Implementation of Multi-channel FIFO in One BlockRAM with Parallel Access to One Port

Zhipeng Gong[1,2], Tefang Chen[1]
1 School of Information Science and Engineering, Central South University, Changsha, China
2 College of Electrical and Information Engineering, Hunan Institute of engineering, Xiangtan, China
Email:{zpgong2000@126.com, ctfcyt@163.com}

Fumin Zou
School of Information Science and Engineering, Fujian University of Technology, Fuzhou, China
Email:fuminzou@ngi.fj.cn

Li Li, Yingxi Kang
College of Electrical and Information Engineering, Hunan Institute of engineering, Xiangtan, China
Email:{lili@hnie.edu.cn, yxkang@hnie.edu.cn}

***Abstract*—Because of flexibility of application and high cost performance, the low-and-middle-end FPGA has obtained an extensive market. As a fundamental memory structure, the FIFO memory is widely used in FPGA based project in various manners. But limited by the resources in chip and imperfection of development tools, the problem that the number of memory is insufficient while the overall capacity is enough often occurs in the implementation of multi-channel FIFO. This paper surveys various occasions of applications of multi-channel FIFO and put forward a method to achieve multi-channel FIFO in a single FPGA BlockRAM, which would support the parallel access to one port. The method may help to solve the problem mentioned above and improve the utilization of storage resources obviously. The steps of implementation and partial source code are present together with the detail analysis of simulation timing. Practical application indicates that the method is successful and effective.**

***Index Terms*—FPGA, FIFO, BlockRAM, Multi-channel**

I. INTRODUCTION

With the rapid development of FPGA technology, PFGA is widely used in the field of communication, medical instrument, consumer electronics, display, portable terminal, and so on[1][2][3]. Among various levels of FPGA, the low-and-middle-end FPGA has won a wide market due to its low cost, high performance, technical maturity and short design cycle [4]. As a fundamental storage structure in the design of system based on FPGA, FIFO is usually used as data buffer of digital signal processing system, communication bridge between network of different data rates and communication interface between modules in different clock domain [5][6]. Sometimes, in practice, it is needed to combine more than one FIFO into a new structure as multi-channel FIFO, according to rule of access to it, multi-channel FIFO generally can be classified into four categories as follows: 1) Serial Input and Serial Output FIFO (SISOFIFO), each channel of SISOFIFO is independent, no parallel operation at read port or write port is permitted, i.e., the data of the SISOFIFO is written in channel by channel and read out channel by channel. there is no constraint for access among channels; 2) Serial Input and Parallel Output FIFO (SIPOFIFO), just as its name implies, data of the SIPOFIFO is written in channel by channel but read out at the same time; 3) Parallel Input and Serial Output FIFO (PISOFIFO), the data of the PISOFIFO is written in at the same time but read out channel by channel, namely, the write port should be able to be accessed in parallel; 4) Parallel Input and Parallel Output FIFO (PIPOFIFO), the data of the PIPOFIFO is written in simultaneously and read out simultaneously, in other words, both ports of PIPOFIFO should be able to be accessed in parallel.

To implement FIFO in the FPGA based project, FPGA development tools often provide a relevant IP core generator [7]. With this generator, users can easily customize the FIFO step by step. But problem would occur when a FIFO is created by the IP core generator. Because each FIFO created by this means would consume a BlockRAM[8] in FPGA, even if the FIFO is a very small one. That is, once a BlockRAM is used, even a small part of it is used, the whole BlockRAM can't be used by others any more, and thus the remaining memory space is wasted. Generally, BlockRAM is a rare resource in low-and-middle-end FPGA, its number is limited, and it is often organized in uniform size. The capacity of a BlockRAM is always more than that of a FIFO need in many cases, e.g., in the case of multi-channel FIFO, the project need a lot of FIFO in number, but each in small size. Under this circumstance, we have to face the problem that there is enough memory in amount capacity but serious lack in number.

Currently, there are few literatures available for the solution to this problem. M.A.Khan proposed a method to implement a 5-channel SISOFIFO in literatures [9]. In

doi:10.4304/jcp.9.5.1193-1200

order to realize easily, the SISOFIFO is created based on distributed RAM, i.e., based on registers, not BlockRAM. Obviously, the distributed RAM in FPGA is used to realize the all kinds of logic on original purpose, and the amount of it is limited too. So this method is not suitable for the situation when there is a large amount of data to store. At the same time, this method can't make use of BlockRAM, so it would waste the memory resource.

