Final Report: Hardware Accelerated Market Order Packet Generation

Ankur Gupta, Dhananjay Palshikar, Mithila Paryekar, Sushant Bhardwaj, Yasser Mohammed
Under the guidance of Prof. Stephen Edwards
Department of Electrical Engineering
School of Engineering and Applied Science, Columbia University in the City of New York
{ ag3187, dp2575, mmp2178, sb3322, ym2364}@columbia.edu

Abstract

The designed system aims at accelerating the release of packets on the network. Optimization is achieved in terms of reducing the latency, decreasing the data uploaded on the Avalon bus which will eventually lead to power optimization. A software application running on a soft-processor would change the transaction data going over the network in runtime. This document describes the overall architecture of the system, along with describing the design of the custom ethernet accelerator. Typically the ethernet controller should be capable of receiving and sending data over the network. Our implementation accelerates the sending of data to the network while receiving of data is handled in software.

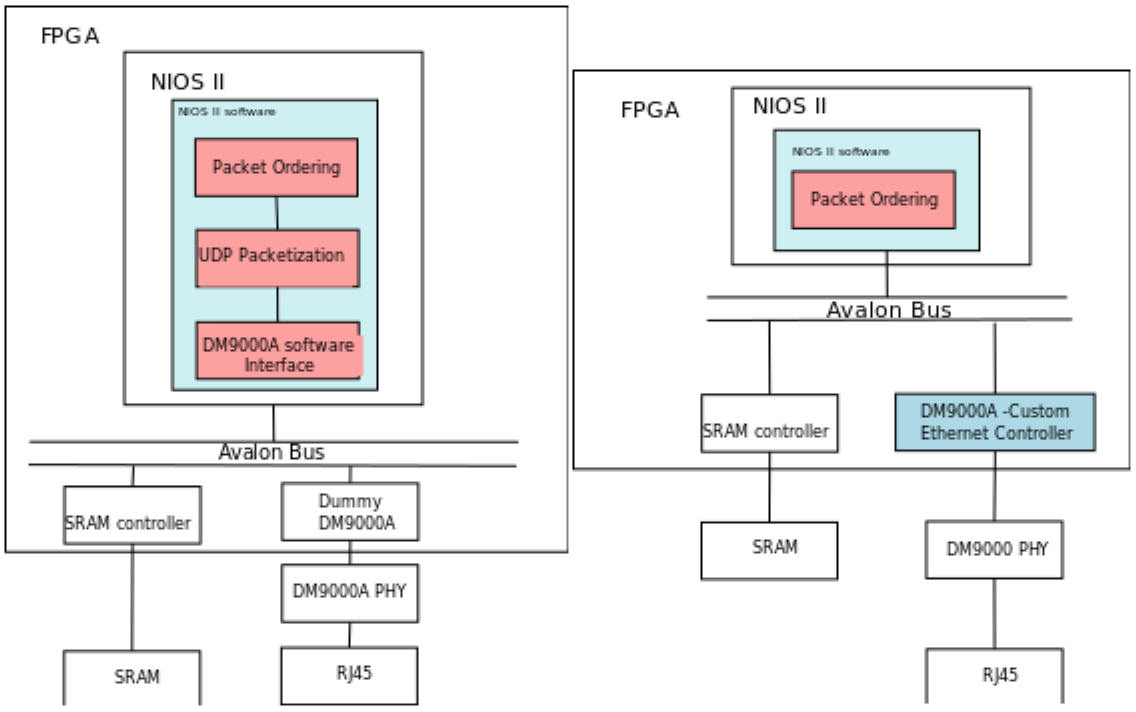
Keywords: FPGA, Nios II, ethernet accelerator

1 Introduction

Various applications nowadays require the fast transmission of data in the order of gigabytes per second. High frequency trading used in stock markets is one such example. A major bottleneck in any such system is the overhead of the operating system due to the limitation of CPU speed. This occurs as packetization is implemented using a software application. The speed of these systems can be greatly increased if packetization is done in hardware instead. The duration of time we minimized starts from the point when the user initiates the transaction through a software application up until the point when the ethernet controller puts the data on the network. Our system implements the transport layer processing along with the software interface of DM9000A ethernet controller on an FPGA. The FPGA also has an on board soft-processor which serves as an end point and a source of input for our custom hardware component. The system will be developed for the Altera DE-2 FPGA board.

2 Architecture

In the Lab 2 assignment of the Embedded Systems course, we implemented an UDP-chat application which implemented the UDP packetization in software. The architecture of the Lab 2 assignment is shown in Fig:1(a), and is further described in the following section. The system we implemented has been shown in Fig:1(b). The accelerated ethernet controller listens on the Avalon bus, for data from the application.



(a) *(b)*

Fig1: (a) schematic of system used in Lab2 (b) schematic system we have implemented

2.1 Lab 2 Architecture

In the Lab 2 assignment the software on the NIOS-2 generated data in the form of UDP packets. It was then sent to the ethernet controller via the DM9000A software interface, as can be seen from Fig:1(a). Every time there was data to be sent, the entire packet was sent to the ethernet controller. In terms of a stock market, an architecturally similar system will have the application running on an operating system, which adds to its overheads. The software application will have to compete with other processes in order to get scheduled. Although Lab 2 does not have an operating system on the NIOS-II processor, its architecture is similar in the sense that UDP protocol is handled in the software. Lab 2 provided us a good starting point for the design of our project.

2.2 System Architecture

Data packets sent over the network follow a protocol similar to the OUCH protocol. OUCH is a digital communications protocol which allows customers of the NASDAQ (National Association of Securities Dealers Automated Quotations) to conduct business in the options market adapted for streaming (FAST) [1], where often the difference in contents of the UDP packet are restricted to a few fields in the payload, (assuming UDP header remains the same; fixed sender, fixed recipient). Thus, our system implemented the UDP packet generation on the FPGA, in effect replacing the NIOS II software block in Fig:1(a), with the DM9000A block in Fig:2(a). The software component on NIOS would now only send limited filed information amounting to 13 bytes (see Table1).

2.3 DM9000A accelerated block for UDP

Our task for the project was to build the DM9000A block as an Avalon peripheral on the Altera-II FPGA. This peripheral stores a generic pre-decided UDP packet in its internal memory. Every time a request on the Avalon Bus is made by the NIOS-II, the packet payload contents are minimally modified and passed over to the DM9000A PHY interface. The DM9000A accelerated block has the following components:

- Avalon Peripheral Component, for reading from the Avalon bus.
- UDP Packetization Component, for modifying the contents of UDP packets
- DM9000A initialization component adapted from Lab2, to initialize the ethernet Physical Layer Component for the first time a packet is sent.
- DM9000A Communication component: sends UDP packets to the physical layer.

The component was designed in VHDL and synthesized and deployed on the FPGA chip on-board the Altera DE-2 evaluation Board. The next section shows the format of UDP payload used for the project.

2.4 UDP-Payload Format

The numbers seen with the field names indicate the size of each field in bytes.

Price (4)	Name (4)	Buy/Sell (1)	Quantity (4)

Fig2: UDP Payload Format

2.5 Protocol for DM9000A

In order to minimize communication over the Avalon bus, we designed a protocol which could eliminate the need for the DM9000A controller to wait for each field, in case only a subset of fields has changed. Table 2, shows the format of instruction which would be sent over the Avalon bus. The *Offset* field in the protocol decides which address has to be written to. Within the DM9000A controller this is interpreted as the payload field to be modified. *Wait* tells the controller to wait for more fields and not send the packet right away. *Data* carries the content of the modified field.

Wait (Y/N)	Data

Fig3: Protocol for DM9000A

Price (00)	Name (01)	Quant (10)	Buy/Sell (11)

Fig 4:codes for different payload fields as sent over address (1 downto 0)

3 Implementation

3.1 DM9000A Custom Hardware

The DM9000A has been created as an all-encompassing unit which contains the following components:

- Automata: the central logic of our custom hardware which accomplishes the basic read and write commands to the PHY in hardware. In addition to these functions, it also contains the following entities:
    - Initializer: this component was adopted from the DM9000.v file of the Lab2 by converting the Verilog logic into VHDL. It basically hardwires the input commands coming from the NiOS to the output ports of the DM9000A. It does the job of initializing the DM9000A hardware on the Altera through a software program.
    - Multiplexer (MUX): initially designed as a separate component it was finally adapted into the automata for simpler functionality. This component switches between the Initializer and actual custom hardware logic when the software has completed the initialization of DM9000A hardware.
- Iterator: it accomplishes the functions required to transmit a packet in hardware similar to that done in the Lab2 DM9000.c software program. It has been designed to loop over the automata; thereby using the basic read and write commands but changing the value of the address and data in accordance to the sequence of registers mentioned in the DM9000A.c program.

The detailed design and logic of these components has been mentioned in the following sections.

3.1.1 Automata
Filename: DM9000_custom_all.vhd

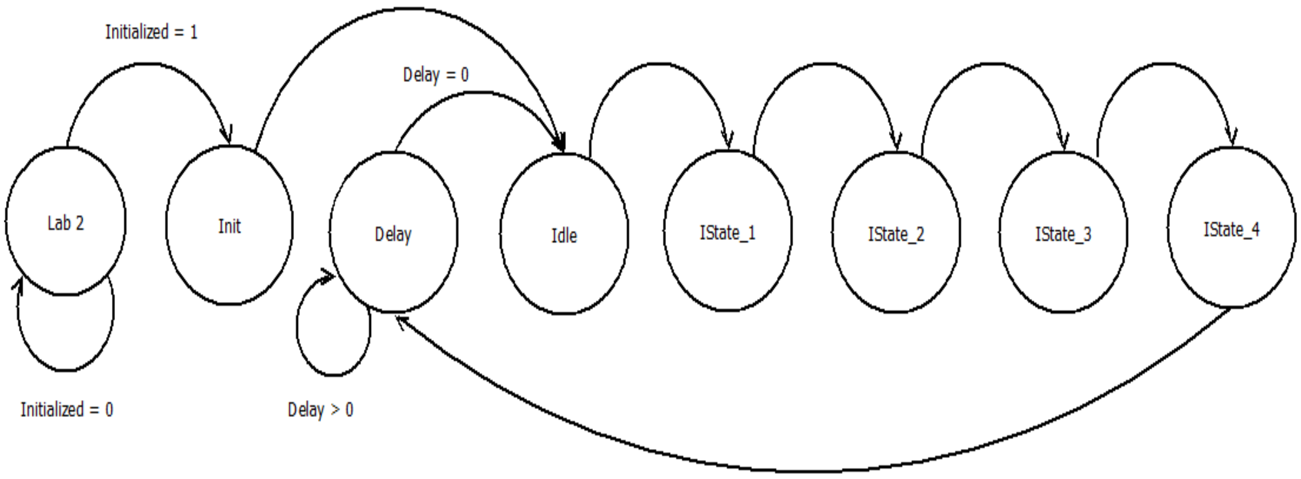


Fig4 : State Diagram of Automata

Ports:

