Hardware Design and Implementation of IP-over-1394 Protocol Stack and Its Evaluation

Mohd Yusairi Bin Abu Hassan† and Kôki Abe†

This paper describes a hardware design of a subset of the Internet protocol IP over IEEE1394 interface (IP over 1394) and its implementation on an FPGA. It evaluates the design by counting the hardware costs required for the implementation. It is intended for use in such applications as home networks, where transferring and processing digital AV data at low costs and by low power consumption is desirable. Using a system clock of 49.152MHz, we verified that packets sent from an application on top of the protocol stack were correctly received by the other protocol stack via IEEE1394 port at a transfer rate of 400 Mbps. We also verified communication behaviors of the design with an isochronous resource manager to reserve a channel prior to data transmissions. The hardware cost of the IP layer is 84% of the link layer's cost, meaning that the IP-over-1394 protocol stack can be realized by hardware by spending an extra cost less than that of the basic hardware which already exists.

1. Introduction

It is now popular to connect multiple PCs and peripherals by means of networks at home. Demands on dealing with audio/video (AV) data through the network have also been increased. Networking AV devices with the PCs requires processing, transmitting, and receiving the digital AV data at high speeds. Furthermore for home use, it is important to realize the network at low costs. The less silicon usage results in the less power consumption.

Due to its distinct features such as high speed data transfers at low cost, plug-and-play functions, etc., IEEE1394 interface[1], also known as FireWire, has been considered to be a good candidate of a physical link for home networks. A standard protocol for the Internet Protocol over IEEE1394 interface (IP over 1394) has been specified as an RFC[2], and some implementations of the protocol by software exist[3),4].

Software implementations, however, require a platform of PC with high performance to process bulk of data flowing through the interface at high speed. If a dedicated hardware processing the protocol is realized at low cost, it will alleviate the loads from the platform, enabling to embed low cost processors with low power consumption in networked devices. We can integrate the dedicated hardware into a system on a chip by spending an extra amount of silicon resource. Thus there is a chance of realizing a cost-effective high-speed home networks with less power consumption by implementing IP-over-1394 protocol stack in hardware, which depends on the cost required for the stack.

A hardware implementation of IPv4 has been reported[5], and a reduced set of IPv6 has been realized also in hardware[6]. But those works are all based on Ethernet.

This paper examines the feasibility of implementating IPv4-over-1394 protocol stack from transport layer downto data-link layer all in hardware and evaluates its hardware costs. We adopt an FPGA as the implementation device and measure the costs by counting the amount of logic elements in the FPGA required for the implementation.

The FPGA is useful in that it facilitates prototyping the design, verifying the operations, and evaluating relative costs among protocol layers. The link layer at the bottom is normally implemented in hardware as 1394 controller, whose hardware cost is a good measure of dispensable cost value. The FPGA realization also gives an lower bound of performance of hardware implementation: ASIC realizations will yield higher operation speeds and lower gate counts.

In the following we will first overview the IEEE1394 features. Then we will describe operations of IP-over-1394, design of hardware modules, implementation results, experiments for the design verification, and evaluation of the results followed by conclusions.

† Department of Computer Science, the University of Electro-Communications

2. Features of IEEE1394

The IEEE1394 supports high data transfers with 100Mbps to 400Mbps. (IEEE1394b enables transfers at a rate of 3.2Gbps[7].) The speeds are enough for transferring large and fast packets of such as audio/video data. With the feature of isochronous transfer ability, it supports stream data transfers with real time gurantee at constant transfer rates.

The IEEE 1394 is a bus architecture and not I/O interface like Ethernet. It means that IEEE1394 is a collision free medium managed by a full arbiter. The IEEE1394 arbiter realizes a fairness protocol, which prevents the bus from being monopolized by the fastest node. Therefore, all nodes have equal opportunity to send.

With plug-and-play functions, IEEE1394 supports automatic configuration which allows a new device to be attached or removed from the bus without needs to set or reset the node ID and addresses, a feature different from USB or Ethernet.

3. Operations of IEEE1394

In this section we overview the transactions handled by of IEEE1394 and the operations performed by the link layer (LLC) and physical layer (PHY) for executing the transactions.

