# MANAGEMENT OF THE ROUTER'S INPUT BUFFER FOR MPP COMPUTERS WITH A DOUBLE-LOOP HYPERCUBE INTERCONNECT NETWORK

**Milen Georgiev Angelov**
Technical University – Varna, Chief Assistant Professor, PhD, Engineer

**Abstract**: *Data flow control in routers — which are among the main functional units in multiprocessor MPP systems — allocates their buffers and channel bandwidth to data units (packets and flits). Efficient implementation of this mechanism is essential for achieving high throughput and low latency. In this paper, a variant of input buffer control for receiving data into a selected FIFO queue of a router for a Double-loop Hypercube (DLH) interconnect network is presented. The analysis is performed at the low (hardware) level over a bidirectional full-duplex channel using Virtual Cut-Through flow control. Control state machines, their operating algorithms, and control signals are presented. The software component developed on this basis implements simulations of queue allocation and release during receiving data. The obtained results prove the correctness of the architectural solutions used – a high degree of utilization of the input channels, their frequency bandwidth, and throughput is achieved.*

**Keywords**: *Router, Virtual Cut-Through (VCT), input buffer, FIFO queue, Flow Control Digit (flit)*

## I Introduction

Routers in multiprocessor systems are of a great importance for their performance, as they form the cores of their interconnect networks [1, 5]. The design of a router of this class includes determining the flow control technique, the number of virtual channels, buffer organization, switches, and pipeline organization [10]. The goal is to minimize the latency of the transferred data units (flits and packets) while satisfying the bandwidth requirement.

The main principles and means used in creating the architectural platform of the router under consideration and its input buffers are:

1). Interconnect network: Static, with DLH topology, decentralized control, and asynchronous communication;

2). Routing: Minimal adaptive routing, allowing a packet to select one of several possible shortest paths [6, 7]. The Routing and Arbitration Unit block (shown below in Fig. 1) makes routing and arbitration decisions for each arriving packet;

3). General architecture: The input buffers (Fig. 1) consist of a pool of dedicated FIFO queues [2] that are directly connected to the Crossbar switch [5], with arbitration performed at each output channel;

4). Switching technique: Data transfer at the packet level using VCT switching technique and pipeline organization at the flit level for receiving and sending;

5) Input buffers: Based on a pool of eight FIFO queues (Fig. 2), where each queue is used to store only one packet whose length does not exceed the number of its storage elements, each of which can hold exactly one flit;

6). Packet transfer: A packet consists of one header flit (Header), a body of several flits (Body), and one tail flit (Tail). The information in the header flit determines its route. It is assumed that the packet may have variable length not exceeding 16 flits and that each flit is 32 bits wide;

7). Packet route: Consists of an ordered sequence of nodes in the network from the sender to the receiver. The resources used by a packet during its transfer are FIFO queues, switching elements (Crossbar), and link lines. The destination address in the Header flit, the routing algorithm, and the network state relative to the current node on the route determine the further path of the packet to the next node. After a packet's Header flit is received into a queue, the router determines the output channel based on the state of the resources (occupied buffers and output channel status) using the minimal adaptive routing algorithm. Allocation of input queues to output channels is performed using the iSLIP algorithm with Round-Robin arbiters (RRA) at the outputs [3];