With experience in previous FPGA projects, we have realized the design of PIPOFIFO of multi-channel FIFO and successfully applied it to data buffer of bus transceiver of Multifunction Vehicle Bus (MVB) and data cache of large LED display screen. The structure of PIPOFIFO is depicted in Figure 1. To realize the PIPOFIFO, a simple DPRAM is instantiated from a BlockRAM at first, its memory space is divided into multiple parts according to the number of the channels of PIPOFIFO (4 channels in Figure 1). The input data of all channels, from din_0 to din_3, are written in simultaneously under the uniform signal wr_enx, and the output data of all channels, from dout_0 to dout_3, are read out simultaneously under the uniform signal rd_enx. The label logics of all the channels are the same, so one set of them is enough. Actually the PIOPFIFO can be simply viewed as a binding of multiple normal FIFOs and act as one normal FIFO.

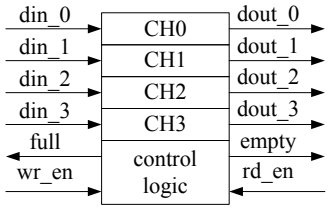

Figure 1. Structure of PIPOFIFO

In this paper, we propose a new solution for the remaining three kinds of multi-channel FIFO to implement them in one BlockRAM. As to SISOFIFO, the channels are independent and there are no timing constraints among them, so it is relatively easy to achieve. In the coming sections, we would concentrate our attention on the implementation of SIPOFIFO and PISIFIFO in one BlockRAM, it can significantly improve utilization of BlockRAM, reduce the cost of product and help to enhance market competitiveness.

II. Design And Structure

A. Basic Structure

From the analysis above, we have to implement multi-channel FIFO in one BlockRAM and provide write or read operation in parallel to some extent. It's obviously impossible to realize write or read operation at the same time for multiple FIFOs that built in one BlockRAM directly. In order to solve the problem of parallel access, it is necessary to place registers into the ports with function of parallel access as data buffer. The width of data buffer should be set in accord with the width of corresponding channel, and the depth is decided by the level of parallel access operation. The general structure of multi-channel FIFO is depicted in Figure 2. It is mainly composed of write control logic, DPRAM, read control logic and input/output buffering registers. The buffering registers are located at the parallel port, i.e., write port for PISOFIFO, and read port for SIPOFIFO. DPRAM can be instantiated from BlockRAM with IP core generator. Simple DPRAM is enough here because there is only one write port and so half of memory capacity can be saved by this means. Each channel have its own private memory space in the DPRAM, all the private memory spaces can't be overlapped. The read control logic and write control logic are relatively complex and there are differences in their structure between the scenes of serial access and parallel access, we'll describe this in detail later.

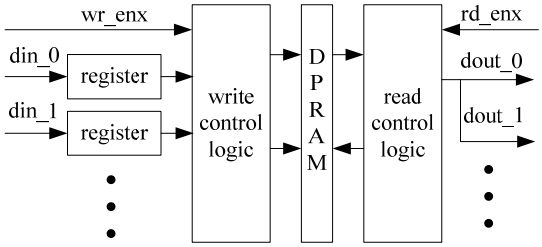

Figure 2. General structure of multi-channel FIFO

B. Parallel Write Control Logic

Parallel write control logic is designed to receive input data of all channels and write them into the DPRAM at corresponding area. The structure of write control logic in parallel is depicted in Figure 3. After receive a parallel write command, i.e. when wr_enx is active, the input data of all channels are registered. On detecting of data coming, the internal write control logic would activate the internal write signals for every channel in turn to fetch the data from registers and write to the DPRAM at corresponding memory area. Here is how it works:

Step 1. The control logic enters the idle state when system resets, all the labels are initialized at this time, e.g. p_write_ready, the label of write in parallel getting ready, is set to 1.

Step 2. When wr_enx is active, internal control logic accepts the input data of all channels and then stores them in corresponding buffering registers. p_write_ready is set to 0. A new external write command would not be accepted before p_write_ready is set to 1 again.

Step 3. Set write_in_process to 1, which means the control logic is enter the procedure of carrying data from buffering registers to DPRAM.

Step 4. Internal control logic generates the internal write command vector wr_eni, this causes a internal write operation for each channel, take channel 0 for example, the internal write operation goes as follows:

Step 5. Number of write channel is decided by ch_write, it is set to 0 firstly, i.e., the first channel is selected;

Step6. The write address of DPRAM wr_addr is a combination of ch_write and write_p_0, i.e., wr_addr={ch_write,write_p_0}, where write_p_0 means the write address pointer of channel 0.