3.1 Transactions

For transmitting or receiving IEEE1394 packets, two types of data transfer services are supported: isochronous and asynchronous data transfers. A set of transactions are provided for the servises. Most transactions are classified into request and response types. A node makes a request to the other node and will receive a response from the node. When recieving a packet, the receiver node will send an acknowledge packet to tell the sender that the packet has been received with or without errors.

3.1.1 Isochronous Data Transfer

This type of transaction transfers data via a channel with a reserved bus bandwidth. Neither acknowledge nor response packets are necessary for this type of transaction.

3.1.2 Asynchronous Data Transfer

An asynchronous transaction called asynchronous stream packet transfers data via a channel and has the same packet format as the isochoronous transaction except that it does not require bus bandwith. The asynchrounous stream packet also known as Global Asynchronous Stream Packet (GASP)[8] is used in IP-over-1394 data transfer.

3.1.3 Other Transactions

Since IP over 1394 uses the asynchronous stream packet format, a channel must be reserved in advanced. Allocating channels to nodes requesting data transfers must be managed by a central node in an IEEE1394 network. The function is called isochronous resource manager (IRM) and the node equipped with the function is called IRM node.

To reserve a channel the following transactions are used:

- Cycle Start
  A packet called 'cycle start' is broadcast by a node taking a role of IRM. From the node ID specified in the packet the IRM node is known.
- Read Request/Response Data Quadlet
  These transactions are used to obtain available channel numbers within an address location in the IRM node. 'Read response data quadlet' is a response to 'read request data quadlet.'
- Lock Request/Response
  In order to reserve a channel, the node must gain a preemptive access to the IRM node. These transactions provide the mechanism to enable such an access.

3.2 Operations by LLC and PHY

The IEEE1394 interface consists of link and physical layers which are usually called LLC and PHY, respectively. Fig.1 shows the interface between PHY and LLC.

Two terminals Ctl0 and Ctl1 forming a bidirectional control bus control the flows of data between PHY and LLC. A bidirectional control bus D[7:0] transfers data. IEEE1394 supports S100 (100Mbps) using D[1:0], S200 (200Mbps) using D[3:0], and S400 (400Mbps) using D[7:0] terminals. The Lreq terminal controlled by LLC sends serial service requests to PHY for taking the access of the serial-bus for packet transmission.

3.2.1 Transmit Operation

Fig.2 shows the timing required for transmit operation. In the figure 'PHY' and 'LLC' indicate the periods when PHY and LLC control the terminals, respectively. To send a packet

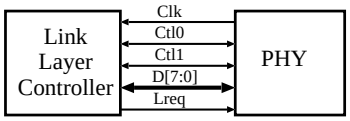

Fig. 1 PHY-LLC interface

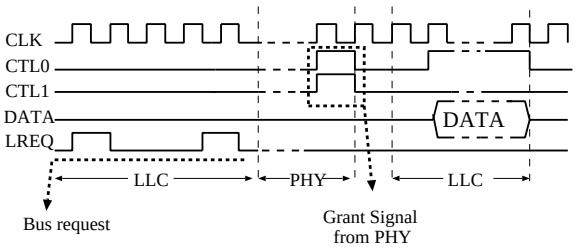

Fig. 2 Transmit operation.

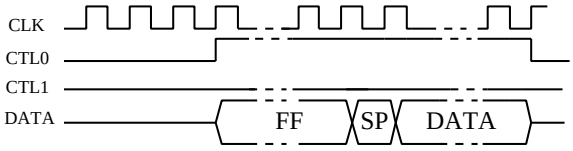

Fig. 3 Receive operation.

data, a 'Bus request' must be performed by LLC to PHY through the Lreq terminal by a request with high priority specifying a transfer speed for transmission. After LLC sends the bus request, it must wait for a grant by watching both Ctl0 and Ctl1 going to high before sending the packet using D[7:0] terminals. While transmiting the packet, Ctl0 and Ctl1 must be set high and low, respectively.