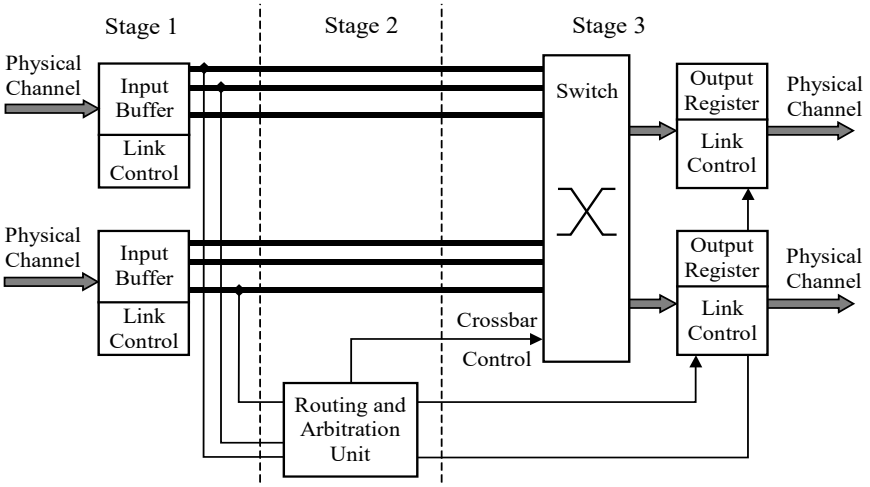


Fig. 1. Pipeline organization in a router for flit transfer

8). Pipelining: The three separate stages of data transfer at the physical level, shown in Fig. 1, are: Writing a flit into a free and selected queue of the input buffer (Flit Write); Routing, arbitration, and path setting through the switch (Routing, Arbitration and Path Setting); Switch Traversal [3, 4, 5].

**II Architecture of the input buffer of the router**

The input buffers of the router are critical for achieving low latency and high throughput during packet transfer [8, 9]. Fig. 2 shows the architecture of the input buffer. It implements the

VCT switching technique, enabling transfer on packet-level. The input buffer consists of a demultiplexer DEMUX, eight FIFO queues, a PRIORITY ENCODER, a B_FIFO_STATUS register for queue status, a DC decoder for FIFO queue selection, and control logic. The necessary control signals are defined and described in the program section below. The following principles and solutions are used in this architecture:

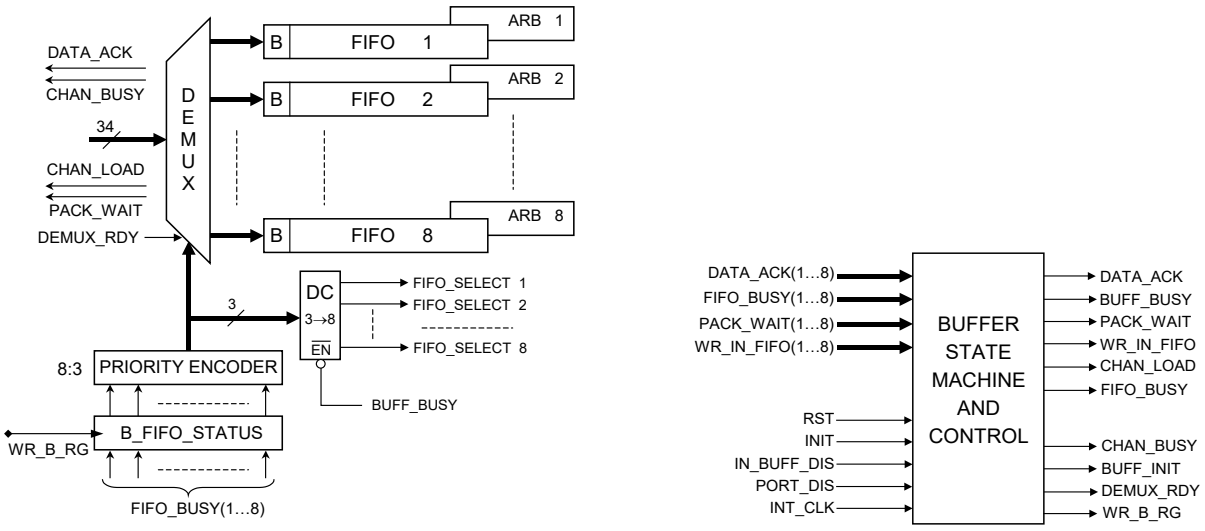


Fig. 2. Architecture of the input buffer of a router and its control signals

1). The basis of the input buffer is the pool of FIFO queues, whose structure is shown below in Fig. 3. Each queue has its own separate local arbiter ARB[k], k∈[1..8] (Fig. 2), which generates requests (not shown here) upon receiving a Header flit in it towards the arbiters of possible output channels;

2). The flow control signals refer to packets, not to flits;

3). Each queue can be in one of two states: occupied or free, symbolically shown in Fig. 2 by the flag B (Busy). The state of an input channel is encoded with signals as follows: A). Busy (CHAN_BUSY=1) – all queues are occupied; B). Heavily loaded (CHAN_BUSY=0, CHAN_LOAD=1) – only one queue is free; C). Lightly loaded (CHAN_BUSY=0, CHAN_LOAD=0) – two or more queues are free. This state is forwarded upstream to the previous router;

4). The states of the queues in a buffer before each packet reception are stored in the B_FIFO_STATUS register via the WR_B_RG signal. The priority encoder always selects the first free queue. It can accept any packet arriving on the link, regardless of which output channel it is destined for. After the start of packet reception, the selected queue transitions to the occupied state (B=1);

5). Each of the router's output channels uses its own RRA (not shown here), ensuring fairness in servicing the input FIFO queues. Thus, conflicts and racing occur only for the output channels;
6). The FIFO queues of all input channels are directly connected to the switch (Fig. 1), which allows more than one packet to be sent from one input buffer to the output channels in a given time interval;
7). After a selected FIFO queue receives the HEADER flit of an arriving packet, it generates the PACK_REQ signal (shown in Fig. 3) to the local arbiter, which starts the routing and arbitration phase for the packet.

