Tampere University of Technology

Faculty of Computing and Electrical Engineering

Department of Computer Systems

FPGA—PC Ethernet Communication IPs Reference Manual

Author:
Antti Alhonen

Updated:
November 8, 2011

Contents

1 REVISION HISTORY

Table 1

Revision	Author	Date	Description
1.0	Antti Alhonen	28.10.2011	First published version

2 DOCUMENT OVERVIEW

2.1 SCOPE

This documentation describes the usage of DCS/TUT made IPs to allow simple communication between a computer and various FPGA development boards. We give an overview to all parts needed for bi-directional communication over UDP/IP protocol at 100 Mbit/s with one computer.

The main scope is on the simplicity of usage; therefore, some features such as reliable operation in a network consisting of numerous devices and computers are not guaranteed to work.

2.2 AUDIENCE

- SoC developers
- Kactus 2 design tool users
- Users of development boards that include an FPGA and Davicom DM9000A (or possibly SMSC LAN91C111) Ethernet MAC/PHY

2.3 RELATED DOCUMENTATION

Table 2

Document	Description
UDP/IP with VHDL	Implementation specification of the UDP/IP packetizer used in Trace Monitor
DM9kA controller	Implementation specification of the DM9000A ethernet MAC/PHY interface

3 INTRODUCTION

3.1 CONTACT INFORMATION

If you face any problems using the IPs offered, please do not hesitate to ask for help or to give suggestions. You can contact Antti Alhonen (*antti.alhonen@tut.fi*).

3.2 INTRODUCTION

It is very usual to use FPGA's for non-self-contained systems that need to communicate with computers to transfer large amounts of data. Standard UART is very simple for transferring small commands and debug information, but not fast enough to transfer, for example, video or audio data.

Multiple solutions exist, such as USB, PCI, PCI Express, Firewire and Ethernet connection. Ethernet connection has some benefits such as simplicity and existing communication routines in all major operating systems without a need to write device drivers. The biggest drawback is the lack of guarantee for transfers. However, this problem can be managed.

Usually, a soft-core processor is synthesized on FPGA; then, a *software* controller and drivers are provided for the external ethernet chip. In this case, it is possible to implement complex protocols such as TCP/IP. The drawback is the area overhead of the processor, extra latency because of the software, possible bandwidth decrease depending on the processor and the need of embedded processor and software. Sometimes, a more simple HW approach is needed or preferred.

3.3 BRIEF DESCRIPTION

This solution implements UDP/IP protocol; a simple, packet-switched, low-latency, low-overhead communication between two devices: an FPGA board and a PC.

The HW interface consists of two FIFO interfaces (send and receive) and a few intuitive control signals and busses such as "start new transfer", "destination IP address", "new packet received" etc. These separate control signals allow very simple usage from HW (VHDL, Verilog etc.) applications.

The implementation includes UDP/IP packetizer and controllers for external Ethernet MAC/PHY chips – currently, two different brands are supported with some restrictions. Our implementation exploits the FIFO buffers provided by these external chips, thus the on-chip FPGA memory requirement is zero.

Ethernet controllers currently supported by us include:

- Davicom DM9000A
- SMSC LAN91C111 (partially)

A GMII output will be offered in the future to enable Gigabit Ethernet operation.

It is **very strongly recommended** that you connect these controllers to a separate network interface card in your PC directly, i.e. without any switch, let alone a router. Both of the network chips mentioned support autonegotiation that should be capable of working with normal cable in this situation, but if the link does not go up, you can try a cross-over cable (cable with RX and TX swapped to allow direct connection).

3.4 Designs provided

Table 3

Design	Files	Description
UDP/IP Packetizer	udp_ip.vhd (toplevel) udp_ip_pkg.vhd (config package) udp.vhd udp_arp_data_mux.vhd ip_checksum.vhd arp3.vhd (optional) arpsnd.vhd (optional)	Offers a simple interface for user application logic to perform TX and RX operations over Ethernet with UDP/IP protocol. The MAC/IP addresses for the FPGA board are set in udp_ip_pkg.vhd. This block is connected to either one of the Ethernet Chip Controllers provided. Combined toplevels to provide this connection are provided, too.
DM9000A or LAN91C111 Controller	DM9kA_controller.vhd (toplevel) DM9kA_ctrl_pkg.vhd (config package) DM9kA_comm_module.vhd DM9kA_init_module.vhd DM9kA_interrupt_handler.vhd DM9kA_read_module.vhd DM9kA_send_module.vhd (substitute lan91c111 for DM9kA for LAN91C111 Controller)	Connected between UDP/IP block and IO pins of the FPGA, these control blocks first initialize the external Ethernet MAC/PHY in question and then control the writes and reads to/from the TX/RX FIFOs, commands etc. The entity-level construction is identical for the two supported controllers but the implementations differ due to very different interfaces of these chips. You can set the FPGA board MAC address in _ctrl_pkg.vhd.
Simple UDP Flood Example	simple_udp_flood_example.vhd	A minimum example to show how TX operations are performed. Creates continuous traffic with adjustable packet size to an IP/MAC address defined as generic parameters. First four bytes include an increasing count to detect packet loss. Connects to the UDP/IP block. A Kactus 2 example file is provided. Also includes the interface for RX operations; it reads out and ignores all incoming packets. You can leave it disconnected if you want to connect the Receiver Example at the same time.
Simple UDP Receiver Example	simple_udp_receiver_example.vhd	A minimum example to show how incoming packets can be handled. It reads out the incoming packet from the RX FIFO and changes a LED status every time a packet is received. Connects to the UDP/IP block. A Kactus 2 example file is provided. Includes also the interface for TX operations; it never sends anything. You can leave it disconnected if you want to connect the Flood Example at the same time.