3.2.2 Receive Operation

Fig.3 shows the timing required for receiving operation. When receiving packets, all bidirectional terminals are controlled by PHY. Incoming packets should start with 0xFFs followed by speed code (SP) and data. The value of SP and the number of data terminals used depend on the packet transfer rate.

4. IP-over-1394 Protocol

Networking protocols are normally structured in a layered stack[9]. The protocol stack we are going to implement is shown in Fig.4 together with data encapsulation and decapsulation. As shown in the figure, the stack includes UDP (user datagram protocol) at the transport layer, IP (Internet Protocol) at network layer, and IEEE1394 at data-link layer; the latter two constitute the IP-over-IEEE1394. The current version of the implementation does not include some protocols such as ICMP, ARP and RARP.

4.1 Transport Layer

For the transport layer UDP is employed instead of reliable protocol such as TCP. Althouth UDP provides no reliablity, it requires minimum resources and overheads for realizing the transport layer.

A kind of reliable data transmission is guaranteed by the mechanism of IEEE1394 link layer, where a channel between nodes is reserved before transmission. Retry protocols are also provided within 1394 specifications[2]. For more sophisticated reliabilities or flow control necessary for each application, we rely on functions to be supplied by the application itself.

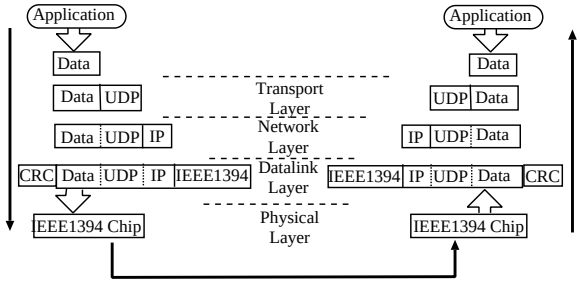

Fig. 4 UDP/IP protocol suite.

4.2 Network Layer

The network layer protocol is used to set an IP address where the packet must be transmitted for all types of network. We adopt TinyIP[10], a reduced set of IP version 4, for the network layer. TinyIP is suitable for our purpse in that it provides an end host with IP functions required for home networking with minimum costs.

4.3 Data Link Layer

We use the IEEE1394 for this layer. It is reasonable to assume a node taking a role of a gateway to the Internet in an IP-over-1394 home network. The node would usually be a PC. We rely on such a PC for the IRM function, and do not include the function into the current implementation of the IP-over-1394 module.

When using IEEE1394 link header for sending IP datagrams, it is specified to transmit them via asynchronous stream packets. The asynchronous stream packets must conform to the GASP format. IP datagrams are prefixed by an encapsulation header containing type field identifying datagrams IPv4, ARP, etc.

5. Design of IP-over-1394 Module

A hardware module realizing the function of the protocol suite listed in Tab.1 has been designed. We call the module "fireIP."

The block diagram of `fireIP` is shown in

Table 1 Protocols constituting the `fireIP`.

Layer	Protocol
Transport	UDP
Network	TinyIP
Link	IEEE1394 GASP

Fig.5. It consists of `fire_trans` and `fire_rec` for transmit and receive operations, respectively, and `control` for establishing connection between nodes.

The module `fire_trans` for transmitting data consists of `net_trans` and `link_trans`. The module `net_trans` processes the transport and network layer protocols for data to be transmitted and hands the results to the module `link_trans` for the link layer protocol. The module `fire_rec` for receiving data consists of `net_rec` for the transport and network layers and `link_rec` for the link layer.

5.1 LLC - PHY interface

First we describe some modules controlling the interface between the link and PHY layers.

- `fifo`: This module is used as a buffer to retain data in the module before tranferring them to other modules.
- `crc`: This module calculates the CRC (Cyclic Redundant Check) value while transmitting and receiving packets. In receiving, the calculated value is compared with the received CRC. In transmitting, the calculated value is attached to the packet.
- `lreq`: This module issues a bus request through the Lreq terminal to get a permission for using the bus from PHY.
- `trans`: This module controls the use of data_out[7:0] according to the speed request. It also controls the transmission timing. The `trans` module starts transmiting link packets after receiving the grant signal from PHY. When transmiting data to PHY, `trans` asserts ctl0_out terminal to request PHY that the data be transmitted.
- `rec`: The PHY hands a received packet to the module `rec` by asserting ctl0_in terminal. Terminals of data_in[7:0] used by the module to receive the packet depend on the transmission speed. The `rec` module then puts the packet into a fifo and asserts a signal to `link_rec` to inform of the reception.

