Design and Implementation of Load-Balanced Multipath Self-routing Switching System

Qian Zhan, Zhi-Pu Zhu, Li Ma, and Hui Li*, *Member, IEEE*

School of Electronic and Computer Engineering, Peking University

Shenzhen Eng. Lab of Converged Networks Technology; Shenzhen Key Lab of Cloud Computing Tech. & App

Shenzhen, China

zhanqian0218@gmail.com

*corresponding author: huilihuge@163.com

Abstract— **In order to ensure high quality of service (QoS) for Next Generation Network (NGN), we construct a new Load-Balanced Multipath Self-routing Switching Structure which consists of the same two multipath self-routing fabrics. The result of simulation is inspiring for achieving 100% throughput and no delay or jitter. For this reason, we start on the implementation on an Altera StratixIV FPGA. And the whole FPGA system is designed into two collaborative components: the UDP system and the register system. With two algorithms around input and output two stages, incoming traffic is transformed into uniformity and then to their final destinations. During the later period debugging, software simulation platform and automated test platform are built, which contribute to our work very much. At last, we carry out several experiments to test and verify our system. The report of the test result accords with what we expected.**

***Keywords*— Load-Balanced; Multipath Self-routing Switching Fabric; FPGA; UDP; Register**

I. INTRODUCTION

In recent years, with a rapid increase in the number of Internet users, the network scale expands unceasingly. Rich Internet applications, especially the popularity of online video services [1], contribute to the network congestion that almost everyone experienced. This phenomenon puts forward a huge challenge to the vital component, the router. Actually, the router has become a significant bottleneck in the development of the network. On the other hand, on the basis of TCP/IP, the network layer of Internet only provides the best effort delivery rather than the commitment for quality of service (QoS) [2]. Therefore, looking for a new switching system which is more efficient and supports QoS is a key research and development point.

In order to improve the performance of routers and reduce implementation costs, various kinds of solutions are proposed. The Load-Balanced Birkhoff-von Neumann switch [3] interests us for that it can achieve 100% throughput with most network traffic by using a balancer to equalize input flows. However, the structure does not maintain the order of packets after switching and the average queuing delay increases linearly with the number of ports. Obviously, it is not suitable for large scale extension. On the contrary, another structure we focus on, the Banyan-based Quasi-Circuit Switch [4] has low component complexity O ($N\log_2 N$, N is the number of ports) and the ability of self-routing and distributed processing. However, because of the blocking feature, QoS is not ensured.

Based on the advantages and disadvantages of the above two kinds of structures, we propose a Load-Balanced Multipath Self-routing Switching Structure by connecting two multipath self-routing fabrics in series. The first one acts as a load-balancer and the other one severs as a self-routing forwarder. Concentrators, which are made up by basic sorting units, are sorted by the arrangement rules of Multistage Interconnection Network to construct the whole structure. Theoretical analysis and NS2 simulation indicate that our model can obtain 100% throughput under normal circumstances and easy to be expanded in size [5].

Further, we translate the theoretical model into a modular FPGA system which consists of two main parts: the UDP system and the register system. And then, the whole system has been implemented on an Altera StratixIV FPGA. In the testing phase, our system works steadily and efficiently and meets the basic requirements of QoS applications.

The rest of the paper is organized as follows. Theoretical basis and modeling are introduced in Section II. Section III describes the system design and implementation based on FPGA. Section IV presents system testing with real network traffic, and then Section V summarizes the whole work.

II. THEORETICAL BASIS AND MODELING

The overall switching system consists of load balancing stage, routing switching stage and some auxiliary modules. Both of the two main components are structured by 2×2 sorting units which are based on the theory of algebraic distributive lattices and arranged in certain order. In order to meet the performance requirement and make the price to the minimum, not only have we designed a new network structure according to the basic function of the sorting unit, but also a perfect in-band signaling system and the matched control mechanism.

A. 2×2 Basic Sorting Unit

The 2×2 basic sorting unit is a sequential logic circuit, with two inputs and two outputs (respectively called 0/1 port). As shown in figure 1 (a) and (b), if conflict-free, it has two states:

Bar and Cross [6]. When both inputs contend for the same output, sorting unit will randomly select one of them as the winner and send its packets through the unit; the loser's data will be dropped or misrouted (see figure 1(c)).

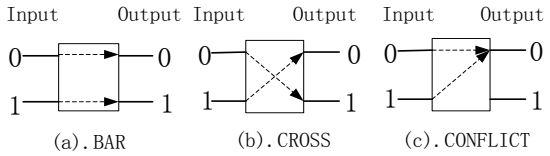


Fig. 1 2×2 basic sorting unit and its states

In-band control signalling can be used to set the connection state of the unit as list in Table I. The unit compares the two-bit in-band signalling A and D of each packet to make the routing decision. The first bit A indicates the activity of an input packet. When A equals 1, it means an active packet is arriving. The second bit D indicates the destination of an input packet. Thus, 10, 11, 00/01 respectively represent the packet which is going to output-0, the packet which is going to output-1 and the dummy packet. Under the synchronous clock, 2×2 basic sorting units act upon the rule: 10<00/01<11, which can also be used in sorting concentrators.

TABLE I. TWO-BIT IN-BAND SIGNALING CONTROL MECHANISM.

Connection State		**Input-1 Control Signaling: A1D1**		
		10	***00/01***	***11***
Input-0 Control Signaling: A0D0	***10***	CONF [a]	BAR	BAR
	00/01	CROSS	EITHER	BAR
	11	CROSS	CROSS	CONF

a. When CONF (CONFLICT), priority decides the state.

According to the theory of algebraic distributive lattices [7], we can further define Ω_{route} = {0-bound, 1-bound, idle}, namely, 0-bound = 10, 1-bound =11, idle = 00/01. So, the former 10<00/01< 11 turns into 0-bound< idle<1-bound. If it is conflict, the choice of BAR or CROSS depends on a specific application such as the priority.

B. Inter-stage Bit-permuting Model

An N×N ($N=2^n$) routing network is a Multistage Interconnection Network (MIN) built by 2×2 basic sorting units. By using first stage permutation σ_0, inter-stage permutation σ_1, σ_2 … $\sigma_{(n-1)}$ and last stage permutation σ_n, the network can be represented as $[\sigma_0: \sigma_1: \sigma_2: \ldots: \sigma_{(n-1)}: \sigma_n]$. Each colon symbolizes a stage of 2x2 units. We can define a Trace sequence and a Guide sequence [8] as follows:

- $T_k=(\sigma_0\sigma_1\ldots\sigma_{K-1})^{(-1)}(n)$ $1\le k\le n$;
- $G_k=(\sigma_0\sigma_1\ldots\sigma_{K-1})(n)$ $1\le k\le n$;

Trace and guide can find a unique route from input to output. As Fig. 2 shows, for the network [: (43): (42) (31): (43):], the Trace is (4, 3, 2, 1) and the Guide is (1, 2, 3, 4). Then, origination address bits $I_1I_2I_3I_4$ one by one are rotated to the rightmost bit position at the successive stages and are replaced successively by bits $O_1O_2O_3O_4$. The destination address bits $O_1O_2O_3O_4$ are specified by Trace or Guide.