**III Structure of a FIFO queue in the input buffer**

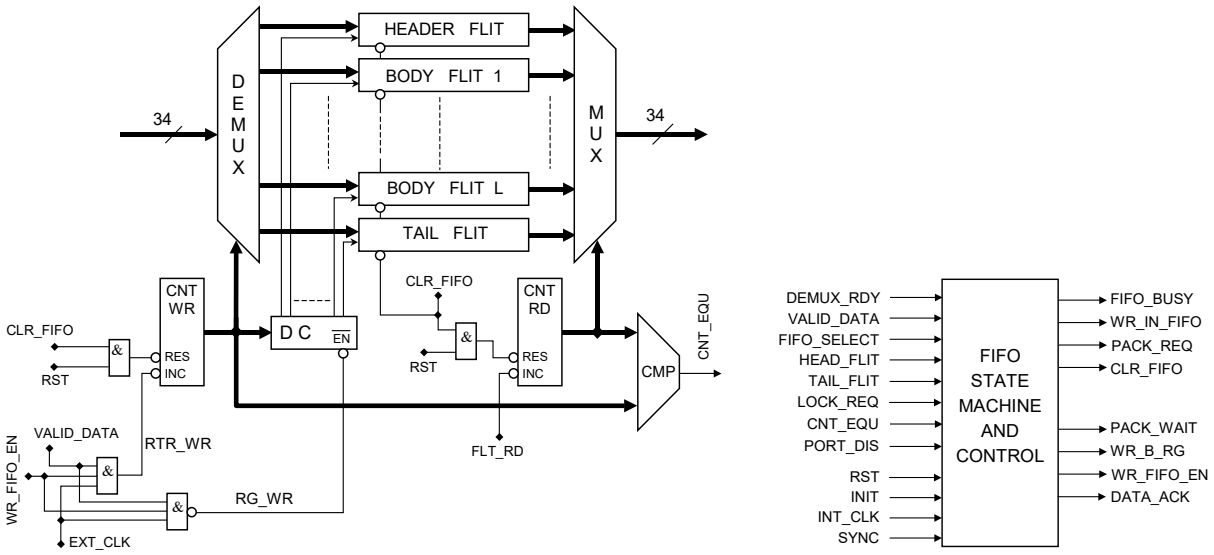


Fig. 3. Structure of a FIFO queue in the input buffer and its finite state machine

Fig. 3 shows the structure of one of the FIFO queues, which consists of the following main components: a demultiplexer DEMUX, (L+2) storage elements for flits, L∈[0..14], a write counter CNT_WR together with a decoder DC for selecting the next empty storage element in the queue, a read counter CNT_RD, a comparator CMP, a multiplexer MUX, logic, and control. The width of the storage elements is 34 bits (see Fig. 2 and Fig. 3). The width of one flit is 32 bits, with the remaining two bits encoding the flit type: HEADER, BODY, or TAIL. The control signals shown in Fig. 3 are described in the program section below.

The start of packet receiving is triggered when a free and selected queue is available and both sides are ready, negotiated via the signals VALID_DATA, PACK_WAIT, and CHAN_BUSY (Fig. 2 and 3). Immediately after receiving the header flit, the PACK_REQ signal is generated, which starts the routing and arbitration phase for the packet (the router's allocator / Routing and Arbitration Unit is not described here). The FIFO queue allows simultaneous write and read of flits. Writing is

performed via the EXT_CLK signal using the CNT_WR counter and DC decoder, while reading is performed using the CNT_RD counter via the FLT_RD signal. Queue release is registered by the CNT_EQU signal after reading the last flit.