5.2 Link Layer

The modules for the link layer protocol deal with two type of asynchronous packet. The asynchronous stream packet is handled by `stream_trans` and `stream_rec`. Other asynchronous packets are handled by `asyn_trans` and `asyn_rec` modules.

- `stream_trans`: After receiving the signal from `ip_trans` to start transmission, `stream_trans` generates a link header, attaches it to IP data, and puts the data into a fifo of `trans` module. The header includes GASP and encapsulation headers. CRC is calculated twice: once for header and the other time for data. When the packet is ready to be transmitted, a signal is asserted to inform `trans` module that IP data has been put into the fifo and ready to transmit.
- `stream_rec`: On detecting the signal from `rec`, `stream_rec` takes the data from the fifo and checks the link header. If the transaction code is equal to 0xA of the GASP type, `stream_rec` will start the header check process. Otherwise, the packet will be ignored. The header check process checks header and data CRCs and the type field value. If both of the CRCs are correct and the type field equals to 0x0800 (IPv4) the packet is accepted and handed to the `ip_rec`. Otherwise, the packet is discarded and `stearm_rec` clears the fifo in `rec`.
- `asyn_trans`: This module includes two modules `asyn_req` and `asyn_res` for generating asynchronous request and response packets, respectively.
- `asyn_req`: This module acts as an asynchronous request packet generator to make request to another node.
- `asyn_res`: This module is used to make response to a request to its own node.
- `asyn_rec`: When receiving a packet which contains a transaction code other than 0xA of stream packet transmission, `asyn_rec` will process the packet and inform the `control` what to do with the packet.

5.3 Network Layer

The network layer consist of two module `ip_trans` and `ip_rec` for transmitting and receiving the IP datagram, respectively.

- `ip_trans`: This module starts its process on receiving a signal from the transport layer by generating its own IP address and providing other header fields including IP checksum. When the data is ready, it is put into a fifo of `stream_trans`. Then `ip_trans` sends a signal to `stream_trans` to inform that a data is ready.
- `ip_rec`: This module reads its fifo when it detects the ready signal asserted by `stream_rec`. It also checks the destination address. If the address matches with its own IP address, it will accept the packet and calculates the IP checksum. If the

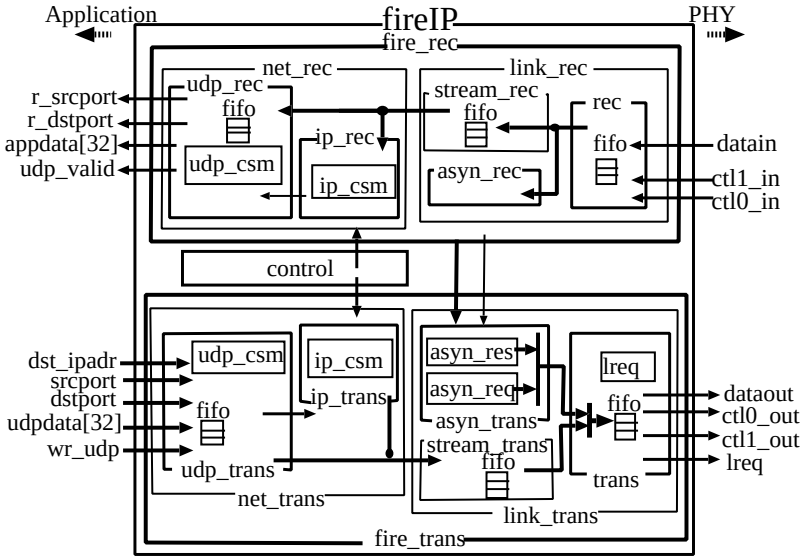

Fig. 5 Block diagram of `fireIP` module.

calculated value equals to the received IP checksum, `ip_rec` asserts a signal to its transport layer's module to inform of the packet arrival.

