ISSN (Online) 2394-2320

International Journal of Engineering Research in Computer Science and Engineering (IJERCSE)

Vol 4, Issue 12, December 2017

Synthesis of Sliding Window Protocol with Piggybacking

[1] Neha Jain, [2] Dr. Manoj Kumar Jain

[1][2] Department of Computer Science, Mohanlal Sukhadiya University, Udaipur, India

***Abstract:*-- Data traffic on communication channel is increasing day by day. To increase the utilization of bandwidth of the communication channel we implement sliding window protocol with the concept of piggybacking on hardware. Much work has been done in this field but in this paper we implement the concept of sliding window with piggybacking. To implement the algorithm on hardware we use a hardware description language like VHDL. We implemented it on Xilinx ISE. Total memory usage of this implementation is 267108 kilobytes.**

Keywords— Piggybacking, Sliding Window Protocol, VHDL, Xilinx ISE

INTRODUCTION

Sliding window protocol is a data transmission protocol. It works on data link layer of OSI model. In today's scenario bandwidth of communication channel is very high. Sliding window protocol sends more than one frame at a time. Thus this protocol utilizes the bandwidth of the communication channel. In this paper to increase the utilization of the bandwidth we implement the Sliding Window protocol with the concept of Piggybacking. Piggybacking means when if the receiver has data to send, it will send this data with acknowledgement. Thus by sending more information in a frame we can utilize bandwidth of the channel more efficiently. We implement this concept using Xilinx ISE.

In [1] author presented a software solution of dynamically sliding window protocol for data synchronization in a flow cytometer. In [2] author proposed a hardware for computing modular exponentiat1ion using the sliding window method. In [3] author proved the protocol against a service description by using boolean logic. In [4] author presented an automated tool, Sliding Window Operation

Optimization (SWOOP) that generates the estimate of speedup for a high performance design. Author determined the speedup by the area of the FPGA. In [5] author studied the effect of concurrency in network processors on packet ordering. In [6] author analyzed the network traffic using the network traffic monitor and he investigated the Internet traffic characteristics through a statistical analysis. In [7] a network processor model was introduced which was used as a basis for a simulation tool. In [8] author explained the role of network processors in active networks.

In this paper we have implemented sliding window protocol with piggybacking on hardware. Section 1 shows the hardware implementation of the algorithm. Section 2 shows the results and section 3 shows the conclusion.

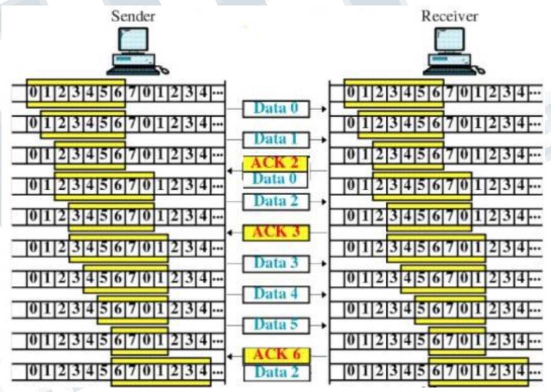

Fig. 1 Sliding Window Protocol with Piggybacking

1. Implementation of Code in Xilinx ISE

A. Piggybacking.v

This is the code that sends/receives the data from sender/receiver. We have assumed that the protocol uses a 16 bit data packet at a time for data transfer. Number of frames used for this data transfer is 8. Hence there are 2 bits of data in each frame getting updated at every active edge of clock cycle.

We have inputs such as sender_availability and receiver_availability to check if data is available to be sent at Sender/Receiver. We have implemented the process of data transfer in data_framing.v code.