**IV Control of the input buffer**

Fig. 4 shows the state graph of the finite state machine of one input buffer, which is shown above in Fig. 2. Below are the states and the signals (defined and described in the program section below) asserted in each of them:

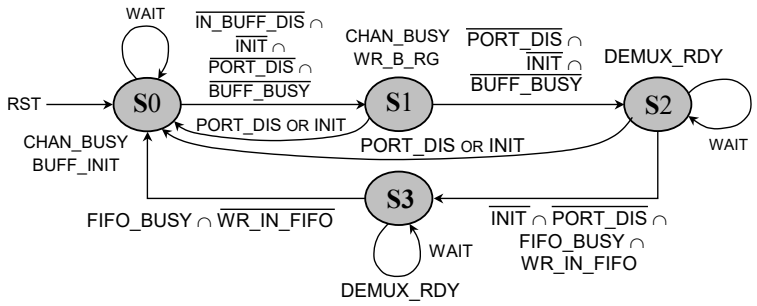


Fig. 4. State diagram of the finite state machine controlling one input buffer

***S0***: Initial state (*Init*). The signals CHAN_BUSY and BUFF_INIT are generated;

***S1***: (*Save_Queue_Status*). The current state of each queue (free/occupied) of the respective input buffer is written into the B_FIFO_STATUS register. The signals WR_B_RG and CHAN_BUSY are generated;

***S2***: (*Wait_Packet*). The start of writing a packet into the selected queue is awaited. This event is registered by the WR_IN_FIFO signal. The DEMUX_RDY signal is generated;

***S3***: (*Write_Packet*). A packet is written into the selected queue of the input buffer and the end of writing this packet is awaited. The DEMUX_RDY signal is generated.

**V Management of a queue in the input buffer**

Fig. 5 shows the state graph of the finite state machine of one queue in the input buffer, shown above in Fig. 3. Below are the states and the signals (defined and described in the program section below) generated in each of them:

***S0***: Initial state (*Init*). The signals FIFO_BUSY and CLR_FIFO are generated;

***S1***: Inactive state (Idle). The queue is not connected to the channel input;

***S2***: Ready state (*Ready*). The queue is selected, connected to the channel input, and awaits writing of a header flit from a packet. The signals FIFO_BUSY, WR_FIFO_EN, and PACK_WAIT are generated;

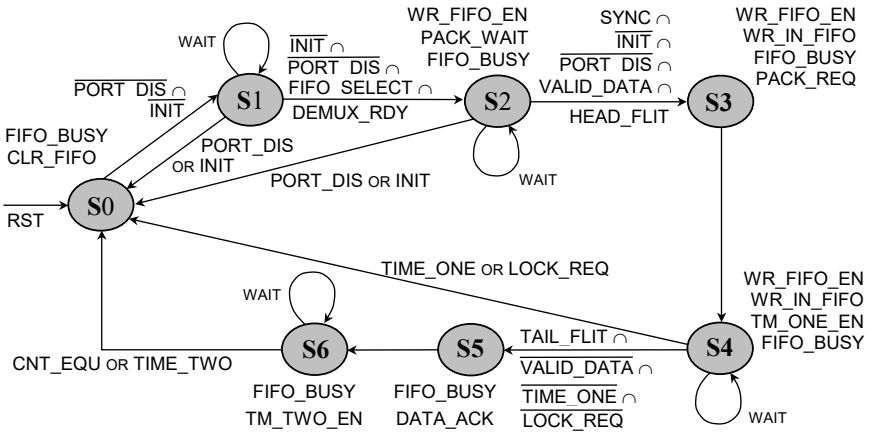


Fig. 5. State graph of the finite state machine controlling one FIFO queue in the input buffer

***S3***: Active state (*Write_Packet*), during which a packet is written. The signals FIFO_BUSY, WR_FIFO_EN, WR_IN_FIFO are generated, as well as the routing and arbitration request PACK_REQ, synchronized with the beginning of an allocation time interval;

***S4***: Active state (*Wr&Rd_Packet*), during which a packet is written and its reading may start and continue. The signals FIFO_BUSY, WR_FIFO_EN, WR_IN_FIFO are generated, and a timeout timer is enabled via the TM_ONE_EN signal;

***S5***: Active state (*Read_Packet*), during which a packet is read. The signals FIFO_BUSY and DATA_ACK (acknowledgment to the previous router of successful packet reception) are generated;

***S6***: Active state (*End_Packet*), in which the end of reading the packet stored in the queue is awaited; this is registered by the CNT_EQU signal. The signals FIFO_BUSY and TM_ONE_EN are generated; the latter enables a timeout timer for exit from the state in case the packet remains unserved for some reason.