5.4 Transport Layer

The modules for the transport layer processes data requested by the application to transmit and data received from the network layer. For the transmission and reception, `udp_trans` and `udp_rec` modules are used, respectively.

- `udp_trans`: Data generated by an application is put into a fifo of `udp_trans` module, where a UDP header is attached to the data. The UDP header includes destination and source ports, data length, and UDP checksum. When data with the UDP header is ready, `udp_trans` sends a signal to `ip_trans` to inform that the UDP data is ready.
- `udp_rec`: This module checks length, header checksum, port etc. If matched, the packet is accepted and `udp_rec` asserts a signal to inform the application of the packet reception.

6. Implementation Method and Results

The Verilog-HDL code of about 4,500 lines in total was developed for the design above described.

A SRAM based FPGA chip APEX20K-400EBC652-1X[11] by Altera has been chosen for implementing the design. The FPGA is capable of realizing 400,000 gate logic circuits using Logic Elements (LEs) and 20 KB memory using Embedded System Blocks (ESBs). An EDA tool of Quartus II ver.2.0 provided by the FPGA vendor and a logic synthesis tool Synplify Pro7.1[12] by Synplicity have been used throughout the implementation.

The Verilog-HDL descriptions have been synthesized and converted to netlists using Synplify, which have been fitted to FPGA cells using Quartus. The compilation results are shown in Tab.2, where the numbers of LE's and ESB's required for realizing each component module of the `fireIP` and the maximum operating frequencies (f_max) of the compiled modules are listed. Relative numbers of LE's and ESB's used for each module per total LEs (16,640) and ESBs (212,992) available in the FPGA are also shown in percentage.

As shown in Tab.2, the maximum frequency of `fireIP` is 56.41 MHz. Because IEEE1394 is specified to operate at 49.152MHz, we can say that the design has an enough speed to operate. The module `fireIP` requires 22% of the 400,000 gates and uses 19% of ESB bits in the FPGA.

7. Design Verification Experiments

In order to verify the design, two sets of experiments (1) and (2) have been performed. In this section we describe the experimental environment, operations to be verified, and results of the verifications for each experiment.

Table 2 Compilation results.

Module	LEs*	ESBs*	f_max
fire_ip	3,661 (22%)	40,960(19%)	56.41
fire_trans	1,394 (8%)	16,384(7%)	71.49
fire_rec	2,171 (13%)	24,576(11%)	55.49
net_trans	945 (5%)	8192(3%)	71.35
udp_trans	510 (3%)	0(0%)	78.08
udp_csm	364 (2%)	0(0%)	58.84
ip_trans	303 (1%)	0(0%)	73.89
ip_csm	255 (1%)	0(0%)	114.98
net_rec	1,047 (6%)	8192(3%)	56.60
udp_rec	576 (3%)	8192(3%)	57.90
ip_rec	460 (2%)	0(0%)	70.04
link_trans	1,166 (7%)	16,384(7%)	60.91
stream_trans	404 (2%)	8192(3%)	108.12
asyn_trans	668 (1%)	0(0%)	81.29
asy_req	323 (1%)	0(0%)	175.44
asy_res	169 (1%)	0(0%)	194.25
trans	150 (1%)	8192(3%)	122.23
lreq	28 (1%)	0(0%)	263.71
link_rec	1,200 (7%)	16,384(7%)	59.35
asyn_rec	531 (3%)	0(0%)	73.45
rec	397 (2%)	8,192(3%)	112.57
crc	191 (1%)	0(0%)	-
fifo	39 (< 1%)	8192(3%)	186.78

*Total numbers of the available LEs and ESBs are 16,640 and 212,992, respectively.