NiOS Avalon Interface

- iDATA: data written from the NiOS on the Avalon Bus
- iCMD: informs whether address of the register or the data is to be written. Address is transmitted when iCMD is 0 and data is transmitted when iCMD is 1.
- iRD_N: indicates NiOS wants to read when this goes low
- iWR_N: indicates NiOS wants to write when this goes low
- iCS_N: chipslect
- iRST_N: reset signal
- oDATA: data read by the NiOS over the Avalon bus
- oINT : ineterrupt port
- iCLK: internal clock of the NiOS

DM9000A Interface

- ENET_DATA: data line from the NiOS to read and write from and to the PHY
- ENET_CMD: indicates if data or address is being transmitted to the PHY
- ENET_RD_N: indicates to the PHY that DMA9000A wants to read data
- ENET_WR_N: indicates to the PHY that DMA9000A wants to write data
- ENET_CS_N: chipselect from the DM9000A
- ENET_RST_N: reset
- ENET_INT: interrupt from the DM9000A

Timing Diagram:

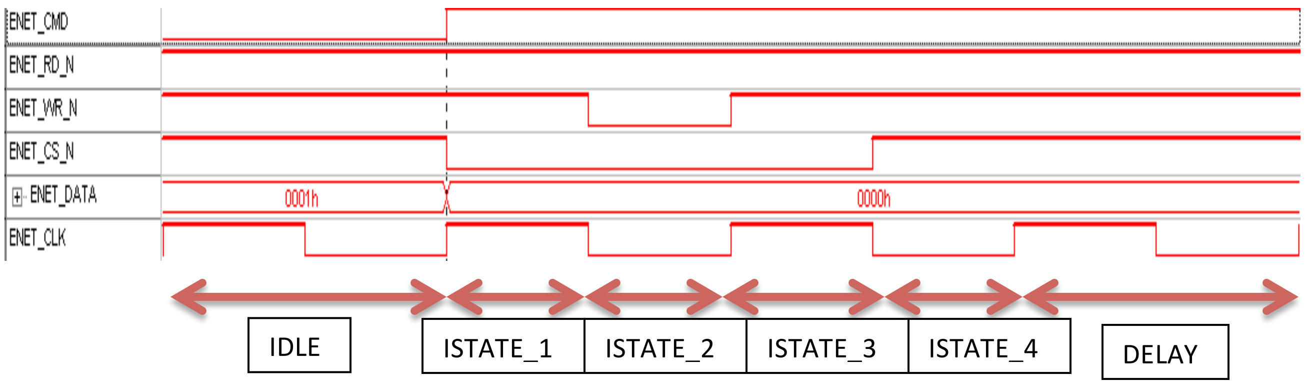


Fig5: Write Cycle

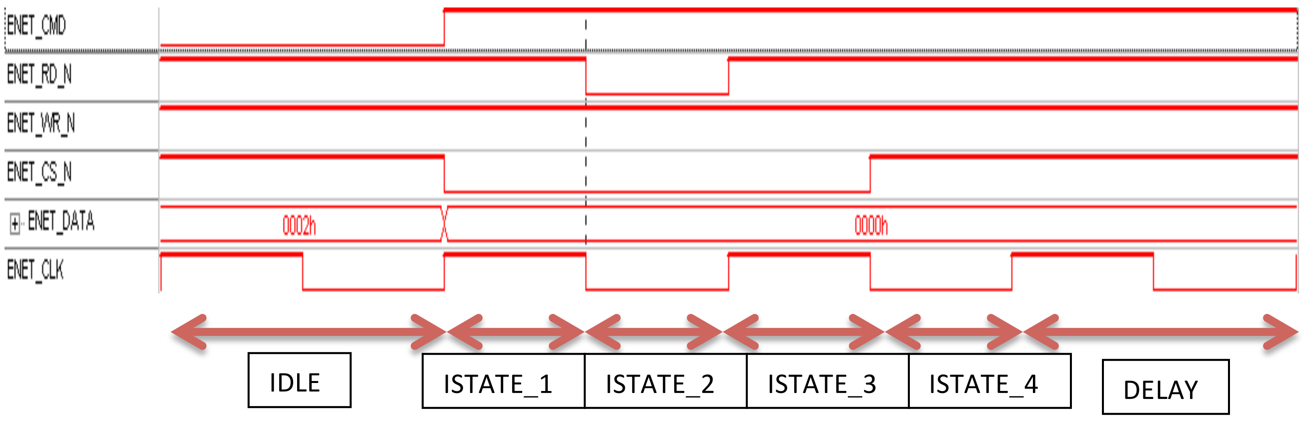


Fig6: Read Cycle

The automata was designed to accomplish the read and write to and from the PHY in hardware as done by the dm9000a_iow and dm9000a_ior functions in software. We designed it a Finite State Machine with 8 states. These states consisted of a LAB2 state for initialization and 7 other states which accomplished the core read and write logics. The LAB2 state also contained the logic for accomplishing the MUX. The FSM has been designed using two processes, one which initiates the state transition and the other which handles the functions inside each state. The transition from one state to another occurred at rising clock edges of the iCLK. Given below is a detailed explanation of what each state accomplishes:

- LAB2: this state represents the state at which DM900A is being initialized in software. It waits for a confirmation from the software over a signal from the software. When this signal is set as high then the FSM moves onto the next state, until then it remains in the LAB 2 state. We have realized the functioning of the MUX using this initializing signal.

- INIT: the functioning of the custom hardware begins from this state. No functionality occurs at this state. It has been used so as to obtain the correct timing of functions occurring in the next state.

- IDLE: this state has the same functionality as the previous state and has been used so that all updates from the iterator are received by the custom hardware before the chip select goes low

- ISTATE_1: In this state the ENET_CS_N is set low. This informs the PHY that some data transaction is required between it and the custom hardware.

- ISTATE_2: the iterator sends out an is_read command when it wishes to read to data from the PHY rather than write to it. In this state, if the is_read signal is low, the custom hardware sets the ENET_WR_N signal low so as to inform the PHY about a write cycle. Else the ENET_RD_N command is set low so as to initiate a read cycle.

- ISTATE_3: the ENET_WR_N or the ENET_RD_N come back to their normal (high) state in the next clock cycles after ISTATE_1
- ISTATE_4: in the next clock cycle after which the PHY interface read and write signals return to their high state, the PHY chip select also returns high. This is depicted in this state.

- DELAY: between a register address write command and a following data write command (write cycle) or a data read command (read cycle), there exists a delay of 20µs. This has been accomplished in hardware in this state. A delay register has been defined, which contains the value 00003E8 in HEX which when decremented over the NiOS time period of 200ns results in the FSM staying at the DELAY state for 20 µs.

This entire component has then been added to the SOPC to be integrated with the NiOS processor and SRAM.

3.1.2Iterator:
Filename: iterator.vhd

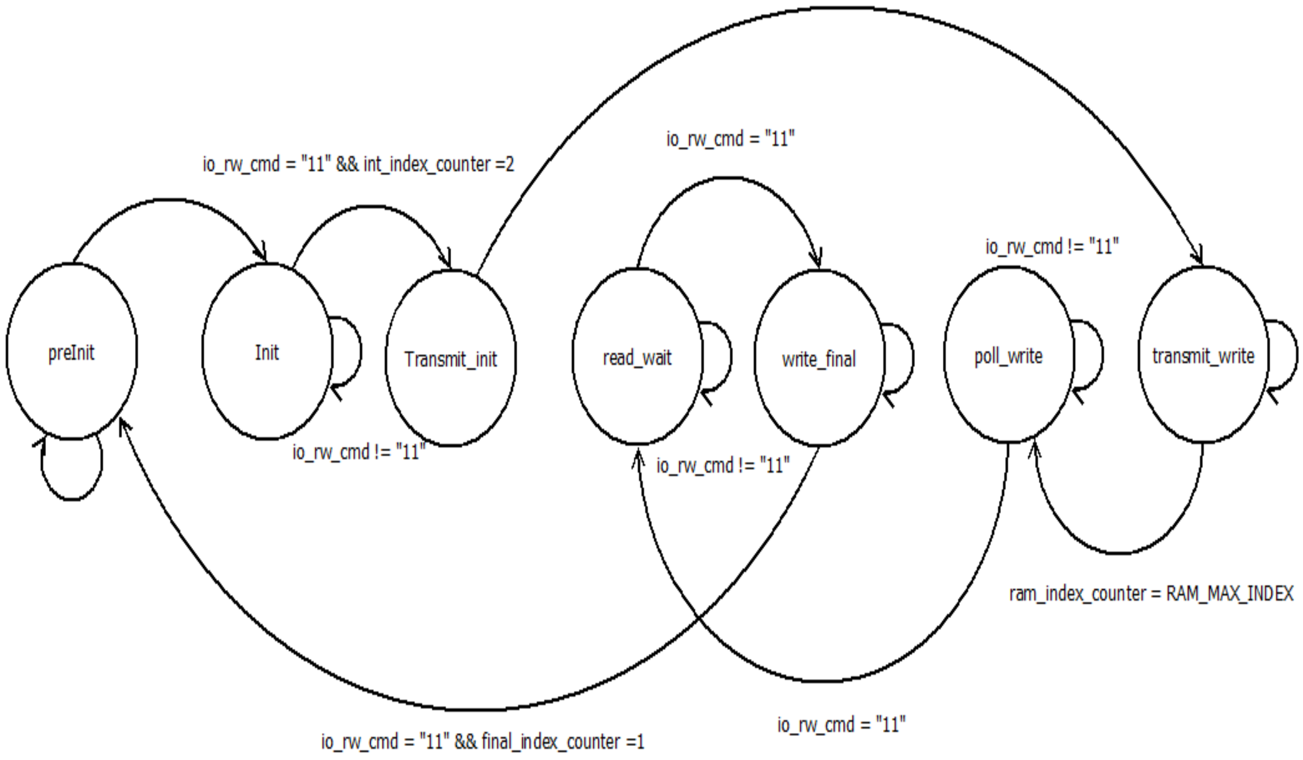


Fig7: State Diagram of Iterator

Ports:

- iCLK: input clock
- req_update: set by the automata at the end of the DELAY state. Indicates register values are required
- input_data: data from read operations is transferred to the iterator through this
- ENET_DATA: data line of the DM9000A
- is_read: output port to inform the automata of a read operation
- ENET_CMD: address/data selector of the DM9000A

The iterator was also designed as a FSM in order to implement the TransmitPacket function from DM900A.c file. It was then included as a component of the custom hardware and was called whenever the signal on the request update port when high. Then the iterator loops over the entire custom hardware, thereby using the read and write mechanisms therein to read and write several registers to and from the PHY.