Step 7. The LSB of the internal write commands vector wr_eni is set to 1, and the input data of channel 0 is written to corresponding memory area in DPRAM ;

Step 8. Write address pointer write_p_0 is increased by 1;

Step 9. Write channel number ch_write is increased by 1, so the next channel is selected;
Step 10. The remaining channels write data to DPRAM in the same way as channel 0 in accordance with the operation sequence from **step 5** to **step9** above;
Step 11. After the whole parallel write operation is completed, and if the FIFO is not full, p_write_ready is set to 1 to show the write control logic is ready for the next write operation;

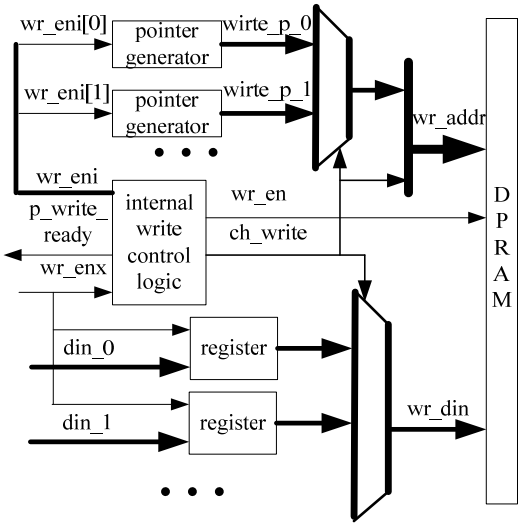

Figure 3. Structure of parallel write control logic

C. Serial Write Control Logic

The structure of serial write logic is depicted in Figure 4, the way it works is exactly the same as multiple independent FIFOs to write separately, it's not necessary to place buffering registers here that used in parallel case, and the labels for parallel write, e.g., p_write_ready, are not needed either. Each channel sets the full label ch_full according to the general rules of common FIFO as an independent FIFO. The write operation of each channel is carried out directly under the extern serial write command wr_enx. The number of write channel is produced by coding wr_enx. It is unnecessary to generate extra internal write command wr_eni, i.e., let wr_eni= wr_enx. Here is how it works:
Step 1. When system resets, the full label of each channel is set to 1, which means no channel is full. Each channel carry out the write operation independently, take channel 0 for example, it works as follows:
Step 2. When external write command wr_enx=0001, the write channel ch_write is coded as 0, i.e., the first channel is selected.
Step 3. The write address of DPRAM wr_addr is combination of ch_write and write_p_0, i.e., wr_addr = {ch_write,write_p_0}, where write_p_0 means the write address pointer of channel 0.
Step 4. The input data of channel 0 is written to corresponding memory area in DPRAM.
Step 5. Write address pointer write_p_0 is increased by 1.
Step 6. When another channel is selected, it works in the same way as channel 0 in accordance with the operation sequence from **step 2** to **step 5** above.

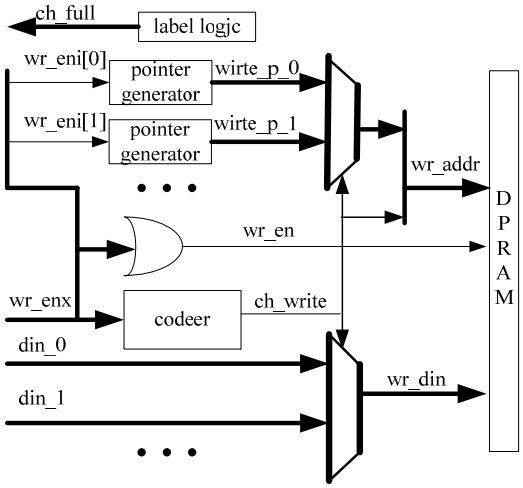

Figure 4. Structure of serial write logic

D. Parallel Read Control Logic

Parallel read control logic is designed to fetch data from proper area in DPRAM and sent them to the output port of corresponding channel. The structure of parallel read control logic is shown in Figure 5. When output buffering registers are free and every channel is not empty, the internal read control logic would give the read command to every channel in turn, thus, the data are read from the DPRAM and registered, and then the label of parallel read data getting ready, p_read_ready, is set to 1. When a active read request, rd_enx, is detected, parallel read control logic will output all the buffered data simultaneously. The following steps would show how it works:
Step 1. When system resets, the label of parallel read getting ready is cleared, i.e., p_read_ready is set to 0;
Step 2. When none of channel is empty, internal read control logic would begin to deliver the data of every channels from DPRAM to corresponding registers as follows;
Step 3. The read channel number ch_read is set to 0, i.e., the first channel is selected.
Step 4. The read address of channel 0 in DPRAM, rd_addr, is a combination of ch_read and read_p_0, i.e., rd_addr = {ch_read,read_p_0}, where read_p_0 means the read address pointer of channel 0;
Step 5. The data of channel 0 would be on the read bus and saved to the output buffering registers when the latch command of channel 0 is active.
Step 6. The LSB of internal read command vector rd_eni is set to 1, and read the address pointer of channel 0, read_p_0, is increased by 1;
Step 7. ch_read is increased by 1, i.e., the next channel is selected;
Setp 8. when another channel is selected, it works in the same way as channel 0 in accordance with the operation sequence from **step 4** to **step 7** above.
Step 9. The label of parallel read getting ready, p_read_ready, is set to 1;
Step 10. When external read request, rd_enx, is detected active, the data of all channels in buffering register will be output simultaneously;
Step 11. p_read_ready is set to 0;

Step 12. Turn to step2.