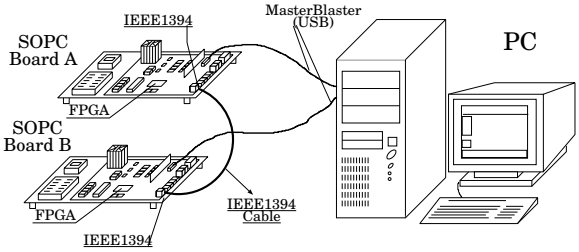

Fig. 6 System used for experiment (1).

7.1 Experiment (1)

The purpose of this experiment is to verify the proper communications between two `fireIP` modules.

7.1.1 Experimental Environment

The experiment has been performed using a system shown in Fig.6. It consists of two boards called SOPC boards A and B and a PC (Windows 2000 running on Pentium4 1.7GHz) for monitoring the behavior of the implementation using SignalTap function[13] provided by Quartus with ESBs. The PC is also used for compiling the design and configuring the FPGAs.

Each of the boards equips with the FPGA and an IEEE1394 PHY chip TSB41LV03[14] by Texas Instruments. A 6pin-to-6pin IEEE1394 cable and two cables called MasterBlasters are also used. The latter cables are used for configuring the FPGAs.

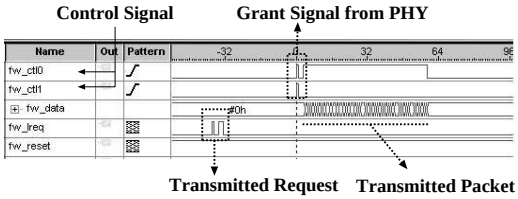

Fig. 7 Waveform observed at SOPC board A in experiment (1).

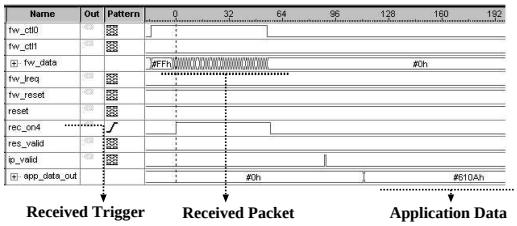

Fig. 8 Waveform observed at SOPC board B in experiment (1).

An IP address was allocatated to each board: 192.168.0.1 for SOPC board A and 192.168.0.2 for SOPC board B.

7.1.2 Operations to Be Verified

- **Transmiting packets**: A simple module has been designed to provide the application data and addresses for triggering the transmission of packet. The application module on the SOPC board A generates data (0x610A) and SOPC board B's IP address, and passes it to `fireIP` module where the data is encapsulated and transmitted to the SOPC board B. An outgoing packet is analyzed on the board A using SignalTap.
- **Receiving packets**: The SOPC board B waits for the incoming packet. The received packet is analyzed on the board B using Signal Tap. Received packets decapsulated by `fireIP` are compared with the transmitted data for verification.

A 49.152 MHz clock has been used for IEEE1394 system clock and the transfer rate of 400Mbps was employed. Note that the maximum frequency of fireIPs is high enough for the clock rate.

7.1.3 Results of Verifications

Figs.7 and 8 show a transmitted packet observed at the SOPC board A and a received packet observed at the SOPC board B. As shown in the figures the waveforms observed are as expected: the data 0x610A reached at the application layer have been checked to be the same as that transmitted by the sender's

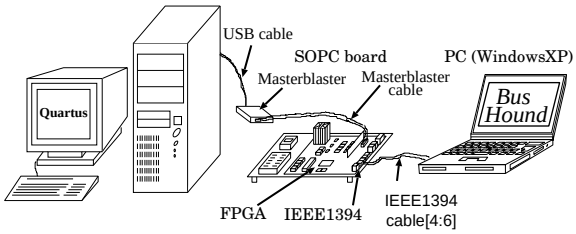

Fig. 9 System used for experiment (2).

application layer. Values of CRC, IP and UDP checksums are calculated correctly at both sides, and the address check functions of each module also work well.

7.2 Experiment (2)

As explained previously, IP-over-1394 packets use the GASP format, where a channel number reserved for communication between nodes is specified. Before transmitting an IP-over-1394 packet, `fireIP` needs to reserve a channel from an IRM node. The purpose of this experiment is to verify `fireIP` to be able to reserve a channel by communicating with the IRM node.