The iterator FSM consists of 7 states, each state signifying a certain functionality which needs to be undertaken in order to send packets from the RAM on to the PHY and finally over the Ethernet. Similar to the automata, the FSM has been designed as two processes, one to do the state transitions and other to implement the actual working of the state. The first loop is sensitive to the time when DELAY state of the automata is complete and the req_update signal is raised. The state changes after that occurs at rising clocks of the iCLK. The functionality achieved in each step has been explained below:

- preINIT: this is the first state that is reached after the DELAY counter in the automata has completed its decrement. The RAM index counter which keeps track of the number of elements from the RAM that have been transmitted is refreshed to 0 in this state.

- INIT: this state two registers, Interrupt Mask Register and Transmit Packet Length (TXPL) register are initialised to their initial values. The value of x”80” is written into the IMR so as to enable the SRAM read/write pointer to return to the start address. The TXPL register has a lower byte and a higher byte which are written to in consecutive clock cycles. This register stores the length of the packet we will be sending. In our case we write the values x”00” and x”37” to the higher and lower bytes respectively in order to indicate that our packet length is of 55 bytes.

- Transmit_Init: this state initialises the INDEX port in order to copy the data of the packet ino the TX FIFO of the internal SRAM

- Transmit_write: the actual writing of data onto the output port occurs in this state and the entire contents of the internal RAM of the DM9000A are written on to the ENET_DATA line. The FSM keeps looping in this state as long as the RAM index counter has not reached the maximum index value and it updates the value of the RAM index in every clock cycle.

- Poll_write: a polling command is issued in this state by writing and then reading the value of Transmit Control Register in this state.

- Read_wait: the Network Status Register is read in delays of 20 μs in order to find out of the bits number 2 and 3 have been cleared or not.

- Write_Final: the Network Status Register is cleared in this state and interrupts are re-enabled of the Network Interface Controller

3.2 Avalon Protocol

As explained in the introduction, this protocol has been adopted so that the DM9000A does not need to wait for all the fields of our UDP payload to be in case only a subset of the field needs to be updated. This entity serves as the packet updater of our system as and when new packets become available. We have designed this system so as to contain two components:

- Change packet hardware: this component interacts with the iterator and updates the contents of the RAM after one payload has been transmitted completely
- Change packet Software: this program performs register writes and updates the values of BUY_SELL, PRICE, NAME and QUANT through the NiOS and Avalon.

3.2.1Change Packet Hardware
Filename: change_pack.vhd

Ports:

- clk: a 25 MHz clock which runs this component
- reset_n: reset signal from the NiOS
- read_n: signal to indicate that NiOS wants to read from this component
- write_n: signal to indicate that NiOS wants to write to this component
- chipselect_n: chip select signal from the NiOS
- address: the lower two bits on this port inform which field of the UDP payload is being modified, the higher MSB indicates if it is a magic packet or not
- readdata: data to the NiOS is sent over this port
- writedata: data from the NiOS is received on this port
- irq: interrupt request sent to the NiOS
- BUY_SELL, PRICE, NAME and QUANT: output ports which carry the corresponding data as in the UDP payload. Their sizes are the same mentioned in the UDP packet.
- wait_t: when low, it indicates there are more fields to be updated. It is communicated by the NiOS on the MSB of the writedata

Functionality:

From the UDP payload diagram, it is seen that 15 bits of data are transmitted at one time. As each register of the payload field is 4 byte long and the length of the Avalon bus is 16 bits, fields such as NAME, PRICE and QUANT are transmitted in 3 packets from the NiOS, while the BUY/SELL field requires a single packet. The core functionality of this hardware is to receive these packets from the NiOS and integrate them into one to store it in the RAM. After each packet is received, writedata's MSB is checked to find out if there are more packets are expected or not.

At the end of this integration the interrupt is raised so as to inform the iterator (via the NiOS) that new packets are available.

3.2.2 Change Packet Software
Filename: main.c

The software does a series of register writes so as to update the new values of the payload fields. As mentioned in the previous section, the NAME, PRICE and QUANT are written in 3 packets while the BUY/SELL is written in one.

4. Workflow and Challenges

4.1 Custom Hardware and Iterator

We started our project by going through the Lab2 components and getting a thorough understanding of how the DM9000A works and how packets are being written to the PHY in software. The working of lab2 hardware was then implemented using just the basic components necessary for our project and porting the Verilog code to VHDL format. It was verified that the lab2 software runs on the new SOPC component created.

Next, we generated a basic design of our custom hardware to include a few basic states. This hardware was created to replicate the IOWR and IORD commands of the dm9000a_iow and dm9000a_ior functions, as seen in the DM9000A.c software file. We found out that this design did not work as we had not accounted for the RD_N, WR_N and CS_N which were required to inform the PHY about how the DM900A wished to interact with it. At this point were faced with another problem as we had ignored the delays we need to introduce on the Avalon Slave timing. This had led to errors such as the RD_N and WR_N and CS_N not going low as they were in Lab 2. After discussions with the receiving group we realised that we needed to mention these delays in the SOPC generator. We finally added a 20ns delay for setup, read, write and hold.

After further analysis of the DM9000A software, we found that in order to transmit packets we needed to first read and write data to a set of registers. In essence this meant going through the address and data calls over and over again for several calls. To accomplish this we created a component called iterator and interfaced it with the custom hardware. Control is handed over to the iterator every time the custom hardware completed one entire cycle of operation. The iterator then updated the address and data values in accordance with the TransmitPacket function in the DM9000A software.

4.2 MUX

The entire functioning of our system can be divided into two core functions. Initializing the PHY on the board and making the custom hardware transmit packets through the PHY. In order to use the PHY on the Altera board, it has to be initialized using a sequence of read and write commands. We wished to accomplish this initialization in software using the DM9000A software while the sending of packets was being handled in hardware. Hence we were essentially first using a hardware component to interface with the initialization software and then passing on control over to the custom hardware. To accomplish this we needed to create a multiplexer. What this multiplexer would do was to first interface the NiOS ports such as iCMD, iRD_N, IWR_N etc with the Lab2 hardware and upon initialization interface these with the custom hardware. The Lab2 hardware and the custom hardware were to be added as entities in the MUX file.

We initially converted the Lab2 Verilog file into VHDL and added it as an entity in our multiplexer file. During this conversion we made a mistake of declaring the ENET_WR_N port as bidirectional.

This lead to the WR_N going low in both read and write cycles thereby leading to a failure in initialization. Although we eventually corrected this we noticed that the Lab2 was not getting initialized correctly.

After analysis we figured out that this was because we had added this MUX file onto the SOPC which did not work well with ‘work.entity’ definitions as well as with bidirectional ports. After weeks of debugging and design changes we decided to drop this design of an independent MUX file and a separate Lab2 hardware component. We then moved on the design of the custom hardware as it stands now having the Lab2 as a state of the FSM and the MUX implemented through a signal coming from the software file. Also we revamped the design of bidirectional signals such as ENET_DATA to read from iDATA during read cycles and to write to the oDATA port during write cycles.

Source Code:

DM9000A_custom_all.vhd

```
--this file as of 7 th Arpil 2012 6:17 pm is modified version of the ...software_like file in
--dp2575 account. This adds the ability to write and subsequently read the same register . This
--is done for 3 registers.
--this file is a copy of Dm9000A-custom_software_like_dj.vhd , name changed for simlation
--dhananjay



library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity DM9000A_custom_all is
port(
-- NIOS Avalon Interface
                    iDATA : in std_logic_vector(15 downto 0); --data
                    --iCMD: in std_logic_vector (1 downto 0);--address
                    iCMD: in std_logic_vector (3 downto 0);--address
                    iRD_N: in std_logic;--read_n active low, when 0, NIOS wants to read
                    iWR_N: in std_logic;--write_n active low, when 0, NIOs wants to read
                              iCS_N: in std_logic;--chipselect
                              iRST_N: in std_logic;--reset_n
                              oDATA: out std_logic_vector(15 downto 0);---writedata
                              oINT: out std_logic;--Interrupt irq
iCLK : in std_logic;
--DM9000A interface

          ENET_DATA:inout std_logic_vector(15 downto 0);
ENET_CMD: out std_logic;
ENET_RD_N: out std_logic;
ENET_WR_N: out std_logic;
ENET_CS_N: out std_logic;
ENET_RST_N: out std_logic;
ENET_INT: in std_logic;

--DEBUG interface
STATE_OUT: out std_logic_vector(3 downto 0);
DELAY_COUNTER:out std_logic_vector(27 downto 0);
DBUG_Reg_Index:out std_logic_vector(5 downto 0);
dbug_io_rw_cmd:out std_logic_vector(1 downto 0);
dbug_state_out_iter: out std_logic_vector(2 downto 0);
dbug_ENET_DATA:out std_logic_vector(15 downto 0);
dbug_req_update:out std_logic



);
end DM9000A_custom_all;

architecture behav of DM9000A_custom_all is

--signals to map the data from the change packet to the iterator
signal BUY_SELL_p: std_logic;
signal PRICE_p: std_logic_vector(31 downto 0);
signal NAME_p: std_logic_vector(31 downto 0);
signal wait_t_p: std_logic;
signal QUANT_p:  std_logic_vector(31 downto 0);
signal sent_state_p: std_logic;
signal save_state: std_logic;



--mealy machine
```

```
 type state is (LAB2,INIT,IDLE,DELAY,IState_1,IState_2,IState_3,IState_4); -- READ , WRITE
from the perpective
--of this device
  signal state_sig: state:=LAB2;
 signal sig_req_update:std_logic:='0';
 signal sig_ENET_DATA: std_logic_vector(15 downto 0):=x"0000";
 signal sig_ENET_CMD:std_logic:='0';
 signal sig_is_read:std_logic:='0';
signal sig_iCMD:std_logic_vector(1 downto 0):="00";
 signal address_done:bit:='0';
 signal input_data: unsigned(15 downto 0):=x"0000";
 signal data_done,init_done:bit:='0';
 constant STD_DELAY_PHY:unsigned(27 downto 0):=x"00001F4";--"EE6B280"; --10 secs (freq of
NIOS=50MHz)
 signal delay_std_counter:unsigned(27 downto 0):=STD_DELAY_PHY;--2000
signal sig_cust_free: std_logic:='1';
signal initialized: std_logic:='0';
signal check_intialized: std_logic_vector(3 downto 0):=x"3";
signal reg_state_sig:state:= LAB2;
signal input_data_internal:std_logic_vector(15 downto 0):= x"0000";
signal read_now_N:std_logic:='1'; -- same as ENET_RD_N, used for reading exactly when the data
is to be read
--- this handles the global state and the outputs
---can relate to the lines of code in C,see its file for more on this
component iterator
port(	ENET_CLK: in std_logic;
	req_update: in std_logic;
	ENET_CMD :out std_logic;
	ENET_DATA:out std_logic_vector(15 downto 0);--for verifying when looping over read,and
for writing data
	input_data: in std_logic_vector(15 downto 0); -- for reading ENET_DATA, inout is not the
best option,need to have 2 separate I/O
	is_read : out std_logic;	--simply means do not write to the bus(if
high)dbug_io_rw_cmd: out std_logic_vector(1 downto 0);
```

```
		BUY_SELL_i : in std_logic;
	PRICE_i: in std_logic_vector(31 downto 0);
	NAME_i :  in std_logic_vector(31 downto 0);
	wait_t_i: in std_logic;
	QUANT_i : in std_logic_vector(31 downto 0);
	sent_state_i: out std_logic; --Set high when packet sent successfully
		dbug_io_rw_cmd: out std_logic_vector(1 downto 0);
		dbug_reg_index: out std_logic_vector(1 downto 0);
		dbug_state_out: out std_logic_vector(2 downto 0);
		dbug_req_update: out std_logic

);
end component;

component change_packet
port(

	clk   : in std_logic;                  -- Should be 25.125 MHz
	reset_n : in std_logic;
	read_n : in std_logic;
	write_n : in std_logic;
	chipselect_n : in std_logic;
	address : in std_logic_vector(2 downto 0);
	readdata : out std_logic_vector(15 downto 0);-- this is the data that the nios wants us to wite
	writedata : in std_logic_vector(15 downto 0); -- this is sent from the nios
	sent_state : in std_logic;
	BUY_SELL: out std_logic;
	PRICE : out std_logic_vector(31 downto 0);
	NAME :  out std_logic_vector(31 downto 0);
	wait_t: out std_logic := '1';
	QUANT : out std_logic_vector(31 downto 0)
);
end component;
```

```
begin
--main process



E1: iterator port map(
            ENET_CLK=>iCLK,
            req_update=> sig_req_update,
            ENET_DATA=> sig_ENET_DATA,
            is_read=> sig_is_read,
            ENET_CMD => sig_ENET_CMD,
            input_data=>input_data_internal,
            dbug_io_rw_cmd=>dbug_io_rw_cmd,
            dbug_reg_index=> DBUG_Reg_Index(1 downto 0),
            dbug_state_out=> dbug_state_out_iter,
            dbug_req_update=>dbug_req_update,

    BUY_SELL_i => BUY_SELL_p,
    PRICE_i => PRICE_p,
    NAME_i => NAME_p,
    wait_t_i=> wait_t_p,
    QUANT_i => QUANT_p,
    sent_state_i => save_state

);

E2: change_packet port map(
    clk   => iCLK,
    reset_n => iRST_N,
    read_n => iRD_N,
    write_n => iWR_N,
    chipselect_n => iCS_N,
    address => iCMD,
    readdata => oDATA,
    writedata => iDATA,
   irq => oINT,

  BUY_SELL => BUY_SELL_p,
  PRICE => PRICE_p,
  NAME => NAME_p,
  QUANT => QUANT_p,
  wait_t => wait_t_p,
  sent_state => sent_state_p
);



process (iCLK)
begin
if (rising_edge(iCLK)) then

sig_req_update <= '0';
reg_state_sig <= state_sig;
state_sig <= LAB2;
case state_sig is
                    when LAB2=>
                              sig_cust_free <= '1';
                              if initialized = '1' then
                                        state_sig <= INIT;
                              else
                                        state_sig<= LAB2;
                              end if;

when INIT=>
          state_sig <= IDLE;
          delay_std_counter <= STD_DELAY_PHY;
          sig_req_update <= '1';



when IDLE=>
```

```
          state_sig <= IState_1;
          delay_std_counter <= STD_DELAY_PHY;
when IState_1=>
          state_sig <= IState_2;
          delay_std_counter <= STD_DELAY_PHY;

when IState_2=>
          state_sig <= IState_3;
          delay_std_counter <= STD_DELAY_PHY;

when IState_3=>
          state_sig <= IState_4;
          delay_std_counter <= STD_DELAY_PHY;

when IState_4=>
          state_sig <= DELAY;
          delay_std_counter <= STD_DELAY_PHY;



when DELAY=>

if delay_std_counter > x"00000" then
          delay_std_counter <= delay_std_counter -1;
          state_sig <= DELAY;
else  --delay afer idle state
          delay_std_counter <= STD_DELAY_PHY;
          sig_req_update <= '1';
          state_sig <= IDLE;

end if;
when others=>



end case;
end if;
end process;



oData <= ENET_DATA ;
input_data_internal <= ENET_DATA(15 downto 0) when read_now_N = '0';

sig_iCMD <= iCMD;
--ENET_CMD <= iCMD(0) when state_sig=LAB2 else sig_ENET_CMD;

dbug_ENET_DATA <= std_logic_vector(input_data);
ENET_DATA <= iDATA when iWR_N='0' and state_sig=LAB2
else sig_ENET_DATA when sig_is_read='0' and state_sig/=LAB2
else (others=>'Z');

process
(state_sig,delay_std_counter,iCMD,iRD_N,ENET_INT,iCS_N,iWR_N,iRST_N,input_data_internal
)
begin
--Defualt
        ENET_CS_N <= '1';
        ENET_WR_N <='1';
        ENET_RD_N <='1';
        ENET_RST_N <= '1' ;
        oINT <= '0';
        ENET_CMD <= '0';
        read_now_N <= '1';
        STATE_OUT <= x"A";
--      ENET_DATA <= (others => 'Z');
        input_data <= (others => 'Z');

case state_sig is
when LAB2=>
        STATE_OUT <= x"9";
        ENET_RD_N <= iRD_N ;
        ENET_WR_N <= iWR_N ;
        ENET_CS_N <= iCS_N ;
        ENET_RST_N <= iRST_N ;
        ENET_CMD <= sig_iCMD(0);
        --if iWR_N ='0' then
        --      ENET_DATA <= iDATA;
--       else
        --      ENET_DATA <= (others => 'Z');
--      end if;
        ENET_CMD <= iCMD(0);
        oINT <= ENET_INT;
when INIT=>
        STATE_OUT <= x"0";

when IDLE=>
        STATE_OUT <= x"1";
        ENET_CMD <= sig_ENET_CMD;
        DELAY_COUNTER <= std_logic_vector(delay_std_counter);
--      if sig_is_read ='0' then
--              ENET_DATA(7 downto 0) <= sig_ENET_DATA;  --eventually a reset should also
be used
--      end if;

when IState_1 =>
        STATE_OUT <= x"2";
        ENET_WR_N <= '1';
        ENET_RD_N <= '1';
        ENET_CMD <= sig_ENET_CMD;
        ENET_CS_N <= '0';
        DELAY_COUNTER <= std_logic_vector(delay_std_counter);
--if sig_is_read ='0' then
--      ENET_DATA(7 downto 0) <= sig_ENET_DATA;  --eventually a reset should also be used
```

```
--end if;
--

when IState_2 =>
        STATE_OUT <= x"3";
        ENET_CMD <= sig_ENET_CMD;
        ENET_CS_N <= '0';
        DELAY_COUNTER <= std_logic_vector(delay_std_counter);
if sig_is_read ='0' then
--      ENET_DATA(7 downto 0) <= sig_ENET_DATA;  --eventually a reset should also be used

        ENET_WR_N <= '0';
else
--      --input_data_internal <= (ENET_DATA(7 downto 0));
----    input_data <= unsigned(ENET_DATA);
        ENET_RD_N <= '0';
        read_now_N <= '0';
end if;

when IState_3 =>
        STATE_OUT <= x"4";
        ENET_CMD <= sig_ENET_CMD;
        ENET_CS_N <= '0';
        DELAY_COUNTER <= std_logic_vector(delay_std_counter);
--if sig_is_read ='0' then
--      ENET_DATA(7 downto 0) <= sig_ENET_DATA;  --eventually a reset should also be used

--end if;


when IState_4 =>
        STATE_OUT <= x"5";
        ENET_CMD <= sig_ENET_CMD;
        ENET_CS_N <= '1';
        DELAY_COUNTER <= std_logic_vector(delay_std_counter);
--if sig_is_read ='0' then
--      ENET_DATA(7 downto 0) <= sig_ENET_DATA;  --eventually a reset should also be used

--end if;

when DELAY=>
        STATE_OUT <= x"6";
        ENET_CMD <= sig_ENET_CMD;
        DELAY_COUNTER <= std_logic_vector(delay_std_counter);
        ENET_WR_N <= '1';
        input_data <= unsigned(input_data_internal);

                --ENET_DATA <= std_logic_vector(input_data);
when others=>
        STATE_OUT <= x"1";
        DELAY_COUNTER <= std_logic_vector(delay_std_counter);

end  case;



end process;



process(iClk)
begin
 --avalon logic for intialization
if rising_edge(iClk) then
        if iCS_N = '0' then
                if sig_iCMD(1) = '1' then
                        if iWR_N = '0' then
                                check_intialized <= iDATA(3 downto 0);
                        end if;
                end if;
        end if;
        if check_intialized = x"5" then
                initialized <= '1';
        end if;
        end if;
end process;

process(iCLK)
if rising_edge(iClk) then
if wait_t_p = '0'then
save_state <= '0';
else
save_state <= '1';
end if;
end process;




end behav;
```