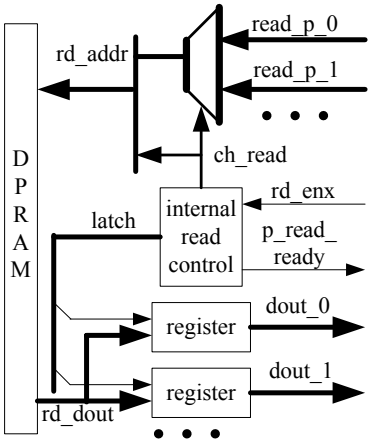

Figure 5. Structure of parallel read control logic

E. Serial Read Control Logic

Serial read control logic is designed to read data from multi-channel FIFO channel by channel. No output buffering registers are needed at this moment, and neither do the labels related to parallel read control logic. It is unnecessary to generate extra internal read command rd_eni any longer, external read command can be used as internal read command directly, i.e., let rd_eni= rd_enx. Each channel set the empty label ch_empty according to the general rules of common FIFO as an independent FIFO. The read operation of each channel is carried out at the external read requests directly. The code of read channel, ch_read, is decided by external read command rd_enx. How it works is similar to that of serial write logic.

F. Label Logic

Label logic is designed to generate all kinds of labels for multi-channel FIFO, including general labels as empty, full, and special labels for parallel access. The structure of label logic is shown in Figure 6. General labels can be produced by common rules of FIFO. Here are rules for special labels:

1) For PISOFIFO, p_write_ready is set to 1 when the input registers are empty and no channel of the PISOFIFO is full. Once the buffering data begin to write to DPRAM, p_write_ready is set to 0.

2) For SIPOFIFO, p_read_ready is set to 1 when data in registers of all channels are ready. Once an external read commands is detected active, p_read_ready is set to 0.

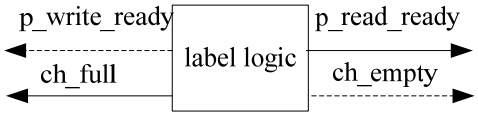

Figure 6. Structure of label logic

III. IMPLEMENTATION

A. Objedtive Platform

We constructed a XILINX FPGA development platform based on XC3S400A, which comes from the SPARTAN3A family, to implement the multi-channel FIFO. ISE13.2 XST [10] is employed as the synthesis tool. A 4-channel SIPOFIFO and a 4-channel PISOFIFO are realized on this platform. A 512 x 25 DPRAM is instantiated from a BlockRAM of 18Kb in size. The assignment of memory space to the 4 channels of FIFO is depicted in Figure 7.

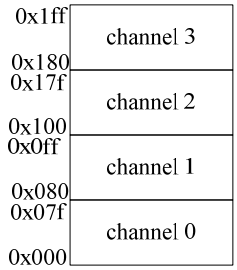

Figure 7. Assignment of memory space

We employ the Verilog Hardware Description Language (Verilog HDL) to implement SIPOFIFO and PISOFIFO, and part of relevant source code is presented in appendix A

B. Synthesis Result

The synthesis result of PISOFIFO and SIPOFIFO is listed in TABLE 1, including the resources they used and the maximum work frequency they can reach. From table 1, we can learn that each multi-channel FIFO needs only one BlockRAM and a few additional logic resources, the maximum frequency is up to 200Mhz. these features would enable the multi-channel FIFO to be integrated in most of projects based on FPGA and work well.

TABLE 1.
SYNTHESIS RESULT

FIFO type	Slices	LUTs	RAMB16BWEs	Frequency
PISOFIFO	226	271	1	193
SIPOFIFO	234	282	1	208

IV. SIMULATION AND EXPERIMENT

We employ modelsim6.5d [11] as the simulation tool and select XC3S400A as the objective chip, which comes from XILINX SPARTAN3A family of low-cost. The test bench runs on a personal computer with a core I3 2.3GHz processor and 2 GB of random access memory.

A. Simulation Of SIPOFIFO

As described earlier, a DPRAM of 512 words is divided into four parts, i.e., there are 128 words per part. Each channel takes up one part as shown in Figure 7. In test bench, we write 64 data, from 0x000000 to 0x00003f, to each channel in turn. The timing diagram of serial write is shown in Figure 8. For further explanation, we take the write operation of channel 0 for example. The data input to din_0 ranges from 0x000000 to 0x00003f, increased by 1 at a time. The write address of DPRAM, wr_addr, is composed of write channel number and write address pointer, i.e., wr_addr = {ch_ write, write_p_0}. As for channel 0, ch_write = 0. write_p_0 ranges from 0x00 to 0x3f. So the write address of DPRAM, wr_addr, ranges from 0x000 to 0x03f. The serial write operation of other channels is similar to that of channel 0.