7.2.1 Experimental Environment

A set of sub-experiments have been performed using the system shown in Fig.9. It consists of a PC (WindowsXP) taking a role of IRM node and an SOPC board implementing the `fireIP` module. A software tool, Bus Hound 4.0[15], is used at the PC for monitoring 1394 packets.

7.2.2 Operations to Be Verified

Here we verify the following operations:

- **Obtaining IRM node identifier**: As described in section 4.3, the IRM function which manages the channel allocation is assumed to be implemented on a gateway, which is the PC in this experiment. The IRM node will broadcast a cycle start packet. By receiving the packet, `fireIP` detects the identifier of the IRM node.
- **Obtaining valid channel numbers**: By sending a read request data quadlet packet to an address location of the IRM node, valid channel numbers are obtained.
- **Reserving a channel**: A channel number must be locked to reserve. The lock operation starts by sending a lock request packet to the IRM node. If a lock response packet is responded with no error, the lock operation ends successfully.

7.2.3 Results of Verifications

Fig.10 shows the waveforms observed in the experiments. At the bottom we see that the

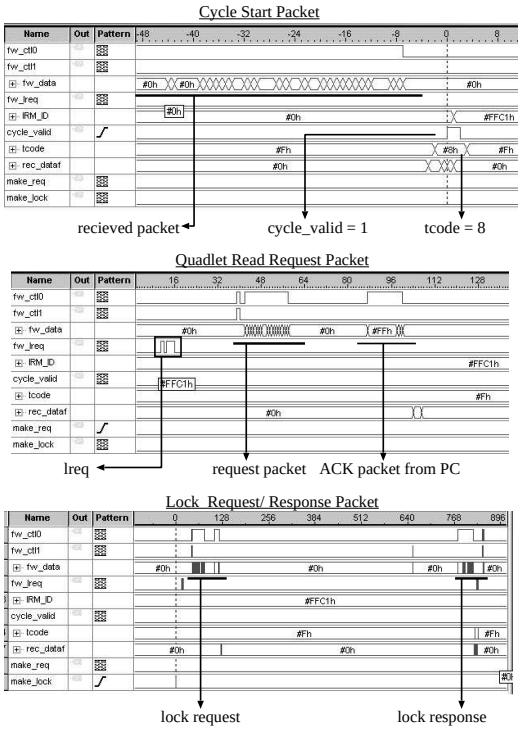

Fig. 10 Waveforms observed at SOPC board in experiment (2).

lock operation ends successfully by receiving a lock response packet properly. A cycle start packet from the IRM node notifies the IRM node identifier. A channel selected from valid ones indicated in the quadlet data responded to the read request packet was found to be reserved by lock request/reply transactions. We therefore verified all the operations required for reserving a channel used for transmitting IP-over-1394 packets.

8. Evaluation and Discussion

As shown in Tab.2, the maximum frequencey f_max of `fireIP` is faster than 49.152 MHz which corresponds to 400 Mbps, the upper limit of 1394's throughput. Thus the processing speed of the module is faster than the communication speed of the 1394 medium, meaning that the performance of the implementation meets the requirements.

The hardware cost in terms of number of FPGA logic elements of the IEEE1394 link layer is 2,366, while total costs of network, transport, and application layers are 1,992. This means that the upper layers' costs are 84% of the link layer's. That is, when implementing IP over 1394 by hardware, the cost does not ex-

ceed twice the cost of the IEEE1394 link layer which is normally realized by hardware and its cost can be regarded dispensable.

As for the absolute value of the costs, the whole protocol stack of IP over 1394 is found to use 22% of 400 Kgates FPGA, which are 0.2% of Pentium 4 processor requiring 42,000 Kgates[16].

In this paper the descriptions of the hardware design have been synthesized and mapped onto an FPGA. They can easily be retargetted to an ASIC. The recent progress in semiconductor technology enables larger number of gates to be fabricated on a chip. In such circumstances the IP-overd-1394 stack can be embedded as a functional module in a system-on-a-chip. Because the area cost required for the stack is negligible compared to that of high performance processors, spending a little extra cost leads to improving performance largely. The improvements will be significant when transmitting and processing AV data occurs frequently.