Change_pack.vhd

```
--The avalon bus for writedata is 16 bits wide
-- Out of 16 bits, higher 2 bits are offset
-- 00 offset means price field is being sent, 01 means name field, 10 means quantity and 11 means
buy_sell
-- price_t, name_t and quant_t are 32 bit vectors containing the corresponding values received from
NIOS
-- wait_t signal is active low when there is no more data is being sent by NIOS and the values can be
passed to the iterator
```

```
-- When WAIT signal from the nios is high it means that there are further changes to be made to the
document else not.

library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
use ieee.std_logic_unsigned.all;

entity change_pack is

  port (

    clk   : in std_logic;                    -- Should be 25.125 MHz
    reset_n : in std_logic;
    read_n: in std_logic;
    write_n : in std_logic;
    chipselect_n : in std_logic;
    --address : in std_logic_vector(15 downto 0); -- this sends the offset value of the data that is being
sent. Only 1 downto 0 is to be used
    address : in std_logic_vector(1 downto 0);
    --readdata : out std_logic_vector(15 downto 0);-- this is the data that the nios wants us to wite
    writedata : in std_logic_vector(15 downto 0); -- this is sent from the nios

    -- these are the output ports that have to be port mapped to the top level entity
    BUY_SELL: out std_logic;
    PRICE : out std_logic_vector(31 downto 0);
    NAME :  out std_logic_vector(31 downto 0);
    wait_t: out std_logic; --active low, checks if more packet fields are being sent from NIOS
    QUANT : out std_logic_vector(31 downto 0);
    sent_state: in std_logic -- Set high from iterator when packet sent successfully

    --HEX00, HEX11, HEX22, HEX33, HEX44, HEX55, HEX66, HEX77 -- to give to 7-segment
displays
     --  : out std_logic_vector(6 downto 0):="1111111"
```

```
           );


end change_pack;

architecture packet of change_pack is

 signal packet_type:std_logic;
 signal counter:std_logic_vector(1 downto 0):="00";
 signal price_t: std_logic_vector(31 downto 0);
 signal name_t: std_logic_vector(31 downto 0);
 signal quant_t: std_logic_vector(31 downto 0);

 signal ram_address: std_logic_vector(1 downto 0); -- this is the signal that will store the offset value
from the bus
 --signal sent_state: std_logic := '1';

 begin

 process (clk)
           begin
           if rising_edge(clk) then
             if reset_n = '0' then
             --readdata <= (others => '0');
         counter <= (others => '0');
         wait_t <= '1';

         else
         --PRICE <= price_t(31 downto 0);
         PRICE <= X"00000000";
           if chipselect_n = '0' then
                 --if (address >= X"0000" and address <= X"ffff") then
```

```
        --Storing the offset value sent from nios and using the case statement to check what type
of packet it is
            ram_address <= address(1 downto 0);
            if write_n = '0' then

                case ram_address is
                  when "00" =>  --The total size of the packet is 32 bits and is sent in 3
packets from nios
                                if counter = "00" then
                               --Storing the first part of the packet
                                price_t(14 downto 0) <= writedata(14 downto 0);
                                counter <= counter + 1;
                                elsif counter = "01" then
                               --Storing the second part of the packet
                                price_t(29 downto 15) <= writedata(14 downto 0);
                                counter <= counter + 1;
                                elsif counter = "10" then
                               --Storing the last part of the packet
                                price_t(31 downto 30) <= writedata(1 downto 0);
                                counter <= "00";
                                --PRICE(31 downto 0) <= price_t(31 downto 0);

                                      if writedata(15) = '0' then
                                      wait_t <= '0';
                                      elsif writedata(15) = '1' then
                                      wait_t <= '1';
                                       end if;
                                end if;


                  when "01" =>    if counter = "00" then
                                name_t(14 downto 0) <= writedata(14 downto 0);

                                counter <= counter + 1;
```

```
                        elsif counter = "01" then

                        name_t(29 downto 15) <= writedata(14 downto 0);
                        counter <= counter + 1;

                        elsif counter = "10" then

                        name_t(31 downto 30) <= writedata(1 downto 0);
                        counter <= "00";

                        NAME(31 downto 0) <= name_t(31 downto 0);

                            if writedata(15) = '0' then
                            wait_t <= '0';
                            else
                            wait_t <= '1';
                             end if;
                        end if;


                when "10" =>   if counter = "00" then
                        quant_t(14 downto 0) <= writedata(14 downto 0);
                        counter <= counter + 1;
                        elsif counter = "01" then
                        quant_t(29 downto 15) <= writedata(14 downto 0);
                        counter <= counter + 1;
                        elsif counter = "10" then
                        quant_t(31 downto 30) <= writedata(1 downto 0);
                        counter <= "00";
                        QUANT(31 downto 0) <= quant_t(31 downto 0);
                            if writedata(15) = '0' then
                            wait_t <= '0';
                            else
```

```
                                                  wait_t <= '1';
                                                   end if;
                                             end if;


                                when "11" =>    BUY_SELL <= writedata(0);
                                                if writedata(15) = '0' then
                                                  wait_t <= '0';
                                                  else
                                                  wait_t <= '1';
                                                   end if;



                                end case;




                        end if;
                        else
                              --PRICE(31 downto 0) <= price_t(31 downto 0);
                              PRICE <= price_t(31 downto 0);
                    end if;
                   -- PRICE <= X"00000000";
                end if;
              end if;



end process;

--process(sent_state)
--begin
--        if(sent_state = '1') then
--        wait_t <= '1';
--        end if;
```

```
--end process;

--Check if packet has been sent successfully by the iterator


--process (clk)
--  begin
--    if rising_edge(clk) then
--      if reset_n = '0' then
--        irq <= '0';
--      else
--        if sent_state ='1' then --ideally there should only be 1 packet that should be transmitted
--          irq <= '1';
--          --sent_state <= '0';
--          wait_t <= '1';
--        elsif write_n = '0' and chipselect_n = '0' and address(2 downto 0) = "100" then
--          irq <= '0';  -- important to reset the irq
--        end if;
--      end if;
--    end if;
--  end process;



    end packet;
```