**VI Program section for the management of one FIFO queue in the input buffer**

In the program fragment presented below, the algorithm of the finite state machine for controlling one FIFO queue is developed, written in Verilog (a hardware description language used for the design, simulation, and verification of electronic circuits).

```
// Design Name:          HPC_Router
// Module Name:          Router_FIFO_Queue_FSM
// Project Name:         Router_ FIFO_Queue
// Tool Versions:        Xilinx PlanAhead 14.6
//
// **   DECLARATIONS, INITIALIZATION, CORE LOGIC
// **-------------------------------------------------------------------------------------------**
//
`timescale 1ns/1ps              // Modeling Time, Step of Modeling
```

```
//
//         Constants
// **-------------------------------------------------------------------------------------------------**
`define  DLY                    0.5             // Signals Delay on the Device
`define  TMR_1_bits             4               // Width of timeout Timer_1
`define  TMR_2_bits             6               // Width of timeout Timer_2
`define  Timer_1_Start_Value    4'b 1111        // Current Load Value of Timer_1
`define  Timer_2_Start_Value    6'b 11_1111     // Current Load Value of Timer_2
//
//         Main Program
// **-------------------------------------------------------------------------------------------------**
module Router_FIFO_Queue_FSM (
   input  RST,                  // Asynchronous RESET with active HIGH Level, from HOST
   input  INT_CLK,              // FSM Clock, from Local Node
   input  INIT,                 // Initialization by DMA with active HIGH Level
   input  SYNC,                 // Synchronization, Tsync = 2 x Tint_clk
   input  DEMUX_RDY,            // Enable Input Channel's DEMUX, from this Buffer's FSM
   input  VALID_DATA,           // Enable Data on Physical Bus between Nodes, from other Router
   input  FIFO_SELECT,          // Select the Current Queue by Decoder
   input  HEAD_FLIT,            // Current Flit is HEAD
   input  TAIL_FLIT,            // Current Flit is TAIL
   input  LOCK_REQ,             // Release the Queue from Current Packet
   input  CNT_EQU,              // The Queue is Empty after Read
   input  PORT_DIS,             // Disable the Input Buffer during the Initialization Process
   input  TIME_ONE,             // TimeOut_1, from Counter_1 into FSM
   input  TIME_TWO,              // TimeOut_2, from Counter_2 into FSM
//
   output FIFO_BUSY,            // This Queue in the Moment is BUSY, from FSM
   output WR_IN_FIFO,           // Write Packet into Queue at this Moment, from FSM
   output PACK_REQ,             // Request to Queue's Arbiter for Packet, from FSM
   output CLR_FIFO,             // Clear all Elements into Queue, from FSM
   output PACK_WAIT, // The Queue is Ready to Receive Packet, from FSM
   output WR_FIFO_EN,           // Enable Write Packet via EXT_CLK into Queue, from FSM
   output DATA_ACK,             // All Packet is written into Queue, from FSM
   output TM_ONE_EN,            // Enable Counter_1 -> Timer_1, from FSM
   output TM_TWO_EN,            // Enable Counter_2 -> Timer_2, from FSM
   output Timer_1,              // State of Timer_1
   output Timer_2 );            // State of Timer_2
//
// **   DECLARATIONS                                                                **
// ********************************************************************************
   wire            RST;
   wire            INT_CLK;
   wire            SYNC;
   wire            INIT;
   wire            DEMUX_RDY;
   wire            VALID_DATA;
   wire            FIFO_SELECT;
   wire            HEAD_FLIT;
   wire            TAIL_FLIT;
   wire            LOCK_REQ;
   wire            CNT_EQU;
   wire            PORT_DIS;
   wire            TIME_ONE;
   wire            TIME_TWO;
```

```
//
   wire             FIFO_BUSY;
   wire             WR_IN_FIFO;
   wire             PACK_REQ;
   wire             CLR_FIFO;
   wire             PACK_WAIT;
   wire             WR_FIFO_EN;
   wire             DATA_ACK;
   wire             TM_ONE_EN;
   wire             TM_TWO_EN;       */
//
         reg[8:0]   FSM_FIFO_Queue_State;
//
//       FSM_FIFO_Queue_State have 7 States
// ---------------------------------------------------------------------------------------------