After the write operation of all channels, the input data should be stored in DPRAM at proper area. The storage of the data of channel 0 and channel 1 in DPRAM is shown in Figure 9. According to the design, the memory space of channel 0 ranges form 0x000 to 0x07f and the memory space of channel 1 ranges form 0x080 to 0x0ff. The input data is the same for all channels, ranging from 0x000000 to 0x00003f. From Figure 9, we can discover that the storage of the data is consistent with the design and test bench.

The timing diagram of parallel read is shown in Figure 10. When the output data of every channel has been registered already, p_read_ready is activated to declare that the multi-channel FIFO is ready for parallel read operation. On receiving the external read request, i.e., when rd_enx is detected active, the parallel read control logic will output the data of all channels at the same time. Then the data is kept in the registers of test bench, from dout_reg_0 to dout_reg_3. Obviously, form Figure 10, we can learn that the data of all channels are the same as expected. It means that the parallel read operation is carried out successfully. p_read_ready is deactivated after a read operation, the parallel read control logic begins to read data form DPRAM to registers for the next read operation. For further description, we take channel 0 for example to illustrate the procedure of read operation from DPRAM to registers. When the output buffering registers are empty, it's time to prepare data for output again. Channel 0 is selected firstly to read data from its memory area in DPRAM to its registers. the read channel number, ch_read is set to 0, The read address of DPRAM, rd_addr, is composed of read channel number and read address pointer, i.e., rd_addr = {ch_ read, read_p_0}. When write_p_0=0x01, rd_addr must be 0x001, the data in DPRAM at 0x001would appear on the read bus and then would be latched in corresponding regisers as shown in Figure 10. The read operation of the remaining channels is similar to that of channel 0 and their corresponding addresses in DPRAM are 0x081, 0x101and 0x181 respectively.

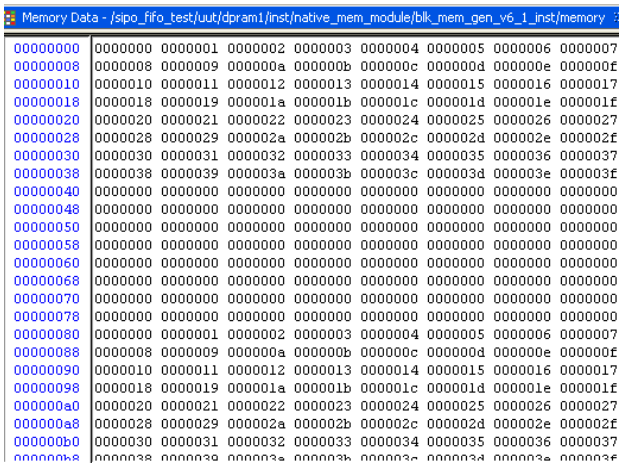

Figure 9. Storage of data in DPRAM

B. Simulation Of PISOFIFO

The timing diagram of parallel write is shown in Figure 11. When the input buffering registers are all empty, p_write_ready is activated to declare that the multi-channel FIFO is ready for parallel write operation. On receiving the external write request, i.e., when wr_enx is detected active, the parallel write control logic will accept the input data of all channels and keep them in the buffering registers simultaneously. During the time that followed, p_write_ready is deactivated and the parallel write control logic begins to write data to DPRAM form registers. For further description, we take channel 0 for example to illustrate the procedure of write operation to DPRAM from registers. When channel 0 is selected firstly to write data from its registers to its memory area in DPRAM, the write channel number, ch_write is set to 0. The write address of DPRAM, wr_addr, is composed of write channel number and write address pointer, i.e., wr_addr = {ch_ write, write _p_0}. When write _p_0=0x01, wr_addr must be 0x001, the data would be written into DPRAM at 0x001 when write enable signal, wr_eni[0], is set to 1, as shown in Figure 11. The write operation of the remaining channels is similar to that of channel 0 and their corresponding addresses in DPRAM are 0x081, 0x101and 0x181 respectively.