We measured the hardware cost by the amount of the required logic elements in the FPGA. The FPGA is suitable for prototyping a design but the design realized on an FPGA is generally slower and larger than that realized on an ASIC. An ASIC version of the current IP-over-1394 design will yield an implementation with higer speed and lower gate counts. Recently the IEEE1394 serial bus has bee revised to support 3.2Gbps[7]. It is possible to obtain a chip operating at this speed if the design described in this paper is realized as an ASIC.

9. Conclusions

A hardware design of IPv4-over-1394 stack and its implementation on FPGA have been described. The protocol stack includes UDP, TinyIP, IEEE1394 as transport, network, and link layers, respectively.

Using a system clock of 49.152MHz, we verified that packets sent from an application on top of the protocol stack were correctly received by the other protocol stack via IEEE1394 port at a transfer rate of 400 Mbps. We also verified communication behaviors of the design with an isochronous resource manager to reserve a channel prior to data transmissions. The hardware cost of the IP layer is 84% of the link layer's cost, meaning that the IP-over-1394 protocol stack can be realized by hardware by spending an extra cost less than that of the basic hardware which already exists.

Communications with software versions of IP-over-1394 have not been confirmed yet. Software implementations of IP-over-1394 are different among operating systems[3),4)]. That implies the standardization of the protocol should yet be scrutinized. Because all the needed transactions have been verified, it would be easy to verify for our implementation to communicate with available IP-over-1394 software implementations if their implementation-dependent parts are identified.

Acknowledgments We would like to express our thanks to Dr. Masachika Harada and Dr. Atsushi Takegami with Texas Instruments Japan Ltd., who provided us with valuable information. We also would like to thank Dr. Takamichi Tateoka with the Department of Computer Science at the University of Electro-Communications for his important suggestions and discussions.

References

1) IEEE Standards Board, "IEEE Standard for a High Performance Serial Bus," *IEEE Std 1394-1995*, 1995.
2) P. Johansson, "IPv4 over IEEE1394," *RFC2734*, 1999.
3) Open Source Development Network, http://www.linux1394.org, 2002.
4) Microsoft Co., http://www.microsoft.com, 2002.
5) K. Morita and K. Abe, "Implementation of UDP/IP Protocol on FPGA and Its Performance Evalution", *Proc. IPSJ General Conf.*, Special 5, pp.157-158, 2001.
6) Y. Izuhara, K. Morita, T. Tateoka, and K. Abe, "Specification of TinyIPv6 Protocol Stack for Remote Control and Its Implementation on FPGA," *IPSJ Journal*, Vol.43, No.11. pp.3540-3548, 2002.
7) IEEE Standards Board, "IEEE Standard for a High Performance Serial Bus - Amendment 1," *IEEE Std 1394b-2002*, 2002.
8) IEEE Standards Board, "IEEE Standard for a High Performance Serial Bus - Amendment 1," *IEEE Std 1394a-2000*, 2000.
9) W. R. Stevens, *TCP/IP Illustrated, Volume 1: The Protocol*, Addison-Wesley, 1994.
10) Takamichi Tateoka and Kôki Abe, "Design and Implementation of TinyIP: A Restricted UDP/IP Stack for Educational Use," *Proc. of Internet Conf.*, pp.99–106, 2002.
11) Altera Co., *APEX 20K Programmable Logic Device Family,* http://www.altera.com/literature/lit-apx.html, 2002.
12) Synplicity Inc., *Synplify Pro 7.1 Release Notes,* http://www.synplicity.com/literature/index.html, 2002.
13) Altera Co., *SignalTap Analysis in the Quartus II Software Ver.2.0,* http://www.altera.com/literature/an/an175.pdf, 2002.
14) Texas Instruments Inc., *TSB41LV03 IEEE 1394a Three Port Cable Transceiver/Arbiter*, 1998.

15) Perisoft, http://www.perisoft.net/bushound.
16) Intel Co., http://www.intel.com/technology/itj
/q12001/articles/art_2.htm, 2001.