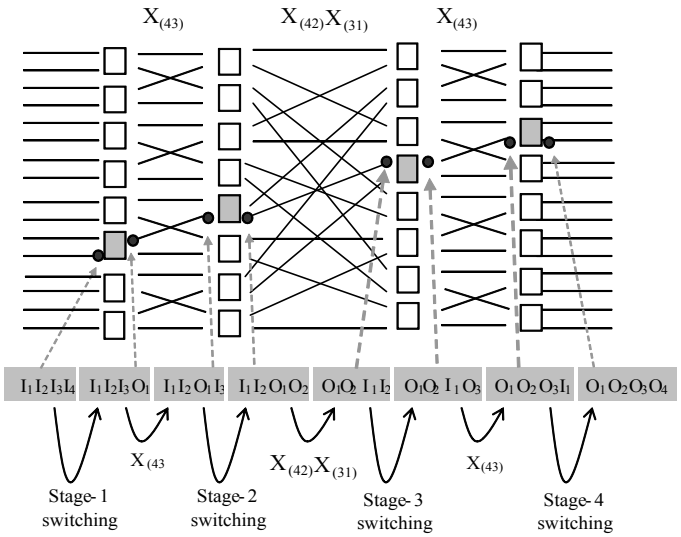


Fig. 2 an example of routing network

C. Multipath Self-routing Switching Structure

Multipath Self-routing Switching Structure (MSSS) [9] is an innovative structure, which combines Multistage Interconnection Network (MIN) with concentrators.

The MIN described in section II B is constructed by plenty of basic sorting units, which are divided into several stages. Each unit works at the state of CROSS or BAR by its rule. This kind of network has the advantages of being highly modular and having low device complexity O ($N\log_2N$). With the help of Trace and Guide, there is a unique route from an input to any output. For this reason, it avoids the scheduling at each time slot and has the ability to be massively expanded.

To construct MSSS, we substitute each basic sorting unit for 2G-to-G concentrator and replace the single cable with a bundle of cables. Fig. 3 illustrates the multipath structure (N=128 M=16 and G=8) which is based on a 16×16 routing network. G shows the size of the group, M is the number of the group and N=M×G indicates the whole number of input/output ports ($G=2^g$, $M=2^m$, $N=2^n$, n=m+g, n, m, g are positive integers). Obviously, we have replaced the basic sorting units in Fig. 2 by 16 to 8 concentrators.

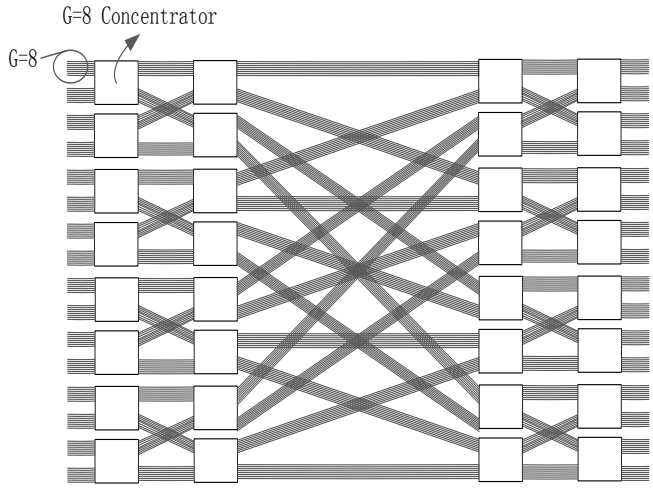


Fig. 3 Multipath Self-routing Switching Structure (M=16, G=8)

Acting as an indispensable part of MSSS, the 2G-to-G concentrator [10] separates the larger G signals of the whole 2G inputs from the other G signals. Finally, it forms two output groups. (The output order within each group is arbitrary.) Intuitively, a 2G-to-G concentrator can be built by two G×G sorting networks for arbitrary 0-1 sequence, each of which is followed by a G-half cleaner. Address arbitrators are attached to the whole outputs to clear misrouted packets. All the above-mentioned sorting networks, half cleaners and arbitrators are constructed by basic sorting units.

D. Load-Balanced Multipath Self-routing Switching Structure

In the project, the size of arriving packets is random. And furthermore, time is slotted and synchronized so that packets can be transmitted within a time slot for each input line of the structure.

As shown in Fig. 4, two MSSSs are used in series to compose the whole structure, with the VOGQs (Virtual Output Group Queues) [5] ahead of the first fabric and the assemblages at the end of the second fabric. Actually, by using simple algorithms and small buffers, the first stage fabric serves as a load-balancer, which spread any pattern of incoming traffic which is to be distributed uniformly to all the ingress ports of the second stage fabric. Then the second stage fabric forwards the data in a self-routing manner to their final destinations. Every G inputs/outputs are bundled into an input/output group. Thus N input lines form M groups on the input side (N=M×G), so is the output side. To ease presentation, IG/OG denotes input/output group, and MG represents a line group between the two stages. In this project, there are 4 IGs, 4 MGs and 4 OGs. Each group has 8 lines. .

VOGQs are responsible for storing packets and making data ready for IGs. We use VOGQ (i,j) to denote the VOGQ whose packets are destined for OGj from IGi.

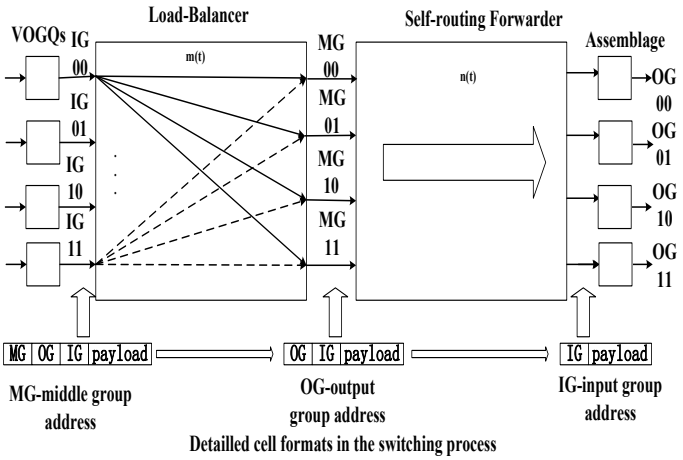


Fig. 4 Load-Balance Multipath Self-routing Switching Structure

Generally, for the system we proposed, the processing of arriving packets at each time slot is composed by several sequential phases which are shown as follows. In addition, to achieve maximum processing speed, we should use pipeline structure as far as possible.