Once data transfer is completed, then this protocol waits for a few clock cycles (3 clock cycles) to check if data is available with sender/receiver or not. If the data is available at receiver, then receiver will send data + ACK to the sender. Here ACK is called final_ack_data_ready as data was available. In case if data is not available after 3 clock cycles, then our ACK will be in terms of final_ack_data_NOT_ready. Same mechanism is also incorporated for reverse communication.

```
module piggybacking(sender_data,ack,
final_ack_data_ready, final_ack_data_NOT_ready,
receiver_data, receiver_data_temp, clock, reset,
sender_availability, receiver_availability, ack);
input clock, reset;
input [D-1 : 0] sender_data;
input sender_availability, receiver_availability;
output reg [D-1 : 0] receiver_data;
output [D-1 : 0] receiver_data_temp;
output ack;
output reg final_ack_data_ready;
output reg final_ack_data_NOT_ready;
reg [15:0] data_from_sender_to_receiver,
data_from_receiver_to_sender;
parameter N = 2, D = 16;
wire [1:0]
frame1,frame2,frame3,frame4,frame5,frame6,frame7,fra
me8;
reg data_receiver_at_sender, data_received_at_receiver;
wire [15:0] receiver_data_temp_reg;
wire [15: 0] dout;
reg ack_sender, ack_receiver;
wire temp_ack;
reg ack_by_sender, ack_by_receiver;
assign receiver_data_temp_reg = receiver_data_temp;
assign ack = temp_ack;
data_framing d1 (.data_in(sender_data),
.data_out(receiver_data_temp),
.clock(clock),
.reset(reset),
.sender_availability(sender_availability),
.receiver_availability(receiver_availability),
.frame1(frame1), .frame2(frame2),
.frame3(frame3), .frame4(frame4),
.frame5(frame5), .frame6(frame6),
.frame7(frame7), .frame8(frame8),
.temp_ack(temp_ack));
always @ (posedge clock)
begin
if (reset)
data_from_sender_to_receiver <= 0;
else if
(sender_availability == 1 && receiver_availability == 0)
data_from_sender_to_receiver <= dout;
else if (sender_availability == 0 && receiver_availability
== 1)
data_from_receiver_to_sender <= dout;
end
always @ (posedge clock)
begin
if (reset)
begin
ack_by_receiver <= 0;
end
else if (temp_ack == 1)
begin
@ (posedge clock); @ (posedge clock);
@ (posedge clock); @ (posedge clock);
@ (posedge clock); @ (posedge clock);
@ (posedge clock); @ (posedge clock);
@ (posedge clock); @ (posedge clock);
@ (posedge clock); @ (posedge clock);
if (receiver_availability==1)
begin
ack_by_receiver <= 1'b1;
@ (posedge clock);
ack_by_receiver <= 0;
end
else
begin
ack_by_receiver <= 1'b1;
@ (posedge clock);
ack_by_receiver <= 0;
end
end
end
always @ (posedge clock)
begin
if (reset)
begin
ack_by_sender <= 0;
end
else if (temp_ack == 1)
begin
@ (posedge clock); @ (posedge clock);
@ (posedge clock); @ (posedge clock);
@ (posedge clock); @ (posedge clock);
@ (posedge clock); @ (posedge clock);
@ (posedge clock); @ (posedge clock);
@ (posedge clock); @ (posedge clock);
if (sender_availability==1)
begin
ack_by_sender <= 1'b1;
@ (posedge clock);
ack_by_sender <= 0;
end
else
begin
ack_by_sender <= 1'b1;
@ (posedge clock);
ack_by_sender <= 0;
end
end
end
always @ (posedge clock)
begin
if (reset)
begin
final_ack_data_ready <= 0;
final_ack_data_NOT_ready <= 0;
end
else if (temp_ack)
begin
@ (posedge clock); @ (posedge clock);
@ (posedge clock); @ (posedge clock);
@ (posedge clock); if (sender_availability == 1 ||
receiver_availability == 1)
begin
final_ack_data_ready <= 1;
@ (posedge clock);
final_ack_data_ready <= 0;
end
else
begin
final_ack_data_NOT_ready <= 1;
@ (posedge clock);
final_ack_data_NOT_ready <= 0;
end end
else
final_ack_data_ready <= 0;
final_ack_data_NOT_ready <= 0;
end
endmodule
```