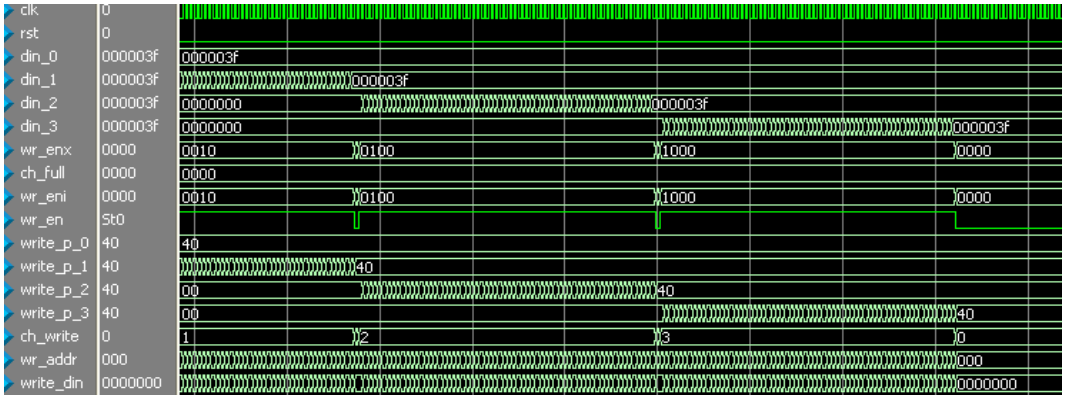

Figure 8. Timing diagram of serial write

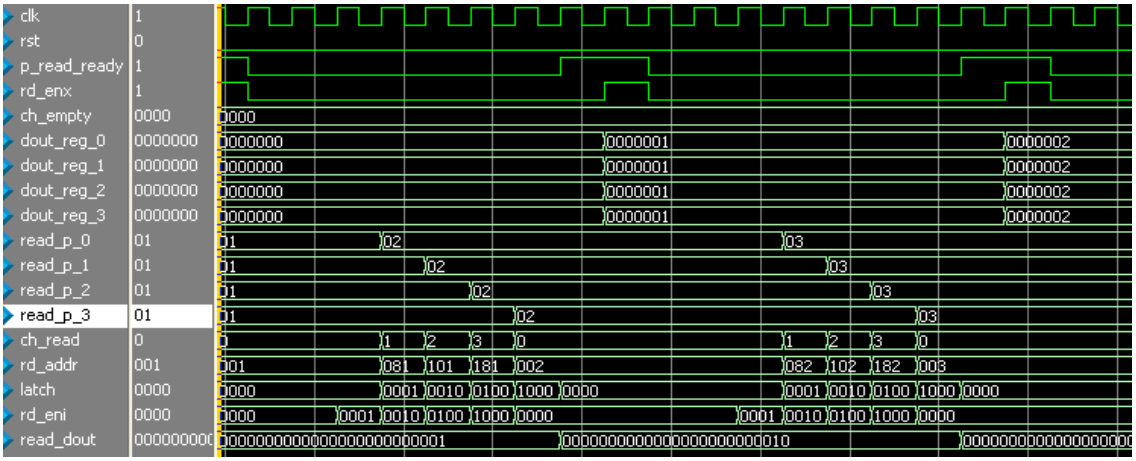

Figure 10. Timing diagram of parallel read

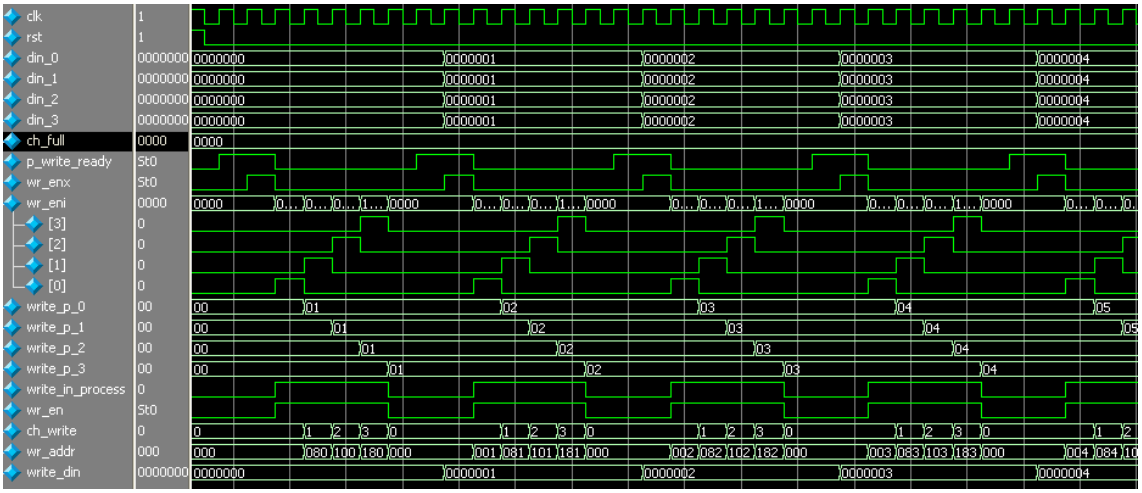

Figure 11. Timing diagram of parallel write

C. Conclusion Of Simulation

Through the simulation and analysis above, we can found that the timing waveform is consistent with the previous design of multi-channel FIFO, which is implemented with one BlockRAM and qualified to provide the parallel write or parallel read.