1) *Preparatory phase*: New packets arrive during this phase. With checking and judging, the packet which is destined for OGj from IGi, is stored into VOGQ (i,j).
2) *Splitting phase*: Packets in VOGQs are split into cells according to Algorithm 1. And each cell will be added with some certain packet headers.
3) *Balancing phase*: With the help of MG tags, cells will be routed to every middle group simultaneously and uniformly. When the cells reach the middle groups, MG tags will be dropped.
4) *Routing phase*: Cells are further to their final destinations directed by OG tags. When they get through the second stage fabric, a self-routing forwarder, the OG tags will be discarded.
5) *Assembling phase*: Cells which arrive simultaneously are to be assembled to original packets according to Algorithm 2. When completed, packets will be output from the OGs.

Algorithm 1: For each input group, packets stored in VOGQs should be split into cells with equal length during splitting phase. Furthermore, we add MG tags, OG tags, IG tags and some other control messages ahead of each cell. The MG tags are set artificially. For example, a packet is split into five cells and their respective MG tags should be 0, 1, 2, 3, 0, orderly. If the following packet can be split into three cells, the tags will be 1, 2 and 3. The rule is also suitable for other various packets.

Algorithm 2: For each output group, cells with the same IG address are assembled during assembling phase. IG tags, sequence numbers and flags of the last cells will help us to reorganize the scrambled cells. For example, we get a few of cells in OG address 01. Some of them have the same IG tag 00 and the sequence numbers 3,2,4,1. By the way, the cell with sequence number 4 is marked with the trailing flag. The others

own IG tag 10 and the sequence numbers 3, 2, 1, but no cell has the trailing flag. By now, we can easily get the packet which is from IG 00 by connecting the cells together in the order 1,2,3,4. For the packet from IG 10, we still need to wait for the last cell to arrive.

As Algorithm 1 is introduced, some extra bytes are added at the same time, like the tags, flags and so on. Now, we try to analyse the extra overhead of our theoretical model. In the system, the standard size of a cell is 128Byte and the header of a cell has 8Byte additional control information. For a 1000Byte packet (1000=7×128+104), it will be split into eight cells and the last cell has only 104 bytes. To keep the same size, 24Byte invalid information should be padded to the last cell. Consequently, all the overhead is 8×8+24=88Byte (8.6%). The calculation above is just for the 1000Byte packet, and generally, different sizes of packets would be split into different numbers of cells. Besides, various last cells may have diverse numbers of padding bytes. Considering that the size of a standard MAC frame ranges from 64Byte to 1518Byte, the worst case appears when the packet size is 1409Byte (1409=11×128+1) and the extra overhead is 11×8+127=215Byte (14%). When a 128Byte packet arrives, we would be happy for the extra overhead only occupies 8 bytes (0.0625%). In practical applications, the extra overhead is always acceptable for the reason of statistical average.

III. SYETEM DESIGN AND IMPLEMENTATION BASED ON FPGA

We use Verilog HDL to carry on the main design and Tcl script language to build an operating platform for the register system. Functional simulation is also an important part, which is implemented by Perl and Makefile. Perl can be used for generating different kinds of packets and Makefile usually for simulation platform.

A. the Overall Architecture of the System

The whole system is implemented on an Altera StratixIV FPGA, with a Marvell 88E1111 PHY chip being used for physical layer. The TSE (Triple-Speed Ethernet) IP core, interacted with the PHY chip through RGMII interface, provides standard Ethernet frames.

We divide the system into two main parts, the user data path (UDP) system (There are the same four UDP systems in the whole system, each of which servers a group of MSSS and we just need to introduce one of them in the paper.) and the register system. They are independent structurally and interrelated functionally. The UDP system is responsible for data processing and cell switching with many sub-modules, FSMs (finite-state machines) and FIFOs in it. The data flow is shown as the dark thick arrows in Fig. 5. In order to facilitate debugging, we have designed the register system to monitor signals and states in UDP in real-time. Its data flow is shown as the light-coloured thick arrows in Fig. 5.

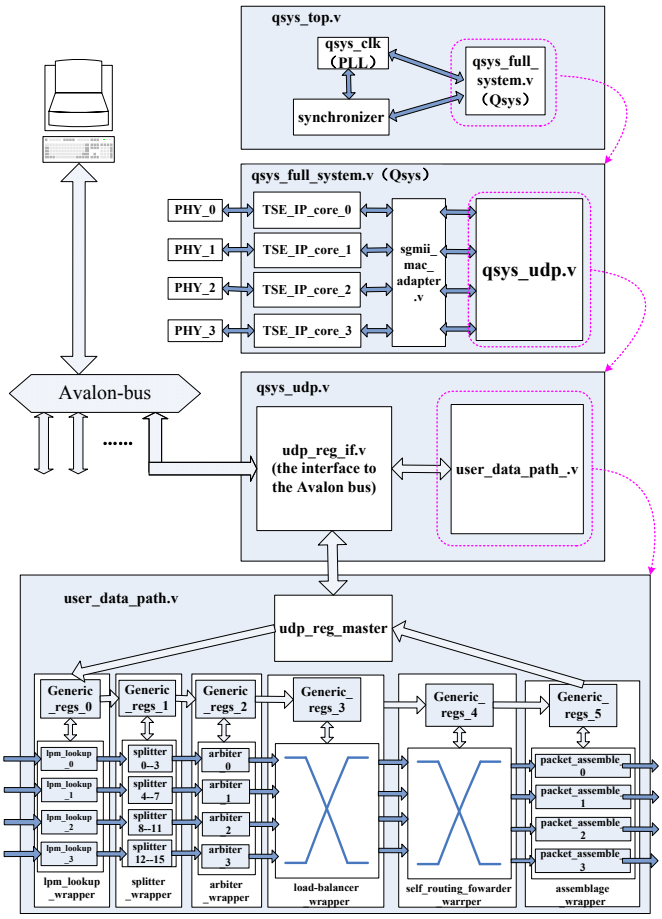


Fig. 5 the overall architecture of the system

Fig. 5 describes the overall architecture of the system and it contains detailed information of ever hierarchy. At the top, the phase-locked loop (PLL) provides high-quality clock signal for the entire system and the synchronizer can be used to reset the system. Qsys_full_system consists of four ISE IP cores, sgmii_mac_adapter and qsys_udp. The latter two parts are fully designed by us. Qsys_udp includes udp_reg_if and user_data_path. Udp_reg_if is the interface between the Avalon bus and registers. User_data_path is the most complicated one, which covers seven main modules. They are lpm_lookup_wrapper, splitter_wrapper, arbiter_wrapper, load-balancer_wrapper, self-routing_forwarder_wrapper, assemblage_wrapper and udp_reg_master. The first six modules own their respective sub-modules and the same register generic_reg. Udp_reg_master control all the registers below and it connects to udp_reg_if.