B. dataframing.v

This code decides how the data transfer takes place via data frames. In our protocol, we have assumed that there are data packets each of 16 bit and data frames of 2 bits. Hence for a data transfer to take place, 8 clock cycles are required. Once data transfer is complete, we assert a temp_ack signal to establish that data transfer has been completed. Then protocol waits for a while to decide whether to assert final_ack_data_NOT_ready(receiver_availability = 0) or final_ack_data_ready (receiver_availability = 1). These signals are then used in Piggybacking.v to give final outputs.

```
module data_framing (data_in, temp_ack, data_out,
receiver_availability, clock,
reset,sender_availability,frame1,frame2,frame3,frame4,frame5,frame6,frame7,frame8);
parameter D= 16, N=2;
input[D-1 : 0] data_in;
input clock, reset;
output [15 : 0] data_out;
reg [15 : 0] data_out_temp;
integer i = 16 ;
input sender_availability,receiver_availability;
output reg [1:0]
frame1,frame2,frame3,frame4,frame5,frame6,frame7,frame8;
output reg temp_ack;
wire [15:0] data_out_1;
always @ (posedge clock)
begin
if (reset)
begin
frame1 = 0; frame2 = 0;
frame3 = 0; frame4 = 0;
frame5 = 0; frame6 = 0;
frame7 = 0; frame8 = 0;
end
else if (sender_availability== 1'b1 ||
receiver_availability== 1'b1)
begin
frame1 = data_in[15:14];
```

```
@ (posedge clock);
frame2 = data_in[13:12];
@ (posedge clock);
frame3 = data_in[11:10];
@ (posedge clock);
frame4 = data_in[9:8];
@ (posedge clock);
frame5 = data_in[7:6];
@ (posedge clock);
frame6 = data_in[5:4];
@ (posedge clock);
frame7 = data_in[3:2];
@ (posedge clock);
frame8 = data_in[1:0];
end
else
begin
frame1 = 0; @ (posedge clock);
frame2 = 0; @ (posedge clock);
frame3 = 0; @ (posedge clock);
frame4 = 0; @ (posedge clock);
frame5 = 0; @ (posedge clock);
frame6 = 0; @ (posedge clock);
frame7 = 0; @ (posedge clock);
frame8 = 0;
end
end
always @ (posedge clock)
if (sender_availability == 1 || receiver_availability ==1)
begin
temp_ack <= 1'b0;
@ (posedge clock); @ (posedge clock);
@ (posedge clock); @ (posedge clock);
@ (posedge clock); @ (posedge clock);
@ (posedge clock);
data_out_temp = {frame1, frame2, frame3, frame4,
frame5, frame6, frame7, frame8};
@ (posedge clock); @ (posedge clock);
@ (posedge clock);
temp_ack <= 1'b1;
@ (posedge clock);
data_out_temp <= 0;
temp_ack <= 1'b0;
end
assign data_out_1 = data_out_temp;
assign data_out = data_out_1;
endmodule
```

C. piggybacking_test.v

This is a simple test bench file in which we generate a clock, reset, data_in, sender_availability/receiver_availability etc inputs and monitor the required outputs on waveforms window as well as console. We have designed the test bench to examine the possible test cases which include:

1) Sender_availability = 1 then Receiver_availability = 1; it gives ACK in terms of final_ack_data_ready
2) Receiver_availability = 1 then Sender_availability = 1; it gives ACK in terms of final_ack_data_ready.
3) Sender_availability = 1 and receiver_availability = 0; it gives final_ack_data_NOT_ready.

This way all the test scenarios can be examined to prove the concept of piggybacking.

```
module piggyback_test();
reg clock,reset;
reg [15:0] data_in;
reg sender_availability, receiver_availability;
wire [15:0] receiver_data_temp;
wire ack;
wire final_ack_data_ready;
wire final_ack_data_NOT_ready;
//wire [2:0] frame;
piggybacking p1 (.sender_data(data_in),
.clock(clock),
.reset(reset),
// .frame(frame),
.sender_availability(sender_availability),
.receiver_availability(receiver_availability),
.ack(ack),
.final_ack_data_ready(final_ack_data_ready),
.final_ack_data_NOT_ready(final_ack_data_NOT_ready,
.receiver_data_temp(receiver_data_temp) );
initial
begin
clock = 1'b0;
reset = 1'b1;
end
always
#2 clock = ~clock;
initial
begin
```

```
#15 reset <= 1'b0;
data_in = 16'b1111000010100101;
sender_availability= 1'b1;
receiver_availability= 1'b0;
#16 sender_availability= 1'b0;
#48 receiver_availability = 1'b1;
data_in = 16'b1010111101010000;
#16 receiver_availability = 1'b0;
#48 sender_availability = 1'b1;
data_in = 16'b0001110001110001;
#16 sender_availability = 1'b0;
#48 receiver_availability = 1'b1;
data_in = 16'b1111111100001111;
#16 receiver_availability = 1'b0;
#48 sender_availability = 1'b1;
data_in = 16'b0000000011111111;
#16 sender_availability = 1'b0;
#200 $stop;
end
initial
$monitor ($time, "clock=%b, data_in= %b,
receiver_data_temp= %b", clock, data_in,
receiver_data_temp);
endmodule
```

2. RESULTS

Schematic diagram of the implementation is as follows:

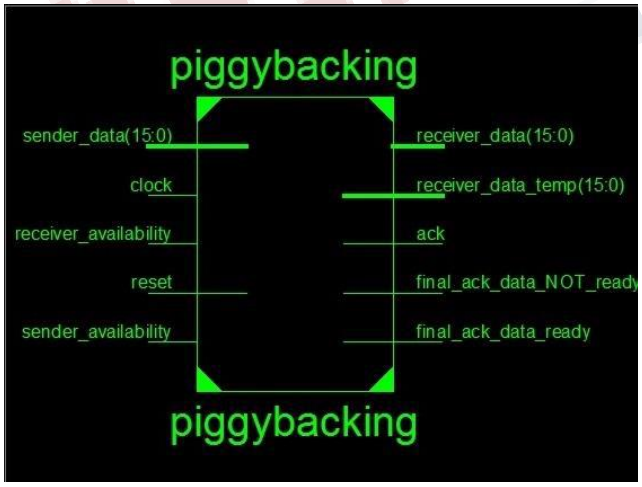

Fig 2 Schematic Diagram

Total memory usage of this implementation is 267108 kilobytes.