         parameter[8:0]
INIT_State                  = 9'b 1_0010_0000,          // State S0
IDLE_State                  = 9'b 0_0000_0000,          // State S1
READY_State                 = 9'b 1_0001_1000,          // State S2
WRITE_PACKET_State          = 9'b 1_1100_1000,          // State S3
WR_RD_PACKET_State          = 9'b 1_1000_1010,          // State S4
READ_PACKET_State           = 9'b 1_0000_0100,          // State S5
END_PACKET_State            = 9'b 1_0000_0001;          // State S6
//
// **   FSM Outputs
// **-------------------------------------------------------------------------------------------**
assign FIFO_BUSY       = FSM_FIFO_Queue_State [8];
assign WR_IN_FIFO      = FSM_FIFO_Queue_State [7];
assign PACK_REQ        = FSM_FIFO_Queue_State [6];
assign CLR_FIFO        = FSM_FIFO_Queue_State [5];
assign PACK_WAIT       = FSM_FIFO_Queue_State [4];
assign WR_FIFO_EN      = FSM_FIFO_Queue_State [3];
assign DATA_ACK        = FSM_FIFO_Queue_State [2];
assign TM_ONE_EN       = FSM_FIFO_Queue_State [1];
assign TM_TWO_EN       = FSM_FIFO_Queue_State [0];
//
         reg[`TMR_1_bits-1:0]   Timer_1;            // TimeOut Timer_1
         reg[`TMR_2_bits-1:0]   Timer_2;            // TimeOut Timer_1
//     All Conditions in one Body
//-----------------------------------------------------
always @(posedge INT_CLK or posedge RST) begin
         if(RST) begin                                      // First is the RESET
           FSM_FIFO_Queue_State <= #`DLY INIT_State;
           Timer_1 <= #`DLY `Timer_1_Start_Value;
           Timer_2 <= #`DLY `Timer_2_Start_Value;
         end
         else begin
//
//       Change the State of FSM, if they are Conditions
//-----------------------------------------------------
         case(FSM_FIFO_Queue_State)
              INIT_State: begin                                 // S0
                if(~INIT & ~PORT_DIS) begin
                  FSM_FIFO_Queue_State <= #`DLY IDLE_State;
                  Timer_1 <= #`DLY `Timer_1_Start_Value;
```

```
        Timer_2 <= #`DLY `Timer_2_Start_Value;
      end
    end
//
    IDLE_State: begin                                   // S1
      if(PORT_DIS | INIT) begin
        FSM_FIFO_Queue_State <= #`DLY INIT_State;
      end
      else begin
        if(~INIT & ~PORT_DIS & FIFO_SELECT & DEMUX_RDY) begin
          begin
            FSM_FIFO_Queue_State <= #`DLY READY_State;
          end
        end
    end
//
    READY_State: begin                                  // S2
      if(PORT_DIS | INIT) begin
        FSM_FIFO_Queue_State <= #`DLY INIT_State;
     end
     else begin
        if(~INIT & ~PORT_DIS & SYNC & VALID_DATA & HEAD_FLIT) begin
          FSM_FIFO_Queue_State <= #`DLY WRITE_PACKET_State;
        end
     end
    end
//
    WRITE_PACKET_State: begin                           // S3
       FSM_FIFO_Queue_State <= #`DLY WR_RD_PACKET_State;
    end
//
    WR_RD_PACKET_State: begin                           // S4
      if(LOCK_REQ | (Timer_1 == 0)) begin
        FSM_FIFO_Queue_State <= #`DLY INIT_State;
      end
      else begin
        if(TAIL_FLIT & ~VALID_DATA & ~LOCK_REQ) begin
          FSM_FIFO_Queue_State <= #`DLY READ_PACKET_State;
        end
        else begin
          Timer_1 <= #`DLY Timer_1 - 1;
        end
      end
    end
//
    READ_PACKET_State: begin                            // S5
      FSM_FIFO_Queue_State <= #`DLY END_PACKET_State;
    end
//
    END_PACKET_State: begin                             // S6
      if(CNT_EQU | (Timer_2 == 0)) begin
        FSM_FIFO_Queue_State <= #`DLY INIT_State;
      end
      else begin
        if(TM_TWO_EN) begin
```

```
                Timer_2 <= #`DLY Timer_2 - 1;
              end
            end
          end
        endcase
      end
endmodule
```

**VII Experimental results**

The experimental results of the study were obtained through simulations using the PlanAhead software product from Xilinx. The simulations investigate and visualize, via timing diagrams, the operation of one FIFO queue under the control of its finite state machine with a clock period of INT_CLK=5.00ns and DLY=0.50ns. These values are used for the visualization of signals on the timing diagrams and for explanations of the queue operation at specific moments in time.