Lastly, the logic utilization is 32% according to the compilation report generated by Quartus II 11.0.

B. Design and Analysis of User Data Path System

Packets enter into the system through the RJ45 network port firstly. And after being processed in physical layer by PHY chip, they will be sent to the UDP system, which is the major part of data processing. There are four main functions

in UDP. First of all, we can extract necessary information from packets or cells, such as the packet length, priority and the target address, etc. Second, by utilizing the information we extract, the UDP system generates various packet headers, which will be very useful to assist the data processing. Third, it achieves load balancing and self routing by constructing the switching fabric. At last, it completes the assembling of the cells.

The left part of Fig. 6 gives us a full view of the process. The solid arrows indicate the direction of data flow. We can see that input packets pass through nine sub-modules (not including PHY) in turn and get back to PHY.

The functions of each sub-module are as follows.

1) *Sgmii_ethernet*: It is an interface module between the UDP system and the external PHY chip. Mainly constructed by Altera Triple-Speed Ethernet (TSE) IP cores, it provides standard Ethernet frames.
2) *Rx_queue*: This sub-module accepts frames, extracts information and generates the splitting header. The information is important for Splitter and will be kept until Assemblage.
3) *Lpm_lookup*: It extracts the information of destination address and priority, which form the LPM header for Self-routing Forwarder.
4) *Splitter*: For efficiency and easiness of implementation, the following sub-modules are designed based on cells. So, the Splitter will be a key sub-module. It splits each packet into several cells and generates three kinds of headers, the load balancing header for Load balancer, the self-routing header for Self-routing Forwarder and the assembling header for assemblage.
5) *Arbiter*: As we know, the group size of MSSS we proposed is G=8. Thus, cells should be placed as a group of eight lines. The size of data on each line is 10bit (8 bits for payload and 2 bits for a control signal). This is what Arbiter do.
6) *Load-balancer*: The structure is the same as MSSS. It transforms the incoming traffic into uniformity.
7) *Self-routing Forwarder*: It is also a MSSS. Cells switch here and then go to their final destinations.
8) *Assemblage*: After switching, groups of cells arrive at every time slot. Assemblage assembles them back to standard Ethernet frames and generates the starting index header, which is just to show the very beginning of each frame.
9) *Tx_queue*: It contains some memory buffers to cache the frames and then sends them back to Sgmii_ethernet.

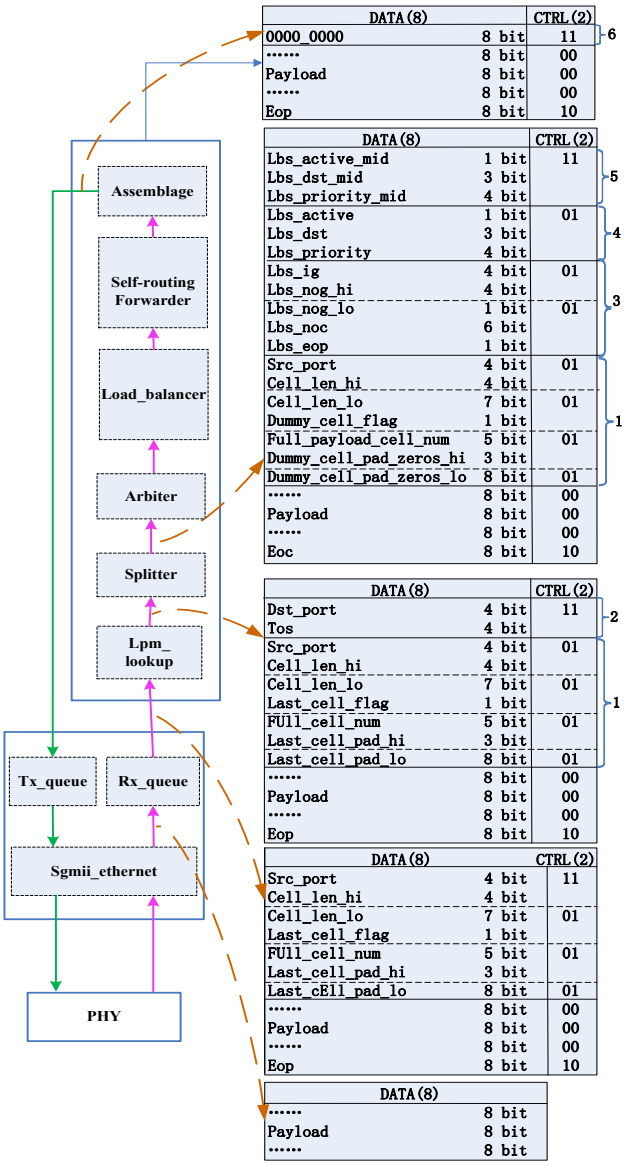


Fig. 6 data processing and data format in UDP

The right part of Fig. 6 pointed by dotted arrows describes the specific data format in each sub-module. Moreover, as shown in the right part, the standard size of data in UDP is 8bit. After going across Rx_queue, splitter_header is produced along with the extra added CTRL signal. The 2bit CTRL signal will be transmitted with data in parallel to assist data identification and processing. The signal "11" signifies that the concurrent data is the first byte of a fresh new packet or cell. And "01" represents the various kinds of headers we generate. "00" shows the payload data and "10" tells us the end of a packet or cell.

To sum up, the UDP system extracts the information from incoming frames, and then generates six kinds of headers which are attached in front of the former frames. We use labels "1", "2", "3", "4", "5", "6" to represent splitting header, LPM header, assembling header, self-routing header, load balancing header and starting label header, respectively.

1) *Splitting header*: It is created by Rx_queue and contains 4 bytes information. Src_port is the source address of the current frame. The 4bit signal can represent 16 ports, but the group size of our system is M=4. Obviously, it is convenient for scale expansion in the future. The sum of cell_len_hi and cell_len_lo is 11bit, which indicates the standard length of the cell. We set them 2`b0001 and 7`b000_0000, meaning 128Byte. Last_cell_flag indicates whether the current frame can be split into a certain number of cells exactly. Set to high, the signal means that the size of the packet isn`t an integral multiple of 128Byte. Thus the last cell should be padded some bytes to keep the same length. Then, last_cell_pad_hi and last_cell_pad_lo show the number of bytes to be padded. Finally, signal full_cell_num is the number of complete cells.
2) *Lpm header*: Compared with others, it`s a simple one for containing only one byte information. Dst_port is the destination port of the packet and tos means the priority.
3) *Assembling header*: Generated by Splitter and carrying important information for Assemblage, this header helps to assemble the cells later. Lbs_ig corresponds to the input group number IG shown in Fig. 4. We know that Arbiter places the cells on a group of eight lines. Lbs_nog_hi and lbs_nog_lo point out which line the cell belongs to. We can enlarge the group size to 32 furthest. Lbs_noc is the significant serial number, declaring the actual position of the cell in the original packet. The last signal lbs_eop is the mark of the last cell. We can use these signals to reassemble the cells and specific methods are mentioned in Algorithm 2.
4) *Self-routing header*: It is just the output group number OG shown in Fig. 4 and it will be dropped after passing through Self-routing Forwarder. In fact, it is the recoding of the Lpm header. Analogously, lbs_active proves the cell active and lbs_dst gives the destination. Reviewing the 2×2 basic sorting unit introduced in the beginning, if the state is conflict, signal lbs_priority will make the decision.
5) *Load balancing header*: Being similar to self-routing header, lbs_active_mid is the significance bit and lbs_priority_mid indicates the priority. The only difference between the two headers is that the destination shown by lbs_dst_mid is the middle group number MG in Fig. 4. Load balancing header is set according to Algorithm 1 and it will be discarded after passing through Load-Balancer.
6) *Starting label header*: Being simple but necessary, it is used to show the very beginning of the frame after assembling.