3.5 Clock and reset

Ethernet controller block and UDP/IP block both run on a synchronous 25 MHz clock. You may need to instantiate an FPGA vendor specific PLL to generate this clock. Naturally, you can use an integer-multiple synchronous higher clock for your own application as long as you make sure the control signals connected to UDP/IP are synchronous and stable to the 25 MHz clock.

Our DM9000A controller outputs the incoming 25 MHz clock directly to an output IO pin that is

connected to the clock input of DM9000A, hence, the communication with the chip is synchronous. DM9000A is connected like this in Altera DE2 development board.

Our LAN91C111 controller, on the other hand, communicates asynchrously with the chip. Hence, the chip needs its own clock source. LAN91C111 is connected like this in Altera Stratix II S180 development board.

An asynchronous active low reset signal is used throughout these designs. After the reset is released (to the high level), the external ethernet chip is reset and the autonegotiation process is automatically started. This will take around five to ten seconds, after which the link_up signal is provided high, new tx operations accepted and incoming packets handled. Note that LAN91C111 uses a "soft" reset in the initialization process and therefore the link does not go down when activating reset but when it is released from the reset.

3.6 DM9000A

Our implementation of DM9000A controller is complete for basic TX and RX operations and comparably well-tested.

3.7 LAN91C111

Due to the very high complexity of usage of LAN91C111 controller and a number of critical HW bugs in the chip, we have not been able to demonstrate a reliably working connection with this chip. We are not sure if we want to work with this chip anymore, so we cannot guarantee any updates. Hence, we would encourage not to use this very peculiar and obsolete chip unless absolutely necessary.

Currently, simple TX-only and RX-only operations are working in test environments, but using ARP causing simultaneous TX and RX operations causes a freezing of the chip hard to analyze. The chip simply stops giving receive interrupts.

Furthermore, due to a critical HW bug in LAN91C111, sending smaller than 66-byte long packets does not work as intended. As a workaround, our controller pads the packets with zeros but cannot provide correct packet length field.

3.8 IP-XACT files provided

We have created a set of IP-XACT descriptions in Kactus 2 design software. It is possible to construct a complete working example using these. Two examples are provided; a TX (send) example and an RX (receive) example.

You need to instantiate one (or both) of the two examples and one of the UDP/IP/Eth CTRL combinations, depending on your external MAC/PHY chip brand.

Ready-to-use examples are provided. Please note that as described in the previous subsection, the LAN91C111 examples may or may not work depending on the network setup.

4 Usage

4.1 Sending a packet (TX)

First, make sure the link is up, i.e., link_up is high. Usually, it takes a few seconds from a reset to link go up, but this can depend on the PC, too. Status LED at the RJ-45 connector lits a few seconds before the link_up signal goes high.

When you have the first word (16 bits) of your payload data ready, do the following:

1. Output the amount of your actual payload data **in bytes** in tx_len
2. Output the destination IP address (the PC) in target_addr, most significant byte first, e.g. x"0A000001" in VHDL for 10.0.0.1.
3. **If** the ARP functionality is not enabled, output the destination MAC address (the PC) in no_arp_target_mac.
4. Output the IP port number where you want the packet to be sent in target_port. (Remember that you need to open this port when designing the PC software.)
5. Output the desired IP port number of the FPGA board in source_port. Usually this does not matter but you can check it in your PC software.
6. Output the first data word in tx_data. This will be in **MSbyte last** endianness; thus, the first byte on the line is located in tx_data(7 downto 0).
7. Output tx_data_valid = '1'. If you are using a FIFO buffer, you can use not empty signal for this one.
8. Raise the new_tx signal—this, alongwith tx_data_valid, causes the UDP/IP packetizer to start reading the data. [1]
9. tx_re works as an acknowledgement signal; supply the next data word immediately, or, if not possible with the next clock cycle, lower the data_valid for the next cycle.
10. After the amount of data indicated by tx_len (i.e., ceiling of tx_len/2) has been read, the chip starts sending **automatically**. Just make sure you feed the amount of data you promised in step 1.
11. You can now start the next packet almost immediately. The controllers have FIFO buffer memory area for the next packet while it is sending the previous one.

