
master_write_32 $m $toe_init 0x0
puts "Request deasserted"

puts "Request connection #2"
# Write a test pattern to the various registers
foreach {r v} {4 0x00000000 8 0x00000000 12 0xffffff00 16 0xffffff00 20 0xffffff00 24 0xffffff00 28 0xffffff00 32 0xffffff00 36 0xffffff00 40 0xffffff00 0x40000000} {
    master_write_32 $m [expr $toe_init + $r] $v
}
puts "Waiting for the request to be processed"
# Wait until the connection has been added to the RAM
puts "Reply #2 : [master_read_32 $m [expr $toe_init + 8] 0x1]"
puts "Request has been processed successfully"
master_write_32 $m $toe_init 0x0
puts "Request deasserted"

puts "Request connection #1 again"
# Write a test pattern to the various registers
foreach {r v} {4 0x00000000 8 0x00000000 12 0x11111111 16 0x22222222 20 0x33333333 24 0x33330000 28 0x44444444 32 0x44440000 36 0x55550000 40 0x66660000 0 0x40000000} {
    master_write_32 $m [expr $toe_init + $r] $v
}
puts "Waiting for the request to be processed"
# Wait until the connection has been added to the RAM
puts "Reply #3 : [master_read_32 $m [expr $toe_init + 8] 0x1]"
puts "Request has failed ERR_FOUND"
master_write_32 $m $toe_init 0x0
puts "Request deasserted"

puts "Request deletion of connection #2"
# Write a test pattern to the various registers
foreach {r v} {4 0x73000000 8 0x00000000 12 0x00000000 16 0x00000000 20 0x00000000 24 0x00000000 28 0x00000000 32 0x00000000 36 0x00000000 40 0x00000000 0 0x80000000} {
    master_write_32 $m [expr $toe_init + $r] $v
}
puts "Waiting for the request to be processed"
# Wait until the connection has been added to the RAM
puts "Reply #4 : [master_read_32 $m [expr $toe_init + 8] 0x1]"
puts "Request succeeded"
master_write_32 $m $toe_init 0x0
puts "Request deasserted"

puts "Request connection #2 again"
# Write a test pattern to the various registers
foreach {r v} {4 0x00000000 8 0x00000000 12 0xffffffff 16 0xffffffff 20 0xffffffff 24 0xffffffff 28 0xffffffff 32 0xffffffff 36 0xffffffff 40 0xffffffff 0 0x40000000} {
    master_write_32 $m [expr $toe_init + $r] $v
}
puts "Waiting for the request to be processed"
# Wait until the connection has been added to the RAM
puts "Reply #2 : [master_read_32 $m [expr $toe_init + 8] 0x1]"
puts "Request has been processed successfully"
master_write_32 $m $toe_init 0x0
puts "Request deasserted"

close_service master $m
puts "Closed master"
# A Tcl script for the Qsys system console

# Start Qsys, open your soc_system.qsys file, run File->System Console,
# then execute this script by selecting it with Ctrl-E

# The System Console is described in Chapter 10 of Volume III of
# the Quartus II Handbook

# Alternately,
# system-console --project_dir=. --script=syscon-test.tcl
#
# system-console --project_dir=. -cl
#  and then "source syscon-test.tcl"

# Base addresses of the peripherals: take from Qsys
set toe_init 0x0
set fifo_out 0x40

puts "Started system-console-test-script"

# Using the JTAG chain, check the clock and reset"

set j [lindex [get_service_paths jtag_debug] 0]
open_service jtag_debug $j
puts "Opened jtag_debug"

puts "Checking the JTAG chain loopback: [jtag_debug_loop $j {1 2 3 4 5 6}]"

jtag_debug_reset_system $j

puts -nonewline "Sampling the clock: "
foreach i {1 1 1 1 1 1 1 1 1 1 1 1} {
    puts -nonewline [jtag_debug_sample_clock $j]
}
puts ""
puts "Checking reset state: [jtag_debug_sample_reset $j]"

close_service jtag_debug $j
puts "Closed jtag_debug"

# Perform bus reads and writes

set m [lindex [get_service_paths master] 0]
open_service master $m
puts "Opened master"

puts "Request connection #1"
# Write a test pattern to the various registers
foreach {r v} {4 0x00000000 8 0x00000000 12 0x11111111 16 0x22222222 20 0x33333333 24 0x33333333 28 0x44444444 32 0x44440000 36 0x55550000 40 0x66660000 0 0x40000000} {
    master_write_32 $m [expr $toe_init + $r] $v
}
puts "Waiting for the request to be processed"
# Wait until the connection has been added to the RAM
puts "Reply #1 : [master_read_32 $m [expr $toe_init + 8] 0x1]"
puts "Request has been processed successfully"
master_write_32 $m $toe_init 0x0
puts "Request deasserted"

puts "The Creation of an entry in the RAM should lead to the outputting of a SYN packet"
puts "****** READING FIFO ******"
for {set i 0} {$i < 14} {incr i} {
    puts"[master_read_32 $m 0x40 0x1]"
}

close_service master $m
puts "Closed master"