C. Design and Functional Simulation of Key Sub-modules in UDP

The design and function realization of a module need to pass the test of simulation software firstly. Furthermore, by using functional simulation, we can visually study the design details, modify the errors and improve efficiency. Our simulation platform is built with the help of three main tools: Perl scripting language, Makefile scripting language and the simulation software Modelsim.

The full name of Perl is Practical Extraction and Report Language and it can easily manipulate numbers, texts, files, and directories. Hence, we take advantage of it to generate various kinds of standard Ethernet frames and some other packets with certain headers. All the data is saved as texts as the input of the whole system or a sub-module.

Makefie is a scripting language with the functions of executing programs and describing their relationship. In the project, there are four IP cores, many modules and complex hierarchies. If we only use the graphical interface or directly use the Modelsim command line, workload will be very huge. Making use of the powerful tools and abundant functions involved in Makefile, we can greatly simplify the work of building the simulation platform.

Modelsim is famous for its friendly command-line operating environment and high accuracy. Commands “vlib”, “vlog”, “vsim” and “add wave” will be frequently used, which represent “building simulation library”, “compiling”, “simulating” and “adding waves”. With using Modelsim, all the operations in a complex graphical interface can be implemented through the command line and run by the script program. After efforts, the automated simulation comes true.

Next, we will introduce the design and verification in detail of several key modules based on the above-mentioned platform.

1) Splitter

As shown in Fig 7, the internal structure of Splitter is simple and intuitive. And its function is just to cut the input variable-length packets into output equal-length cells. But in fact, the realization of this process is not easy. Specific “assembling header”, “self-routing header” and “load balancing header” should be attached to the head of each cell to insure the proper functioning of the three following sub-modules. This attributes the success to the powerful output state machine Out_FSM.

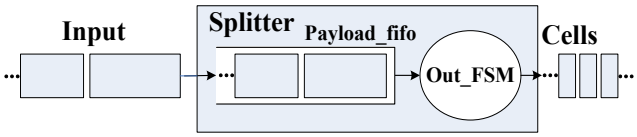


Fig. 7 the design of sub-module Splitter

Fig. 8 displays the simulation waveform of Splitter, the upper half of which shows the incoming packets. We can distinguish them clearly with the help of signal in_wr and signal in_ctrl. Configured by Perl, the lengths of packets in the four rounded rectangles are 70Byte, 128Byte, 256Byte and

429Byte. Based on the principle of splitting, the first one should be padded 58 bytes to reach 128Byte long, the second one is just right a cell and the third one will be cut into two complete cells. The fourth packet has some hurdles: the last one of the four dissected cells needs byte stuffing. In the second half of Fig. 8, we can see the cells and the padding bytes 8`hee.

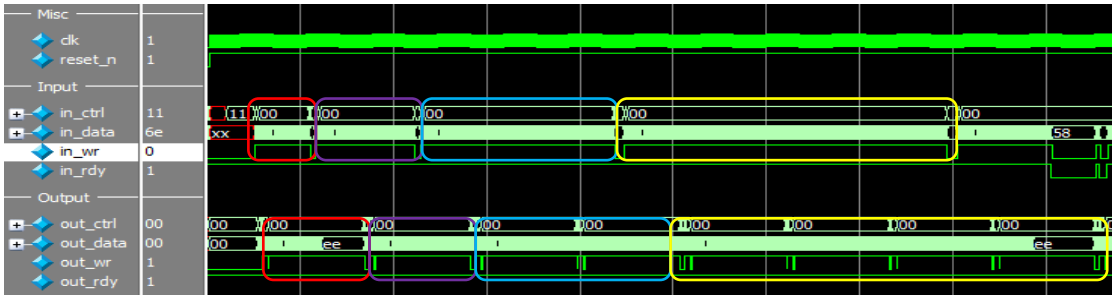


Fig. 8 the simulation waveform of Splitter

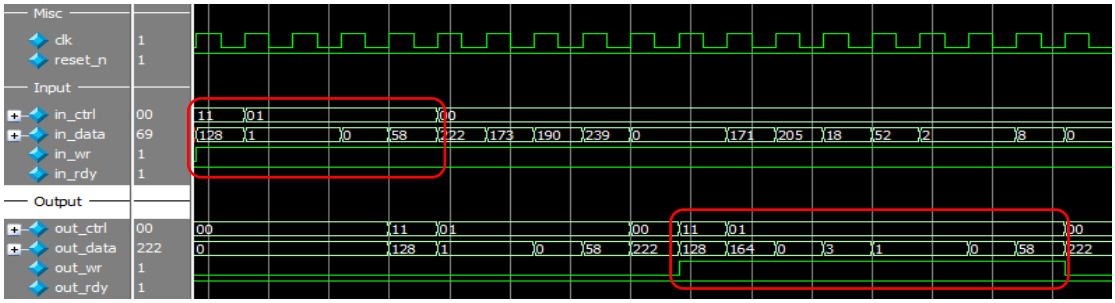


Fig. 9 the details of the header information

Fig. 9 describes the details of the header information. Shown in the two rounded rectangles, input data is the output of Lpm_lookup and it has five bytes header information. When it outputs from Splitter, its header contains eight bytes data: one byte for load balancing header, one byte for self-routing header, two bytes for assembling header and four bytes for splitter header.

2) Multi-path Self-routing Fabric

Composing Load-Balanced MSSS, both Load-Balancer and Self-routing Forwarder are the same MSSS. So we take one for example.

MSSS is the heart of the whole system and it consists of plenty of concentrators which are connected as the way shown in Fig. 3. The concentrator is constituted by 2×2 sorting units which perform the basic operation: comparing the incoming two cells. In the project, a total of 32 ports transmit cells simultaneously because G equals to 8 and M equals to 4.