Iterator.vhd

```
--this file iterates over the FSM defined in  custom_hardware.vhd which is basically a simple generic
read/write to
--registers, the state diagram for this is file is;
--+------+        +----------------------------+        +-----------------+        +-----------+
--| INIT | -->  |Transmit_init(1 half write)  | --> |Transmit_write(n)| --> | Poll_write |
--+------+        +----------------------------+        +-----------------+        +-----------+
--                                                                                          |
--                                                                      +-------+:-------------------------------------
-----+
--                                                                               |Read  wait(1  infinite  read)  -->
write_final(2 writes)|
--                                                                               +------------------------------------
-------------+
--
--
--
library ieee;
 use ieee.std_logic_1164.all;
 use ieee.numeric_std.all;

 entity iterator is
        port(ENET_CLK: in std_logic;
                req_update: in std_logic;
                input_data: in std_logic_vector(15 downto 0);
                ENET_DATA:out std_logic_vector(15 downto 0);--for verifying when looping over
read,and for writing data
                is_read : out std_logic;        --simply means do not write to the bus(if high)
                ENET_CMD: out std_logic;
                -- DEBUG interface
                dbug_io_rw_cmd: out std_logic_vector(1 downto 0);
```

```
                    dbug_reg_index: out std_logic_vector(1 downto 0);
                    dbug_state_out: out std_logic_vector(2 downto 0);
                    dbug_req_update:out std_logic;
                    --Defining the ports to get the values of various fields
    BUY_SELL_i : in std_logic;
    PRICE_i:  in std_logic_vector(31 downto 0);
    NAME_i :  in std_logic_vector(31 downto 0);
    wait_t_i: in std_logic;
    QUANT_i : in std_logic_vector(31 downto 0);
    sent_state_i: out std_logic --Set high when packet sent successfully
);

 end iterator;
 architecture behav of iterator is

 constant RAM_MAX_INDEX :integer := 54; --Actual size of RAM is 55 bytes ,
                                              --42 bytes of header, 13 bytes of header
--indicating position in RAM
 signal sig_input_data:std_logic_vector(15 downto 0):=x"0000";
 type finance_data is array(3 downto 0)of std_logic_vector(7 downto 0);

  constant PRICE: finance_data:= (x"00",x"00",x"00",x"26");
  constant NAME: finance_data:= (x"00",x"00",x"00",x"27");
  constant BUY_SELL: std_logic_vector:= "00000001";
  constant QUANT: finance_data:= (x"00",x"00",x"00",x"28");

  --signal  PRICE: finance_data:= (x"00",x"00",x"00",x"26");
  --signal  NAME: finance_data:= (x"00",x"00",x"00",x"27");
  --signal  QUANT: finance_data:= (x"00",x"00",x"00",x"28");
  --signal BUY_SELL: std_logic_vector(7 downto 0):= "00000001";

--declaring register addresses for DMA9000a PHY
  constant NCR: std_logic_vector(7 downto 0):= x"00"; --Network  Control Register REG. 00
  constant NSR: std_logic_vector(7 downto 0):= x"01"; --Network  Status Register  REG. 01
  constant IMR:std_logic_vector(7 downto 0):=x"FF";  -- NIC Interrupt Mask   Register REG. FFH
  constant PAR_set:std_logic_vector(7 downto 0):=x"80"; -- IMR REG. FFH: PAR only, RX/TX
FIFO R/W
  constant MWCMD:std_logic_vector(7 downto 0):=x"F8"; --TX FIFO I/O port command WRITE
into TX FIFO
  constant IO_addr:std_logic:= '0';
  constant IO_data:std_logic:= '1';
  constant TCR:std_logic_vector(7 downto 0):= X"02";
  constant TCR_set:std_logic_vector(7 downto 0):= X"00";
  constant TX_REQUEST:std_logic_vector(7 downto 0):= X"01";
  constant INTR_set:std_logic_vector(7 downto 0):= X"81";
  type init_address_data_type is array(integer range 0 to 2)of std_logic_vector(7 downto 0);
  type state is (preINIT,INIT,  Transmit_init, Transmit_write, Poll_write, Read_wait, Write_final);--
see state digram for explanations
  signal state_sig:state:= preINIT;
  signal state_sig_out:integer:=0;

  signal io_rw_cmd:unsigned(1 downto 0):="00";-- Write Cycle(00 -address,01 data), Read Cycle(10
add,11 data)
  --declaring arrarys
  -- Intial 3 register that are written to via phy_write
  type address_data_type is array(integer range 0 to 2)of std_logic_vector(7 downto 0);
  signal addr_tx_init:address_data_type:=(IMR,x"FD",x"FC");
  --signal addr_tx_init:address_data_type:=(x"28",x"29",x"FC"); --trying to read product id
  signal data_tx_init:address_data_type:=(PAR_set,x"00",x"37"); --PAR_set address and data values
used in the tx intialization stored sequentially inthe same seq as used in the C file
  signal init_index_counter:unsigned(1 downto 0):="00";
 --MWCMD
 --writing data from RAM array
 type byte_array is array(integer range 0 to RAM_MAX_INDEX)of std_logic_vector(7 downto 0);
 signal RAM : byte_array:=( x"FF", x"FF", "11111111", "11111111", "11111111", "11111111",
 x"01", x"60", x"6E", x"11", x"02", x"0F",
 x"08", x"00",
 x"45",
 x"00",
 x"00",x"9C",

 x"3d", x"35",
 x"00",
 x"00",

 x"80",
 x"11",
 x"00",x"00",
 x"c0",x"a8",x"01",x"01",
 x"c0",x"a8",x"01",x"ff",


 x"67",x"d9",
 x"27",x"2b",
 x"00",x"88",
 x"00",x"00", -- 41 upto here
 PRICE(3),PRICE(2),PRICE(1),PRICE(0),
  NAME(3),NAME(2),NAME(1),NAME(0),
 BUY_SELL,
  QUANT(3),QUANT(2),QUANT(1),QUANT(0));
 signal ram_index_counter:unsigned(5 downto 0):="000000";

 ---register written to after transmit data is written
-- dm9000a_iow(TCR , TCR_set | TX_REQUEST)

--read poll register address
---dm9000a_ior(NSR)

--final writes
```

```
 --address and data values used in the tx intialization stored sequentially inthe same seq as used in
the C file
type final_data_type is array(integer range 0 to 1)of std_logic_vector(7 downto 0);
signal final_index_counter:unsigned(0 downto 0):="0";
signal addr_tx_final:final_data_type:=(NSR,IMR);
signal data_tx_final:final_data_type:=(x"00",INTR_SET);

begin

process(req_update,ENET_CLK,input_data) -- only sensitive when the 'delay state starts',changes
state and counters
begin
                    --defaults --
                    if rising_edge(ENET_CLK) then
                    dbug_req_update <= req_update;
                    if req_update = '1' then --if it is init or delay

                              case state_sig is

                              when preINIT=> --hack for causing a change in stage_sig
                                        state_sig <= INIT;
                                        ram_index_counter <= "000000";
                              when INIT =>
                                        sent_state_i <= '0';
                                        if io_rw_cmd = "11" then
                                                  io_rw_cmd <= "00";
                                                  if init_index_counter < 2 then
                                                            init_index_counter <= init_index_counter +1;
                                                            state_sig <= INIT;
                                                  else
                                                            state_sig <= Transmit_Init;
                                                            init_index_counter <= "00";

                                                  end if;
```

```
                        else
                                io_rw_cmd <= io_rw_cmd + 1;
                                state_sig <= INIT;

                        end if;

                when Transmit_Init =>
                        io_rw_cmd    <=    io_rw_cmd    +    1;    --packets    to    be    written    in
DATA_WRITE mode
                        state_sig <= Transmit_write;

                when Transmit_write =>
                        if ram_index_counter < RAM_MAX_INDEX then
                                ram_index_counter <= ram_index_counter +2;
                                state_sig <= Transmit_write;
                                io_rw_cmd <= "01";
                        else
                                state_sig <= Poll_write;
                                ram_index_counter <= "000000";
                                io_rw_cmd <= "00";
                        end if;

                when Poll_write =>
                        if io_rw_cmd = "11" then
                                io_rw_cmd <= "10"; --Intiating Read Cycle(Address write) for
Poll_write
                                state_sig <= Read_wait;
                        else
                                state_sig <= Poll_write;
                                io_rw_cmd <= io_rw_cmd + 1;
                        end if;
                when Read_wait=>
                        if io_rw_cmd ="11" then
                                if (input_data(2) or input_data(3)) = '1' then
```

```
                    io_rw_cmd<="00";
                    state_sig <= write_final;
                else
                    io_rw_cmd <= "10";
                    state_sig <= Read_wait;
                end if;
            else
                io_rw_cmd <= "11";
            end if;

        when Write_final=>
            if io_rw_cmd = "11" then
                io_rw_cmd <= "00";
                if final_index_counter = "0"  then
                    final_index_counter <= "1";
                    state_sig <= Write_final;
                else
                    --state_sig <= INIT;
                    final_index_counter <= "0";
                    sent_state_i <= '1';
                    if wait_t_i = '0'  then
                    RAM(50) <=  "0000000" & BUY_SELL_i;

                    RAM(45) <= PRICE_i(7 downto 0);
                    --RAM(45) <= "11111111";
                    RAM(44) <= PRICE_i(15 downto 8);
                    RAM(43) <= PRICE_i(23 downto 16);
                    RAM(42) <= PRICE_i(31 downto 24);

                    RAM(49) <= NAME_i(7 downto 0);
                    RAM(48) <= NAME_i(15 downto 8);
                    RAM(47) <= NAME_i(23 downto 16);
                    RAM(46) <= NAME_i(31 downto 24);
```

```
                                        RAM(54) <= QUANT_i(7 downto 0);
                                        RAM(53) <= QUANT_i(15 downto 8);
                                        RAM(52) <= QUANT_i(23 downto 16);
                                        RAM(51) <= QUANT_i(31 downto 24);


                                        state_sig <= INIT;

                                        end if;


                                end if;
                        else
                                io_rw_cmd <= io_rw_cmd + 1;
                                state_sig <= Write_final;

                        end if;

                when others=>


                end case;

            end if;
        end if;
end process;

process(state_sig,init_index_counter,io_rw_cmd,ram_index_counter,final_index_counter)  --changes
the outputs
begin

        ENET_DATA <= x"0000";
        is_read <='0';
        ENET_CMD<= io_rw_cmd(0);
        dbug_io_rw_cmd<= std_logic_vector(io_rw_cmd);
        dbug_state_out <=std_logic_vector(to_unsigned(state_sig_out,3));
        dbug_reg_index <= std_logic_vector(init_index_counter);
        case state_sig is
                when preINIT=>
                        state_sig_out <= 1;
                when INIT =>
                        state_sig_out <= 2;
                        case io_rw_cmd is
                                --write cycle
                                when "00" => --write addr to DM9000a
                                        ENET_DATA(7        downto        0)        <=
addr_tx_init(to_integer(init_index_counter));
                                when "01" =>--write data to DM9000a
                                        ENET_DATA(7        downto        0)        <=
data_tx_init(to_integer(init_index_counter));
                                --read cycle
                                when "10"=> --write addr to DM9000a
                                        ENET_DATA(7        downto        0)        <=
addr_tx_init(to_integer(init_index_counter));
                                when "11"=> --read from DM9000a
                                        is_read <= '1';
                        end case;

                when Transmit_Init =>
                        ENET_DATA(7 downto 0) <= MWCMD;
                        state_sig_out <= 3;

                when Transmit_write =>
                        if ram_index_counter = 54 then
                                ENET_DATA <= x"00" & RAM(to_integer(ram_index_counter));
                        else
                                ENET_DATA    <=    RAM(to_integer(ram_index_counter    +1))    &
RAM(to_integer(ram_index_counter));
```

```
          end if;
          state_sig_out <= 4;
when Poll_write =>
          state_sig_out <= 5;
case io_rw_cmd is
                    --write cycle
                    when "00" => --write addr to DM9000a
                              ENET_DATA(7 downto 0) <= TCR;
                    when "01" =>--write data to DM9000a
                              ENET_DATA(7 downto 0)<= (TCR_set or TX_REQUEST);
                    --read cycle
                    when "10"=> --write addr to DM9000a
                              ENET_DATA(7 downto 0) <= TCR;
                    when "11"=> --read from DM9000a
                              is_read <= '1';
end case;

when Read_wait=>
          state_sig_out <= 6;
case io_rw_cmd is
                    --read cycle
                    when "10"=> --write addr to DM9000a
                              ENET_DATA(7 downto 0) <= NSR;
                    when "11"=> --read from DM9000a
                              is_read <= '1';
                    when others=>
end case;

when write_final=>
          state_sig_out <= 7;
case io_rw_cmd is
                    --write cycle
                    when "00" => --write addr to DM9000a
```

```
                                        ENET_DATA(7              downto              0)              <=
addr_tx_final(to_integer(final_index_counter));
                                when "01" =>--write data to DM9000a
                                        ENET_DATA(7              downto              0)              <=
data_tx_final(to_integer(final_index_counter));
                                --read cycle
                                when "10"=> --write addr to DM9000a
                                        ENET_DATA(7              downto              0)              <=
addr_tx_final(to_integer(final_index_counter));
                                when "11"=> --read from DM9000a
                                        is_read <= '1';
                end case;

                when others=>


        end case;

end process;
end behav;
```