You can set all of these signals simultaneously. However, if you don't, then set the new_tx last.

If you communicate with only one PC and want to hard-code the addresses (necessiating a re-synthesis if the addresses or ports have to be changed), it's easiest to hard-wire these ports when instantiating the component.

Please see the provided example design, simple_udp_flood_example.vhd.

[1] If the ARP functionality is enabled and the FPGA does not yet have the PC MAC address (i.e., this is the first TX operation after power-up, programming or reset), the ARP query will be sent before any payload data is read. It may take a variable time for the PC to answer. Therefore, for time-critical applications, or just to keep things simple, we recommend to disable the ARP if it's not needed. The downside to this is that you need to supply the PC MAC address to the FPGA and if you don't have a separate input system, you need to resynthesize every time if you use multiple PCs.

4.2 Receiving a packet (RX)

After the chip receives a packet from the network with a matching MAC address and a matching IP address (set your FPGA's IP address in udp_ip_pkg.vhd), having correct UDP protocol headers, your application will be notified by rising new_rx. If the IP address or protocol is wrong, the packet will be ignored without a notification.

When the new_rx is up, you can read out the packet as follows:

1. If you need the information, you can read source_addr (source IP address, i.e. the PC), source_port (i.e. the output IP port on the PC, usually not interesting), dest_port (the "FPGA port" where it was "sent to", may be interesting to check to ignore (read out) accidental interactions from PC.)
2. Copy rx_len to a register; you will need to count the bytes you are going to read.
3. If you decide to ignore the packet, still read it out like any data.
4. When rx_data_valid is high and you did not read on the previous clock cycle, you can read out the next data word from rx_data and acknowledge it by setting rx_re = '1' for one cycle.
5. After you have asserted rx_re high for ceil(rx_len/2) times correctly, you have read the whole packet. You don't need to do anything more.

Please note that if you do not read out the whole packet, no new packets are received after a few packets (RX FIFO gets full).

Please see the provided example design, simple_udp_receiver_example.vhd.

5 Application development tips

5.1 MANAGING PACKET LOSS IN HIGH-DATA RATE APPLICATIONS

UDP/IP is a "send-and-forget" protocol. The HW reliability is, however, very high if not practically perfect in a direct connection.

However, due to the nature of PC's and especially their operating systems, a small percentage of packets may be dropped in high-data rate applications; it depends on your needs whether this is a problem or not.

First, to identify the possible problem, it is recommended that your application inserts a packet number to the start of every packet payload, increasing by one after every packet. This way, dropped packets can be counted. You can design your application so that the PC software part can ask to resend a missed packet, however, if the data cannot be regenerated in-situ, you will need a relatively large buffer memory near the FPGA to do this.

Luckily, in practice, it is possible to reduce the number of dropped packets to zero even in long runs with data rates close to maximum. Naturally, this cannot be formally guaranteed but it is a very attractive option due to the simplicity.

There are practically two reasons for dropping a packet:

- (1) Your PC software does not have enough time to read out packets from the the RX buffer of the network socket in the OS network implementation; after the buffer is full, packets are thrown away.
- (2) The operating system's network kernel or device driver does not have enough time to read out the packets from the RX FIFO of the Ethernet card and place them to the buffer mentioned in (1).

To solve problem 1, increase the RX buffer size. You will need OS-specific API calls. If you don't want to do it in your software, you can just increase the default RX buffer of your OS for all programs that do not use their own setting. For example, in MS Windows XP, open regedit, locate HKEY\LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services\AFD\Parameters and add a new DWORD value named DefaultReceiveWindow with a desired RX buffer size in bytes. I have used values in range of about 300 000 with great results (zero drops at 80 Mbps of payload). The default is very small, only 8192 bytes. Note that this will be used for all programs as a default unless the programs define a different buffer size and thus the memory usage may increase. For Linux and other systems, you have to Google around.

To solve problem 2, buy a network adapter with longer RX buffer. Simply put, the older 100 Mbps cards have very short buffers, regardless of the manufacturer. On the other hand, Gigabit cards, even the cheapest ones, have longer buffers. I have about 10-100 ppm of packet loss with a 3Com 100 Mbps card (with 8 kbyte RX FIFO) but zero packet loss with a cheap Realtek Gigabit card, at 80 Mbps, on Windows XP, with increased OS RX buffer size.

6 KNOWN ISSUES

- Lack of workarounds for some LAN91C111 problems; see Section 3.7.