In fig. 3 of waveform if sender availability is high, it means that sender has data to send. In fig. 3 of wave form we can see that there are 8 frames 2 bits per frame. So on every clock cycle one frame will be sent. After all frames are sent ack will be high. In fig. 4 of waveform we can see that ack is set to high. Now we wait for few clock cycle to check if data is there or not. If data is there than final_ack_data_ready will high. If receiver has no data to send than final_ack_data_NOT_ready will high. In fig. 4 of waveform we can see that receiver_availability and final_ack_data signal both are high at the same time. It means that receiver has data to send and it is sending data with acknowledgement.

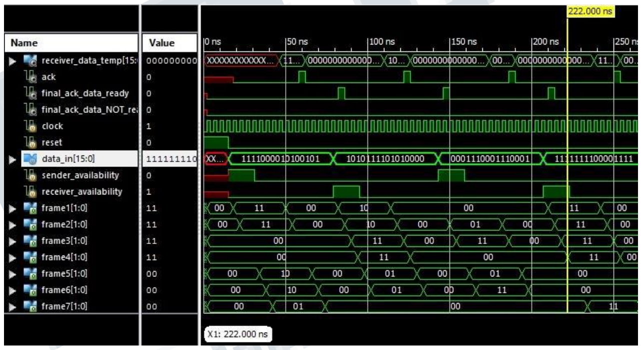

Fig. 3 Waveform 1

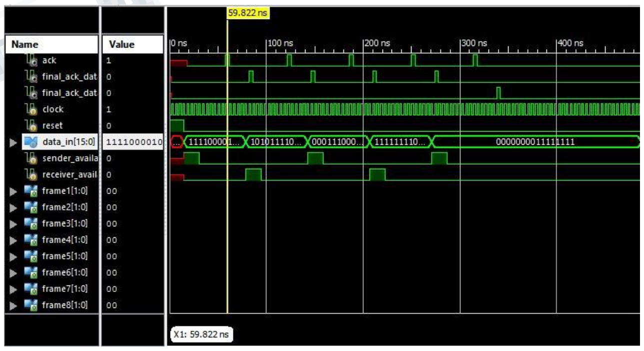

Fig 4 Waveform 2

3. CONCLUSION

By implementing sliding window protocol with the concept of piggybacking we are sending data and acknowledgement in a single frame. Thus we are sending more information in single frame. Thus the utilization of the bandwidth of the channel is increased. The memory consumption of this implementation is also very low that is 267108 kilobytes.

We can use this implementation on network processor to Increase the bandwidth utilization. Hardware implementation is also faster than software implementation.

FUNDING SOURCES

There is no involvement of any funding sources in this work.

REFERENCES

1. J. Ding, Y Shao, D. Zhang "Development of A Sliding Window Protocol for Data Synchronization in a Flow Cytometer," The 26th International Conference on Software Engineering and Knowledge Engineering, Hyatt Regency, Vancouver, BC, Canada, pp. 626-631, January 2014.

2. N. Nedjah, M. Mourelle, "High-Performance Hardware of the Sliding-Window Method for Parallel Computation of Modular Exponentiations," International Journal of Parallel Programming, vol. 37, pp. 537-555, December 2009.

3. O.Drogehorn, H. Hummer, W. Geisselhardt,"Formal Specification and Verification of Transfer-protocols for system-

4. A. Kind, R. Pletka, M. Waldvogel "The Role of Network Processors in Active Networks, IFIP International Working Conference on Active Networks, pp. 20-31, 2003.

5. S. Govind, R. Govindarajan, J. Kuri,"Packet Reordering in Network Processors," Parallel and Distributed Processing Symposium, 2007. IPDPS 2007. IEEE International, June 2007.

6. S. Ata., M. Murata, H. Miyahara, "Analysis of network traffic and its application to design of high-speed routers", IEICE Transactions on Information and systems, pp. 988-995, 2000.

7. J. Fu, O. Hagsand, "Designing and Evaluating Network Processor Applications", In Proc. of 2005 IEEE Workshop on High Performance Switching and Routing (HPSR) Hong Kong, pp. 142-146, 2005.

8. A. Kind, R. Pletka, M. Waldvogel.," The Role of Network Processors in Active Networks", International Federation for Information Processing, pp. 20–31, 2004.