Fig. 10 is the simulation waveform of MSSS and the cell is represented as the data in the circle. Shown as a column, 32 cells are dispatched collectively at every time slot. It is empty if there is no a cell to be sent in a port. In the middle, the width of the rounded rectangle indicates the time interval of cell sending, which is defined as time_slot_plus. The value of time_slot_plus is not set arbitrarily but by calculation. When Arbiter is sending a 136Byte cell (including 8 bytes header information), one byte will leave at every clock rising edge. All of the ports perform the same operation at the same time. On the opposite, Assemblage captures the cells which arrive concurrently, one port by one port. Consequently, the process needs enough time and enough buffers. In theory, 136×8=1088 clock cycles are necessary and we use the number 1100 just in case. This is why the gap between two cells owns about 8 times the width of a cell (see Fig. 10).

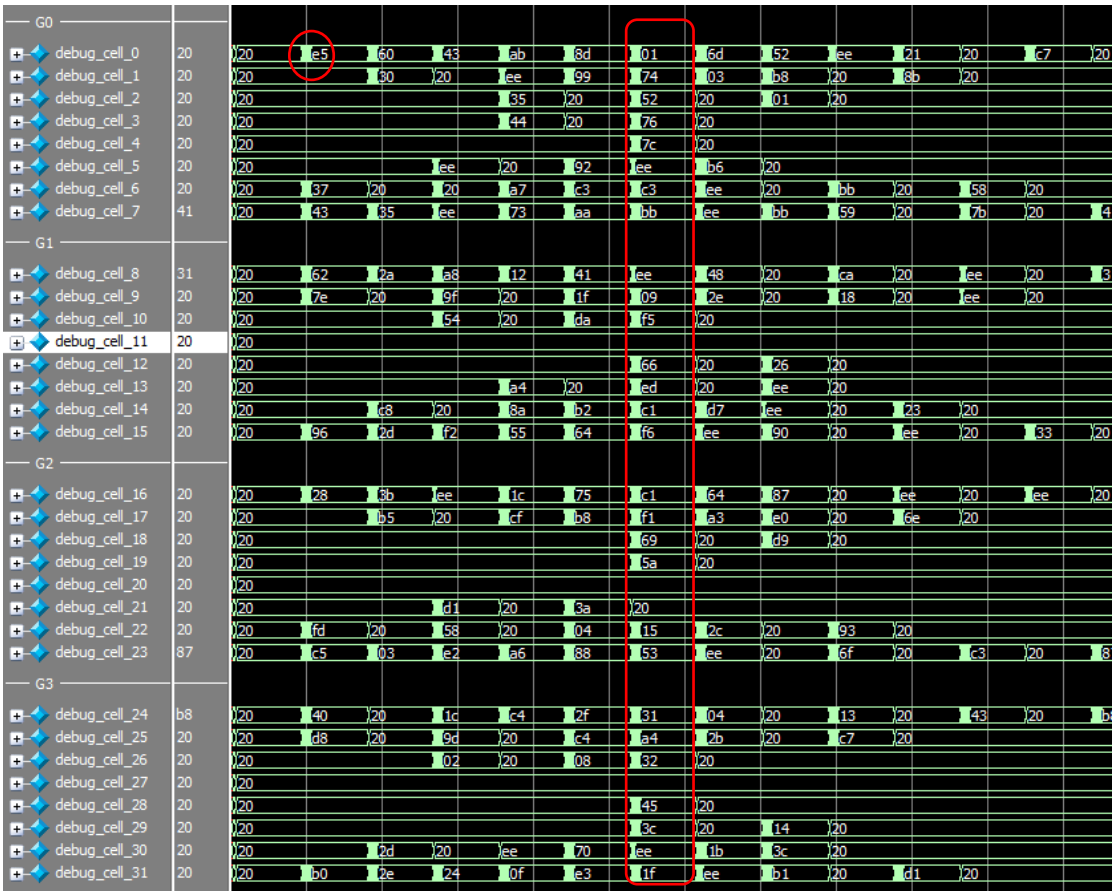


Fig. 10 the simulation waveform of MSSS

3) Assemblage

Assemblage stands in sharp contrast to what's happening in Splitter and Fig. 11 illustrates its complicated design structure. Three state machines FSM1, FSM2, FSM3 are responsible for cell processing with plenty of FIFOs to store temporary data.

For the example of OG2, the assembling of cells is shown as follows.

At the far left part of Fig. 11, cells arrive from 8 lines simultaneously, which turn to serial cells after passing through Cell_buffer. But these cells may come from different source ports and they are very likely to be scrambled after switching. The state machine FSM1 is designed for cell identification and cell classification. By means of the assembling header, we can readily pick out which input line the cell belongs to and then classify the cells into Cell_fifo0 to Cell_fifo15. The most difficult part remains to FSM2 to complete it. We use the same four state machines to assemble every four groups of classified cells into original data. Note that here we have no more need for the packet header and the padding bytes. In the last step, FSM3 outputs the packets with the attached starting label headers from Pkt_fifos, whose situation monitored by Token-bins.

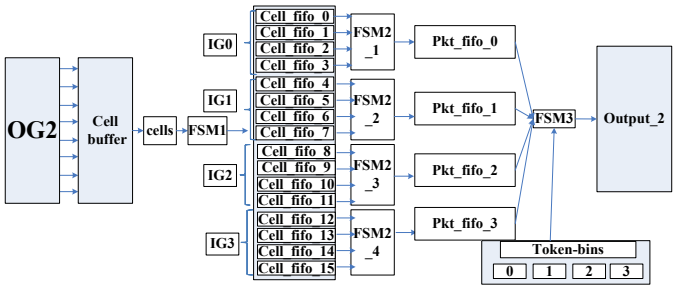


Fig. 11 the design of sub-module Assemblage

As shown in Fig. 12, inputs are serial cells, which can be distinguished by observing signal in_wr and signal in_ctrl. The cell in the circle includes padding bytes, and obviously, it is the last cell of the packet. Based on this information, we can speculate that there is an arriving packet, which consists of seven cells. The output waveform (shown in the rounded rectangle) conforms to the egress rules and proves our judgment.

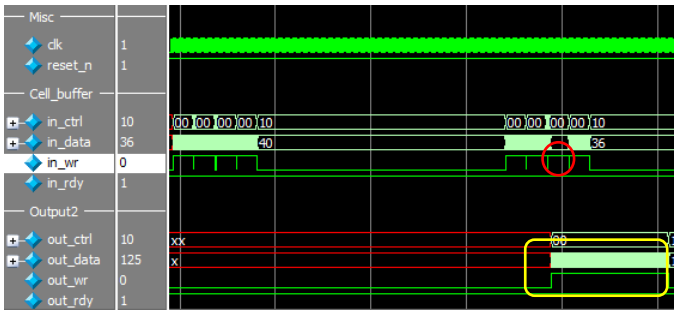


Fig. 12 the simulation waveform of Assemblage

Clearly shown in figure 13, the first byte of the output packet is 8`b0000_0000 and corresponding CTRL signal is 2`b11. Undoubtedly, it is the starting label header.