Packet_hard_soft.vhd (Top-level entity)

```
--
-- DE2 top-level module that includes the simple VGA raster generator
--
-- Stephen A. Edwards, Columbia University, sedwards@cs.columbia.edu
--
-- From an original by Terasic Technology, Inc.
-- (DE2_TOP.v, part of the DE2 system board CD supplied by Altera)
--

 library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity packet_hard_soft is

  port (
    -- Clocks

    CLOCK_27,                               -- 27 MHz
    CLOCK_50,                               -- 50 MHz
    EXT_CLOCK : in std_logic;                       -- External Clock

    -- Buttons and switches

    KEY : in std_logic_vector(3 downto 0);         -- Push buttons
    DPDT_SW : in std_logic_vector(17 downto 0);         -- DPDT switches

    -- LED displays

    HEX0, HEX1, HEX2, HEX3, HEX4, HEX5, HEX6, HEX7 -- 7-segment displays
       : out std_logic_vector(6 downto 0);
    LED_GREEN : out std_logic_vector(8 downto 0);       -- Green LEDs
    LED_RED : out std_logic_vector(17 downto 0);      -- Red LEDs

    -- RS-232 interface

    UART_TXD : out std_logic;                   -- UART transmitter
    UART_RXD : in std_logic;                    -- UART receiver

    -- IRDA interface

--  IRDA_TXD : out std_logic;                   -- IRDA Transmitter
--  IRDA_RXD : in std_logic;                    -- IRDA Receiver

    -- SDRAM

    DRAM_DQ : inout std_logic_vector(15 downto 0); -- Data Bus
    DRAM_ADDR : out std_logic_vector(11 downto 0); -- Address Bus
    DRAM_LDQM,                                  -- Low-byte Data Mask
    DRAM_UDQM,                                  -- High-byte Data Mask
    DRAM_WE_N,                                  -- Write Enable
    DRAM_CAS_N,                                 -- Column Address Strobe
    DRAM_RAS_N,                                 -- Row Address Strobe
    DRAM_CS_N,                                  -- Chip Select
    DRAM_BA_0,                                  -- Bank Address 0
    DRAM_BA_1,                                  -- Bank Address 0
    DRAM_CLK,                                   -- Clock
    DRAM_CKE : out std_logic;                   -- Clock Enable

    -- FLASH

    FL_DQ : inout std_logic_vector(7 downto 0);      -- Data bus
    FL_ADDR : out std_logic_vector(21 downto 0);  -- Address bus
    FL_WE_N,                                    -- Write Enable
    FL_RST_N,                                   -- Reset
    FL_OE_N,                                    -- Output Enable
    FL_CE_N : out std_logic;                        -- Chip Enable

    -- SRAM

    SRAM_DQ : inout std_logic_vector(15 downto 0); -- Data bus 16 Bits
    SRAM_ADDR : out std_logic_vector(17 downto 0); -- Address bus 18 Bits
    SRAM_UB_N,                                -- High-byte Data Mask
    SRAM_LB_N,                                -- Low-byte Data Mask
    SRAM_WE_N,                                -- Write Enable
    SRAM_CE_N,                                -- Chip Enable
    SRAM_OE_N : out std_logic;                     -- Output Enable

    -- USB controller

    OTG_DATA : inout std_logic_vector(15 downto 0); -- Data bus
    OTG_ADDR : out std_logic_vector(1 downto 0);    -- Address
    OTG_CS_N,                                 -- Chip Select
    OTG_RD_N,                                 -- Write
    OTG_WR_N,                                 -- Read
    OTG_RST_N,                                -- Reset
    OTG_FSPEED,                   -- USB Full Speed, 0 = Enable, Z = Disable
    OTG_LSPEED : out std_logic;     -- USB Low Speed, 0 = Enable, Z = Disable
    OTG_INT0,                                 -- Interrupt 0
    OTG_INT1,                                 -- Interrupt 1
    OTG_DREQ0,                                -- DMA Request 0
    OTG_DREQ1 : in std_logic;                      -- DMA Request 1
    OTG_DACK0_N,                              -- DMA Acknowledge 0
    OTG_DACK1_N : out std_logic;                   -- DMA Acknowledge 1

--ETHERNET INTERFACE
    ENET_DATA: inout std_logic_vector(15 downto 0);
    ENET_CMD: out std_logic;
    ENET_CS_N: out std_logic;
    ENET_WR_N: out std_logic;
```

```
ENET_RD_N: out std_logic;
ENET_RST_N: out std_logic;
ENET_INT: in std_logic;
ENET_CLK: inout std_logic:='0';


-- 16 X 2 LCD Module

LCD_ON,                    -- Power ON/OFF
LCD_BLON,                    -- Back Light ON/OFF
LCD_RW,                    -- Read/Write Select, 0 = Write, 1 = Read
LCD_EN,                    -- Enable
LCD_RS : out std_logic;     -- Command/Data Select, 0 = Command, 1 = Data
LCD_DATA : inout std_logic_vector(7 downto 0); -- Data bus 8 bits

-- SD card interface

SD_DATA: inout std_logic;                   -- SD Card Data
--SD_DAT3,                    -- SD Card Data 3
SD_CMD : out std_logic;   -- SD Card Command Signal
SD_CLK : out std_logic;     -- SD Card Clock

-- USB JTAG link

--TDI,                         -- CPLD -> FPGA (data in)
--TCK,                         -- CPLD -> FPGA (clk)
--TCS : in std_logic;         -- CPLD -> FPGA (CS)
--TDO : out std_logic;        -- FPGA -> CPLD (data out)

-- I2C bus

I2C_DATA : inout std_logic; -- I2C Data
I2C_CLK : inout std_logic;   -- I2C Clock
```

```
-- PS/2 port

PS2_DATA,                    -- Data
PS2_CLK : inout std_logic;     -- Clock

-- VGA output

VGA_CLK,                                    -- Clock
VGA_HS,                                   -- H_SYNC
VGA_VS,                                   -- V_SYNC
VGA_BLANK,                                  -- BLANK
VGA_SYNC : out std_logic;                        -- SYNC
VGA_R,                                   -- Red[9:0]
VGA_G,                                   -- Green[9:0]
VGA_B : out std_logic_vector(9 downto 0);                  -- Blue[9:0]

TD_RESET: out std_logic




-- Audio CODEC

--AUD_ADCLRCK : inout std_logic;                    -- ADC LR Clock
--AUD_ADCDAT : in std_logic;                         -- ADC Data
--AUD_DACLRCK : inout std_logic;                    -- DAC LR Clock
--AUD_DACDAT : out std_logic;                        -- DAC Data
--AUD_BCLK : inout std_logic;                       -- Bit-Stream Clock
--AUD_XCK : out std_logic;                          -- Chip Clock

-- Video Decoder

--TD_DATA : in std_logic_vector(7 downto 0);  -- Data bus 8 bits
--TD_HS,                                  -- H_SYNC
```

```
   --TD_VS : in std_logic;                    -- V_SYNC
   --TD_RESET : out std_logic;                 -- Reset

   -- General-purpose I/O

   --GPIO_0,                                  -- GPIO Connection 0
   --GPIO_1 : inout std_logic_vector(35 downto 0) -- GPIO Connection 1
   );

end packet_hard_soft;

architecture datapath of packet_hard_soft is

--signal mSEG7_HEX: std_logic_vector(31 downto 0);
signal DATA_to_OTG: std_logic_vector(15 downto 0);
signal DATA_to_SRAM: std_logic_vector(15 downto 0);

signal SCL_OE_N: std_logic;
signal SDA_OE_N: std_logic;
signal SCL: std_logic;
signal SDA: std_logic;
signal clk: std_logic:='0';
signal timer_irq:std_logic;
signal sig_ENET_DATA:std_logic_vector(15 downto 0):=x"0000";
signal CPU_RESET: std_logic;
signal check_out1:std_logic_vector(3 downto 0):="0000";
signal check_out2:std_logic_vector(3 downto 0):="0000";
signal check_out3:std_logic_vector(3 downto 0):="0000";
signal check_out4:std_logic_vector(3 downto 0):="0000";
--variable ENET_CLK: std_logic;
--always@(posedge OSC_50)           ENET_CLK=~ENET_CLK;


component reset_delay is
```

```
port (
        iCLK: in std_logic;

        signal oRESET: out  std_logic;
        signal Cont: buffer unsigned(15 downto 0):="0000000000000000"
        );
end component;
begin
TD_RESET <= '1';
LCD_ON <= '1';
LCD_BLON <= '1';
FL_RST_N <= '1';
FL_ADDR(21) <= '0';
FL_ADDR(20) <= '0';

process(CLOCK_50)
begin
if rising_edge (CLOCK_50) then
ENET_CLK <= NOT ENET_CLK;
clk <= NOT clk;
end     if;
end process;




U2: reset_delay port map(
oRESET => CPU_RESET,
iCLK => CLOCK_50
);

U3: entity work.packet_soft_hard port map(
        --iCLK_to_the_MUX => CLOCK_50,
            clk => CLOCK_50,
    reset_n => CPU_RESET,

-- the_DM9000A

                ENET_CMD_from_the_MUX => ENET_CMD,
                ENET_CS_N_from_the_MUX => ENET_CS_N ,
                ENET_DATA_to_and_from_the_MUX => ENET_DATA,
                ENET_INT_to_the_MUX =>ENET_INT,
                ENET_RD_N_from_the_MUX =>ENET_RD_N,
                ENET_RST_N_from_the_MUX => ENET_RST_N,
                ENET_WR_N_from_the_MUX => ENET_WR_N,
                                STATE_OUT_from_the_MUX => check_out1(3 downto 0),
                                dbug_ENET_DATA_from_the_MUX => sig_ENET_DATA,
              dbug_io_rw_cmd_from_the_MUX => check_out2(1 DOWNTO 0),
              dbug_req_update_from_the_MUX => check_out3(0),
              dbug_state_out_iter_from_the_MUX => check_out4 (2 DOWNTO 0),
                                --  STATE_OUT_from_the_MUX => check_out,
                                --iCMD_to_the_MUX=>iCMD,--KEY(1 downto 0),
                                --iCMD_chk_from_the_MUX=> HEX0(1 downto 0),

             -- the_SRAM
                SRAM_ADDR_from_the_SRAM => SRAM_ADDR,
                SRAM_CE_N_from_the_SRAM =>SRAM_CE_N ,
                SRAM_DQ_to_and_from_the_SRAM =>SRAM_DQ,
                SRAM_LB_N_from_the_SRAM => SRAM_LB_N,
                SRAM_OE_N_from_the_SRAM => SRAM_OE_N,
                SRAM_UB_N_from_the_SRAM => SRAM_UB_N,
                SRAM_WE_N_from_the_SRAM =>SRAM_WE_N


);
U4: entity work.hex7seg port map(
     input=>check_out1,
    output=>HEX0
);

U5: entity work.hex7seg port map(
    input=>check_out2,
    output=>HEX1
);
U6: entity work.hex7seg port map(
    input=>check_out3,
    output=>HEX2
);
U7: entity work.hex7seg port map(
    input=>check_out4,
    output=>HEX3
);

U8: entity work.hex7seg port map(
    input=>sig_ENET_DATA(3 downto 0),
    output=>HEX4
);

U9: entity work.hex7seg port map(
   input=>sig_ENET_DATA(7 downto 4),
   output=>HEX5
);



--HEX0(6)<='1';
--HEX1(6 downto 4)<="111";

--Hex0<="1111111";
--Hex1<="1111111";
--Hex2<="1111111";
--Hex3<="1111111";
```

```
--Hex4<="1111111";
--Hex5<="1111111";
Hex6<="1111111";
Hex7<="1111111";
end datapath;
```