V. APPLICATION

In practice, the multi-channel FIFO is integrated in a FlexRay [12] communication controller (FCC) based on FPGA. In this project, we select the XC3S400A as the main chip. Its on-chip resources include 400k logic gates, 3584 logic slices, 56Kb distributed RAM, 360Kb BlockRAM, 4 DCM, and so on[13], the total amount of resources can meet the requirement of project. Analysis indicates that there are at least 32 FIFOs needed in the project. 16 of them are used as input data buffer with the structure of parallel write and serial read, the others are used as output data buffer with the structure of serial write and parallel read. If every FIFO is created by the IP core generator directly, these buffers would cost 32 pieces of BlockRAMs. But there are only 20 BlockRAMs in XC3S400A, and then resource of BlockRAM will be seriously insufficient. If we use XC3S1400A instead as the main chip, this situation would be relaxed slightly, because there are 32 BlockRAM in XC3S1400A, it is just enough for the data buffer. But the BlockRAM may be used somewhere else and it would be insufficient again. What's more, the substitute is much more expensive. Further analysis shows BlockRAMs in XC3S400A is perfectly adequate in total capacity but insufficient in number. We decide to employ the method proposed above to realize multi-channel FIFO in one BlockRAM. Study of project shows that the input port has a big data flow while the output port has a relatively less data flow. Hence, the capacity of FIFO of input port should be larger than that of the output port. At last, we decide to implement a 2-channel PISOFIFO in one BlockRAM for input port, each channel is 256 x 25 in size, and 4-channel SIPOFIFO in one BlockRAM for output port, and each channel is 128 x 25 in size. The structure of buffer for FlexRay communication controller is depicted in Figure 12. At last, 12 pieces of BlockRAMs is used for buffers

and 8 pieces of BlockRAM is remained for other purpose. The debug result shows that the system works in accordance with the design as expected.

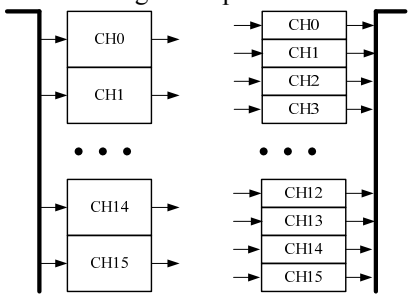

Figure 12. Structure of buffer for FCC

VI. CONCLUSION

This paper introduces a method of implementation of multi-channel FIFO with the utility of parallel access in a single piece of BlockRAM. This method can help to make full use of limited BlockRAM resources in FPGA and reduce the cost of terminal product, and it shows some practical value in project base on low-cost FPGA design. The simulation test and practical application indicate that this method is correct. The multi-channel FIFO created with it can run at a high frequency and is suitable for integration in most FPGA based projects.

APPENDIX A IMPLEMENTTATION WITH VERILOG HDL

A.1 Implementtation Of SIPOFIFO

```
module sipo_fifo(
      … //description of signals
          );
//set internal read command
  always @ (posedge clk or posedge rst)
      if(rst) rd_en_inter_set<=1'b0;
      else if(rd_en_inter_set|rd_command_time)
     rd_en_inter_set<=1'b0;
      //no channel is empty and registers are free
      else if(~|ch_empty&~|reg_label)
rd_en_inter_set<=1'b1;
//labels for buffering registers.
always @ (posedge clk or posedge rst)
      if(rst) reg_label[0]<=1'b0;
      else if(rd_en_inter_shift[3])reg_label[0]<=1'b1;
      else if(rd_enx)reg_label[0]<=1'b0;
…
//latch data to registers.
always @ (posedge clk or posedge rst)
      if(rst)dout_0<=25'h0;
      else if(latch[0]) dout_0<=read_dout;
//generate the write address
wire[3:0] wr_eni=wr_enx;
always @( * )
      case(wr_eni)
            4'b0001: ch_write=2'b00;
            4'b0010: ch_write=2'b01;
            4'b0100: ch_write=2'b10;
            4'b1000: ch_write=2'b11;
      endcase
assign wr_addr={ch_write,
    ({7{wr_eni[0]}}& write_p_0[6:0])|
    ({7{wr_eni[1]}}& write_p_1[6:0])|
    ({7{wr_eni[2]}}& write_p_2[6:0])|
  ({7{wr_eni[3]}}& write_p_3[6:0])};
wire wr_en=|wr_eni;
//select the data to write to DPRAM
assign   write_din= ({25{wr_eni[0]}}& din_0)|
                    ({25{wr_eni[1]}}& din_1)|
                    ({25{wr_eni[2]}}& din_2)|
                    ({25{wr_eni[3]}}& din_3);
//label logic
always @ (posedge clk or posedge rst)
      if(rst) p_read_ready<=1'b0;
      else  p_read_ready<=&reg_label&~rd_enx;
// instance of DPRAM
dpram25_512 dpram1 (
  .clka(clk),
  .wea(wr_en),
  .addra(wr_addr),
  .dina(write_din),
  .clkb(clk),
  .rstb(rst),
  .addrb(rd_addr),
  .doutb(read_dout) );
…
endmodule
```