Fig. 13 the starting label header

D. Design and Analysis of Register System

Another major component of the Load-Balanced Multipath Self-routing Switching Structure (Load-Balanced MSSS) is the register system, which provides an interface to the outside world for the UDP system. In practice, we need a mechanism to control the operations in UDP. And when carrying on back-end design and system debugging, we are looking forward to a window to monitor key signals such as the state of FIFOs, the value of counters and so on. Without register system, we can hardly finish the following work of a big project like ours.

In summary, the register system has two main functions: on the one hand it configures every sub-module in UDP, known as the software register operation; on the other hand it extracts the internal signals from sub-modules in UDP, known as the hardware register operation. The latter is our research focus.

1) the Structure of Register System

Our register system references the pipeline architecture in the NetFPGA program conducted by Stanford University [11]. As shown in Fig. 14, every sub-module in UDP connects with a general register, called Generic_reg. All the general registers and the controller Regs_master form a ring end to end. And then it can interact with the host computer through the Avalon bus devised by Altera. The mutual exchanges of information of the two systems benefit from the IP core JTAG_Avalon_Master_Bridge.

Instead of being connected in a star topology to a central arbiter, the register interfaces of the modules are connected together in a pipeline manner to simplify the process of adding modules. When a new register is inserted, what we need to do is just distributing it the exclusive address space in the pipeline organization while the central arbiter should be modified in the star topology.

Two kinds of registers, software register and hardware register, constitute the generic register Generic_reg which plays a key role. Software register can be written or read by host PC while hardware register can only be read. The interfaces in the system have also two types: one is for the generic registers (signified by the light-coloured arrows in Fig. 14) and another is the interface between a sub-module in UDP and a generic register (signified by the dark arrows in Fig. 14).

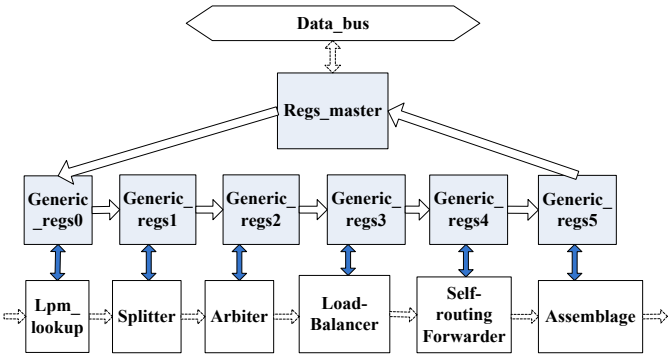


Fig. 14 the whole structure of register system

When the host wants to access a register, it sends an interrogation signal to Regs_master through the Avalon bus. Then the general registers will pass the signal one by one. In this case, the signal will go through all the registers but only one could respond because of the destination information in the signal. A register can only accept the request message belongs to itself by checking the destination address. If valid, it replies immediately and transmits the answers backward until back to host.

Fig. 15 shows the internal details of the two interfaces, one of which is for the registers and it is complicated compared with the other one. According to the official descriptions, the register pipeline is 32-bits wide and runs at 125 MHz. Each module should have two pairs of ports: one for incoming requests and one for outgoing replies. The following set of signals is the input signals for a single module: reg_req_in, reg_ack_in, reg_rd_wr_L_in, reg_addr_in (23-bits), reg_data_in (32-bits), reg_src_in (2-bits). Equivalent signals ending in _out exist for the output port.

Register requests/replies are signified by a high on reg_req_*. reg_req_* should only be high for a single clock cycle otherwise it indicates multiple register access. Note that a module is permitted to take more than one clock cycle to produce a reply but it should ensure that requests following the initial request are not dropped. The reg_rd_wr_L_* signal indicates whether the transaction is a read (high) or a write (low). reg_ack_* should be low when the request is generated and should only be brought high by the module responding to the request.

A module identifies whether it is the target of a request by inspecting the reg_addr_in signal. If the address matches the address range assigned to the module then the module should process the request and generate a response. Once the module has completed any necessary processing it should raise reg_ack_out, set reg_data_out to the correct value in the case of a read, and forward all other inputs to the outputs, all for a single cycle. If a module determines that it is not the target of a request then it should forward all inputs unmodified to the outputs on the next clock cycle.

The reg_src_* signals are used by register request initiators to identify the responses that are destined to the requestor. Each requestor should use a unique value as their source address.

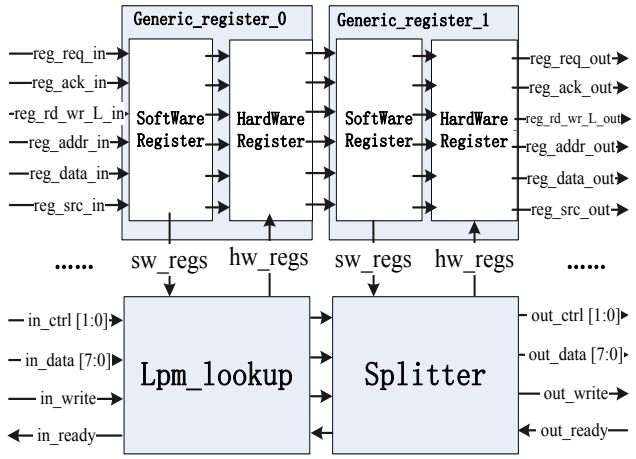


Fig. 15 the two kinds of interfaces in register system

The interface between a generic register and a sub-module in UDP contains two kinds of signals: sw_regs and hw_regs. They undertake responsibility for the two important functions mentioned at the beginning of section III D. In addition, the bit width of sw_regs and hw_regs is set in line with the actual requirement.

2) Software Development Platform for Register System

Software development platform is based on the tool System Console which is used under the environment of Quartus. It provides both debug command line and GUI. Here we use the command line to debug.

On the platform, we mainly perform two tasks with the help of Tcl scripting language. On the one hand, it configures the sub-modules in UDP and the TSE IP core through software registers; on the other hand, it extracts the internal signals of the UDP system to help us debug the system through hardware registers. Every sub-module in UDP has its own debugging interface. Taking the Load-balancer as an example, Fig. 16 shows the pivotal function of our system: load balancing. There are 16384 bytes incoming from IG0. And then, the data is divided into four parts of the same size: 4096 bytes (see the right part of Fig. 16).