Main.c (Software file)

```
#include <io.h>
#include <system.h>
#include <stdio.h>
#include <sys/alt_irq.h>  // the irq functions
#include <alt_types.h>
#include "basic_io.h"
#include "DM9000A.h"
#include <alt_types.h>
//#include "LCD.h"
#define MAX_MSG_LENGTH 128

#define IOWR_packet_data(base, offset, data) \
  IOWR_16DIRECT(base, (offset) * 2, data)

#define price 0
#define name 1
#define quant 2
#define buy_sell 3



// Ethernet (source) MAC address.  Choose the last three bytes yourself
unsigned char mac_address[6] = { 0x01, 0x60, 0x6E, 0x11, 0x02, 0x0F  };

unsigned int interrupt_number;

unsigned int receive_buffer_length;
unsigned char receive_buffer[1600];

#define UDP_PACKET_PAYLOAD_OFFSET 42
#define UDP_PACKET_LENGTH_OFFSET 38

#define IP_PACKET_LENGTH_OFFSET 16
```

```
#define UDP_PACKET_PAYLOAD (transmit_buffer + UDP_PACKET_PAYLOAD_OFFSET)

unsigned char transmit_buffer[] = {
 // Ethernet MAC header
 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, // Destination MAC address
 0x01, 0x60, 0x6E, 0x11, 0x02, 0x0F, // Source MAC address
 0x08, 0x00,                         // Packet Type: 0x800 = IP

 // IP Header
 0x45,              // version (IPv4), header length = 20 bytes
 0x00,              // differentiated services field
 0x00,0x9C,          // total length: 20 bytes for IP header +
                // 8 bytes for UDP header + 128 bytes for payload (change according to payload size)
                // 16,17
 0x3d, 0x35,         // packet ID (increment it!!)
 0x00,              // flags
 0x00,              // fragment offset
 0x80,              // time-to-live
 0x11,              // protocol: 11 = UDP
 0x00,0x00,          // header checksum: *incorrect*(24,25 index)
 0xc0,0xa8,0x01,0x01, // source IP address
 0xc0,0xa8,0x01,0xff, // destination IP address

 // UDP Header
 0x67,0xd9, // source port port (26585: garbage)
 0x27,0x2b, // destination port (10027: garbage)
 0x00,0x88, // length (136: 8 for UDP header + 128 for data)
 0x00,0x00 // checksum: 0 = none

};


unsigned char market_packet[] = {
 0x74, 0x65, 0x73, 0x74, 0x20, 0x6d, 0x73, 0x67,
 0x74, 0x65, 0x73, 0x74, 0x20, 0x6d, 0x73, 0x67,
 0x74, 0x65, 0x73, 0x74, 0x20, 0x6d, 0x73, 0x67,
 0x74, 0x65, 0x73, 0x74, 0x20, 0x6d, 0x73, 0x67,
 0x74, 0x65, 0x73, 0x74, 0x20, 0x6d, 0x73, 0x67,
 0x74, 0x65, 0x73, 0x74, 0x20, 0x6d, 0x73, 0x67,
 0x74, 0x65, 0x73, 0x74, 0x20, 0x6d, 0x73, 0x67,
 0x74, 0x65, 0x73, 0x74, 0x20, 0x6d, 0x73, 0x67,
 0x74, 0x65, 0x73, 0x74, 0x20, 0x6d, 0x73, 0x67,
 0x74, 0x65, 0x73, 0x74, 0x20, 0x6d, 0x73, 0x67,
 0x74, 0x65, 0x73, 0x74, 0x20, 0x6d, 0x73, 0x67,
 0x74, 0x65, 0x73, 0x74, 0x20, 0x6d, 0x73, 0x67,
 0x74, 0x65, 0x73, 0x74, 0x20, 0x6d, 0x73, 0x67,
 0x74, 0x65, 0x73, 0x74, 0x20, 0x6d, 0x73, 0x67,
 0x74, 0x65, 0x73, 0x74, 0x20, 0x6d, 0x73, 0x67,
 0x74, 0x65, 0x73, 0x74, 0x20, 0x6d, 0x73, 0x67
};


int market_packet_length = 128;
unsigned char *p = market_packet;

static void ethernet_interrupt_handler() {
 unsigned int receive_status;
 int i;
     printf("\n\n\n\n\n\n\nPACKET RECEIVED");
 receive_status = ReceivePacket(receive_buffer, &receive_buffer_length);

 if (receive_status == DMFE_SUCCESS) {

#if 1
    printf("\n\nReceived Packet Length = %d", receive_buffer_length);
    for(i=0;i<receive_buffer_length;i++) {
     if (i%8==0) printf("\n");
     printf("0x%.2X,", receive_buffer[i]);
    }
    printf("\n");
#endif



    if (receive_buffer_length >= 14) {
     //  A real Ethernet packet
     if (receive_buffer[12] == 8 && receive_buffer[13] == 0 &&
     receive_buffer_length >= 34) {
    // An IP packet
    if (receive_buffer[23] == 0x11) {
     // A UDP packet
     if (receive_buffer_length >= UDP_PACKET_PAYLOAD_OFFSET) {
       printf("Received: %s\n",
          receive_buffer + UDP_PACKET_PAYLOAD_OFFSET);

     }
    } else {
     printf("Received non-UDP packet\n");
    }
     } else {
    printf("Received non-IP packet\n");
     }
    } else {
     printf("Malformed Ethernet packet\n");
    }

 } else {
    printf("Error receiving packet\n");
 }
    interrupt_number++;
 /* Display the number of interrupts on the LEDs */
 /* Clear the DM9000A ISR: PRS, PTS, ROS, ROOS 4 bits, by RW/C1 */
 dm9000a_iow(ISR, 0x3F);   //interrupt status register

 /* Re-enable DM9000A interrupts */
 dm9000a_iow(IMR, INTR_set);   //interrupt mask register
}

int main()
{

 // int price_1 = 32767, price_2 = 32767, price_3 = 0;
   int price_1 = 32767, price_2 =32767, price_3 = 3;
 //int price_11 = 32767, price_22 = 32767, price_33 = 0;
 int name_1 = 32765, name_2 = 32767, name_3 =3;
 int quant_1 = 32767, quant_2 =32767, quant_3=3;
 int buy_sell_1 = 32767;
 int j;
 int curMsgChar = 0;
 printf("nios up");
 unsigned int packet_length;

    /*while(p != NULL)
    {
       market_packet_length++;
       p++;
    }
    */

 // Initalize the DM9000 and the Ethernet interrupt handler
 unsigned int status=DM9000_init(mac_address);

 if (status==DMFE_SUCCESS){printf("DM9000 initialized succesfully\n");
 //write to DM9000A custom hardware, Init done !!
    IOWR(MUX_BASE, Initialized, 5);
  }
  else{printf("DM9000 initialized FAILED!!!\n");}
  interrupt_number = 0;
  //alt_irq_register(DM9000A_IRQ, NULL, (void*)ethernet_interrupt_handler);

 printf("4840 Lab 2 started\n");
  // Initialize the keyboard

  printf("Ready to send messages\n");

for(j = 0; j<100000; j++);
for(j = 0; j<1000000; j++);{
printf("Calling IOWR1");
IOWR_packet_data(MUX_BASE, price, price_1) ;
IOWR_packet_data(MUX_BASE, price, price_2) ;
IOWR_packet_data(MUX_BASE, price, price_3) ;

}

   //for(;;){
  // Clear the payload
  for (curMsgChar=MAX_MSG_LENGTH-1; curMsgChar>0; curMsgChar--) {
    UDP_PACKET_PAYLOAD[curMsgChar] = 0;
  }

  for (curMsgChar = 0; curMsgChar < market_packet_length; curMsgChar++) {
    UDP_PACKET_PAYLOAD[curMsgChar] = market_packet[curMsgChar];
  }

  printf("\nCurMsgChar length after insert = %d",curMsgChar);



      UDP_PACKET_PAYLOAD[curMsgChar++] = 0; // Terminate the string
      packet_length = UDP_PACKET_PAYLOAD_OFFSET + curMsgChar;
      //do 0 padding to make packet length min 64 bytes
      while(packet_length < 64){
                  UDP_PACKET_PAYLOAD[curMsgChar++] = 0;
                  packet_length++;
                  }

      transmit_buffer[UDP_PACKET_LENGTH_OFFSET] = (packet_length - 34) >> 8;

      transmit_buffer[UDP_PACKET_LENGTH_OFFSET + 1] = (packet_length - 34) & 0xff;

      //increment packet ID 19, 20 posn from strt of packet header
      transmit_buffer[19]+=0x01;

      //39 and 40 contain UDP packet length
      transmit_buffer[IP_PACKET_LENGTH_OFFSET] = (packet_length - 14) >> 8;

      transmit_buffer[IP_PACKET_LENGTH_OFFSET + 1] = (packet_length - 14) & 0xff;

      printf("\n packet length set to: %d",transmit_buffer[39]);

      //setting checksum//
      //take 2s compliment den add

transmit_buffer[25]=transmit_buffer[15]+transmit_buffer[17]+transmit_buffer[19]+transmit_buffer[
21]+transmit_buffer[23]+transmit_buffer[27]+transmit_buffer[29]+transmit_buffer[31]+transmit_bu
ffer[33];

transmit_buffer[24]=transmit_buffer[14]+transmit_buffer[16]+transmit_buffer[18]+transmit_buffer[
20]+transmit_buffer[22]+transmit_buffer[26]+transmit_buffer[28]+transmit_buffer[30]+transmit_bu
ffer[32];
      printf("\n before inversion %d %d",transmit_buffer[24],transmit_buffer[25]);
      //ones compliment or inversion
      transmit_buffer[25]=~transmit_buffer[25];
```

```
      transmit_buffer[24]=~transmit_buffer[24];


      if (TransmitPacket(transmit_buffer, UDP_PACKET_PAYLOAD_OFFSET + curMsgChar + 1)
== DMFE_SUCCESS) {
        printf("\nMessage sent successfully\n");
        //enter packet send

         printf("\n\nSending Packet Length = %d", packet_length);
         int i;
              for(i=0;i<packet_length;i++) {
                     if (i%8==0)
                     printf("\n");
                     printf("0x%.2X,", transmit_buffer[i]);
                  }
           printf("\n");



      } else
        printf("\nMessage sending failed\n");

        // reset data
     for (curMsgChar=MAX_MSG_LENGTH-1; curMsgChar>0; curMsgChar--) {
     UDP_PACKET_PAYLOAD[curMsgChar] = 0;
        printf("Msg to Send: ");

      int delay;
     for(delay = 0;delay < 8000000;delay++)
        {
        }

  //}
  return 0;
    }
}
```