A.2 Implementtation Of PISOFIFO

```
module piso_fifo(
…//description of signals
      );
wire[3:0] rd_eni=rd_enx;
always @ (posedge clk or posedge rst)
      if(rst) p_write_ready<=1'b0;
      else  p_write_ready<=
       ~wr_enx&~write_in_process&~|ch_full;
        //not full, last write operation is over,
always @ (posedge clk or posedge rst)
      if(rst) write_in_process<=1'b0;
      else if(wr_enx&~write_in_process&~&ch_full)
            write_in_process<=1'b1;
      else if(wr_eni[3])write_in_process<=1'b0;
//read buffering registers
always @ (posedge clk or posedge rst)
      if(rst)
            begin
              din_reg_0<=25'b0;
              din_reg_1<=25'b0;
              din_reg_2<=25'b0;
              din_reg_3<=25'b0;
            end
      else if(wr_enx&p_write_ready)
            begin
                  din_reg_0<=din_0;
                  din_reg_1<=din_1;
                  din_reg_2<=din_2;
                  din_reg_3<=din_3;
            end
always @ (posedge clk or posedge rst)
```

```
        if(rst) wr_eni<=4'b0;
        else wr_eni<={wr_eni[2:0],wr_enx};
//write to DPRAM
assign wr_en=|wr_eni;
always @( * )
        case(wr_eni)
                4'b0001: write_din=din_reg_0;
                4'b0010: write_din=din_reg_1;
                4'b0100: write_din=din_reg_2;
                4'b1000: write_din=din_reg_3;
        endcase
//instance of DPRAM
dpram25_512 dpram1 (
    .clka(clk),
    .wea(wr_en),
    .addra(wr_addr),
    .dina(write_din),
    .clkb(clk),
    .rstb(rst),
    .addrb(rd_addr),
    .doutb(read_dout) );
…
endmodule
```

ACKNOWLEDGEMENT

The work in this paper is supported by The National Natural Science Foundation of China (60674003), Major projects of Fujian Province (2011HZ0002-1), Scientific Research Fund of Hunan Provincial Education Department (12C0638) , Research Fund of Hunan Institute of Engineering (2013GK3034), Provincial Natural Science Foundation of Hunan(13JJ9022) and Provincial Science & Technology plan project of Hunan(2013GK3029).

The authors also would like to thank the key laboratory for automotive electronics and electric drive of Fujian province for the support in testing work she provided.

REFERENCE

[1] Z.Lu, Y.Wu, "A Rotation-based Data Buffering Architecture for Convolution Filtering in a Field Programmable Gate Array", Journal of Computers, 8(6): pp.1411-1416, 2013.

[2] S.Kasap and K.Benkrid, "Parallel Processor Design and Implementation for Molecular Dynamics Simulations on a FPGA-Based Supercomputer", Journal of Computers, 7(6): pp.1312-1328, 2012.

[3] A.M.Fernandes, "HDL Based FPGA Interface Library for Data Acquisition and Multipurpose Real Time Algorithms", IEEE Transactions on Nuclear Science, vol.58, pp. 1526-1530, 2011.

[4] A.Afaneh, Y.He, "Implementation of accurate frame interleaved sampling in a low cost FPGA-based data acquisition system", International Conference on Intelligent Data Acquisition and Advanced Computing Systems, IEEE, pp.20-25, Sept. 2011.

[5] G..W.Zhong; H.B.Zheng, et al., "1024-point pipeline FFT processor with pointer FIFOs based on FPGA", International Conference on VLSI and System-on-Chip, IEEE, pp.122-125, Oct. 2011.

[6] H.S.Han, K.S.Stevens, "Clocked and asynchronous FIFO characterization and comparison", IEEE International Conference on Very Large Scale Integration Oct, IEEE, pp.101-108, 2009

[7] Xilinx Inc, LogiCORE IP FIFO Generator v8.2, http://www.xilinx.com, 2011.

[8] Xilinx Inc, LogiCORE IP Block Memory Generator v6.2, http://www.xilinx.com, 2011

[9] M.A.Khan, A.Q.Ansari, "n-Bit multiple read and write FIFO memory model for network-on-chip", World Congress on Information and Communication Technologies, IEEE, pp.1322-1327, Dec. 2011.

[10] Xilinx Inc, ISE Design Suite Software Manuals, http://www.xilinx.com, 2011.

[11] Mentor Graphics Corporation, ModelSim SE User's Manual, http://www.mentor.com, 2009.

[12] FlexRay consortium, FlexRay communication systems protocol specification, version 3.0.1. http: //www.flexray.com, 2010.

[13]Xilinx Inc, Spartan-3A FPGA Family, http://www.xilinx.com, 2009.

Zhipeng Gong Received the B.S. in physics from the Hunan Normal University, Changsha, China, in 1998, and the M.S. in transport information engineering and control from Central South University, Changsha, China in 2005. He is currently working toward the Ph.D. degree with the School of Information Science and Engineering, Central South University.

He is also currently a faculty member with the College of Electrical and Information Engineering, Hunan Institute of engineering, Xiangtan, China. His research interests include embedded system, FPGA and communication network in vehicle.