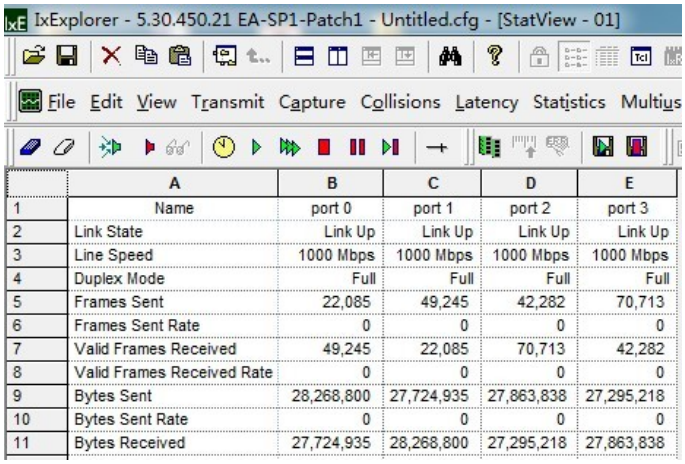
```
** Software Development Platform for Register System**
**************the result of load-balancer **************

   input0(IG0)                    Middle Group(MG)
                              ├----->out_cell_group0: 4096
input_cell_bytes: 16384       ├----->out_cell_group1: 4096
                              ├----->out_cell_group2: 4096
                              ├----->out_cell_group3: 4096
```

Fig. 16 the result of Load-Balancer

IV. SYSTEM TESTING WITH REAL NETWORK TRAFFIC

Having implemented the Load-Balanced MSSS on FPGA, next we should test it with real network traffic.

IXIA 400T network tester is our leading network test instrument. We use four test modules of all the interfaces on the test board, which can generate and capture standard Ethernet frames transmitted at the rate of 10/100/1000 Mb/s. It is so powerful that we can set, if we want to, every byte of a frame to be sent and get detailed and comprehensive information about the frames captured. The tester also provides remote management capabilities. And coupled with the automated platform set up by Tcl scripting language, we can implement remote automated testing.



	A	B	C	D	E
1	Name	port 0	port 1	port 2	port 3
2	Link State	Link Up	Link Up	Link Up	Link Up
3	Line Speed	1000 Mbps	1000 Mbps	1000 Mbps	1000 Mbps
4	Duplex Mode	Full	Full	Full	Full
5	Frames Sent	22,085	49,245	42,282	70,713
6	Frames Sent Rate	0	0	0	0
7	Valid Frames Received	49,245	22,085	70,713	42,282
8	Valid Frames Received Rate	0	0	0	0
9	Bytes Sent	28,268,800	27,724,935	27,863,838	27,295,218
10	Bytes Sent Rate	0	0	0	0
11	Bytes Received	27,724,935	28,268,800	27,295,218	27,863,838

Fig. 17 the statistic views of IXIA

Fig. 17 shows us the final statistical result of a test for four ports. According to our configuration, port 0 and port 1 prepare to receive each other's output data. And port2 and port3 follow the same way. We can see that there is no data dropped at each port in the case of a large number of input data. Next, we are going to conduct more complicated and challenging tests, such as pouring plenty of wrong packets into the system or testing in the case of the maximum throughput.

In order to obtain more realistic test results, we have built a video network platform (VNP) which consists of one server and several clients. On the platform, software VLC media player is installed on all the computers. As shown in Fig. 18, a high-definition movie (1080p) goes though the switching system from the server PC on the left. When the right client PC is receiving data, we cannot feel any delay or unsharpness.

Our two major development and test platforms: the Altera StratixIV FPGA and the IXIA tester are shown in the middle of the picture.

Fig. 18 the video network platform

V. CONCLUSIONS

This paper proposes a new multipath self-routing fabric by merging the Multistage Interconnection Network (MIN) and concentrators. Using the same two MSSS, we construct the Load-Balanced Multipath Self-routing Switching Structure and implement the system model on an Altera StratixIV FPGA. After testing under kinds of network environment, we preliminary confirmed that our system can support QoS applications for Next Generation Network (NGN).

During the process of system implementation, we first devised the overall system and then the two main constituent parts: the UDP system and the register system. When introducing the UDP system, we analysed the design and functions of every sub-module. Based on the cooperative relationship among the sub-modules, we looked through the whole process of data handling in UDP. When presenting the register system, we mainly explained the design of two kinds of interfaces and the software platform.

So far, the scale of our system is still limited (M=4, G=8). And next step, we plan to increase it to M=8, G=16. Meanwhile, the design of large-scale wire-speed multicast base on Load-Balanced MSSS we constructed will be the focus, which needs more excellent design and more thorough support system [12].

ACKNOWLEDGMENT

Our project is supported by National Basic Research Program of China (973 Program) (No.2012CB315904), National Natural Science Foundation of China (No.61179028), Natural Science Foundation of Guangdong Province (No.201101000923), Basic Research of Shenzhen (No.201104210120A). We also acknowledge the valuable feedback from Le Yang (Depaul University) during the preparation of this paper.

REFERENCES

[1] Craig Labovitz, Scott Iekel-Johnson, Danny McPherson, Jon Oberheide, Farnam Jahanian, Internet Inter-Domain Traffic, ACM SIGCOMM 2010;

[2] John Evans, “QoS Decomposed: The Components of the QoS Toolkit”, BRKIPM-2010, Cisco Networkers 2007 Conference;

[3] C. S. Chang, D. S. Lee and Y. S. Jou, “Load Balanced Birkhoff-von Neumann Switches, Part I: One-stage Buffering,” Computer Communications, vol.25 pp.611-622, 2002;

[4] Y. R. Tsai and C. W .Lo, “Banyan-based Architecture for Quasi-circuit Switching”, IEEE ICNS 2006, pp. 23-28;

[5] He W, Li H, Wang B, et al. A Load-Balanced Multipath Self-routing Switching Structure by Concentrators[C]. IEEE ICC 2008;

[6] S. Nojima, et al. "Integrated services packet network using bus matrix switch," IEEEJ. ofSelectAreasCommun.vol. 5, Oct. 1987, pp 1284-1292.;

[7] Li S Y R. Unified algebraic theory of sorting, routing, multicasting, and concentration networks [J]. Communications, IEEE Transactions on. 2010, 58(1): 247-256;

[8] Li S Y R. Algebraic switching theory and broadband applications. Academic Press, 2001;

[9] Hui Li, Wei He, Xi CHEN, Peng Yi, Binqiang Wang, “Multi-path Self-routing Switching Structure by Interconnection of Multistage Sorting Concentrators”, IEEE CHINACOM2007, Aug.2007, Shanghai;

[10] S. Y. R. Li. Algebraic Switching Theory and Broadband Applications. Academic Press, 2001;

[11] Register system - NETFPGA Developers Guide, Available: https://github.com/NetFPGA/netfpga/wiki/DevelopersGuide

[12] Kai Cui, Hui Li, Zhipu Zhu, Fuxin Chen, “Large-scale Wire-speed Multicast Switching Structure Based on Multipath Self-routing Switching Structure and Implemented on FPGA”, ICCIA 2012.