The state machine is asynchronously set to the initial state via RST. All other transitions occur on the rising edge of INT_CLK upon the corresponding valid combinations of input control signals. The input signals to the state machine (except RST, INT_CLK, and SYNC) are grouped into the Out_Vector for greater clarity. They are defined in the program using the *input* directive, while the output signals of the state machine are defined using *output*. The output signals are: FIFO_BUSY, WR_IN_FIFO, PACK_REQ, CLR_FIFO, PACK_WAIT, WR_FIFO_EN, DATA_ACK, TM_ONE_EN, and TM_TWO_EN. The states of Timer_1 and Timer_2 for timeout exit are shown on the timing diagrams below the output signals. One input vector includes INIT, DEMUX_RDY, VALID_DATA, FIFO_SELECT, HEAD_FLIT, TAIL_FLIT, LOCK_REQ, CNT_EQU, PORT_DIS, and RST. The combinations within each vector and the sequence of vectors examine the behavior of the state machine: states, generated output signals, and transitions from one state to another depending on the input signals. The behavior of the queue was analyzed with respect to the process of starting and ending packet write.

Fig. 6 visualizes the start of writing a packet into a selected FIFO queue of the input buffer. After 40 ns the state machine is in the *Ready* state. At 47 ns HEAD_FLIT is activated (presence of a header flit), as a result of which at 50 ns the state machine transitions to the *Write_Packet* state, and at 55 ns it unconditionally transitions to *Wr&Rd_Packet*. Timer_1 is started for a transition to the Init state via TimeOut. At 60 ns the first change in the state of Timer_1 can be seen. In this case, the time from strobing HEAD_FLIT (50 ns) to the start of writing into the queue (60 ns) is two clock cycles.

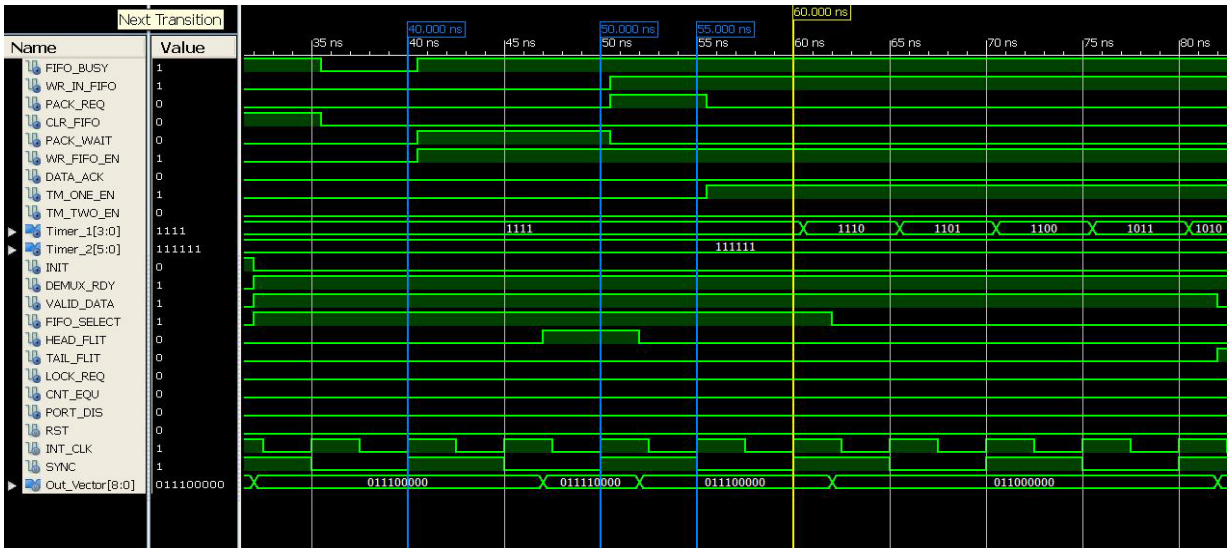


Fig. 6. Start of writing a packet into a queue of the input buffer

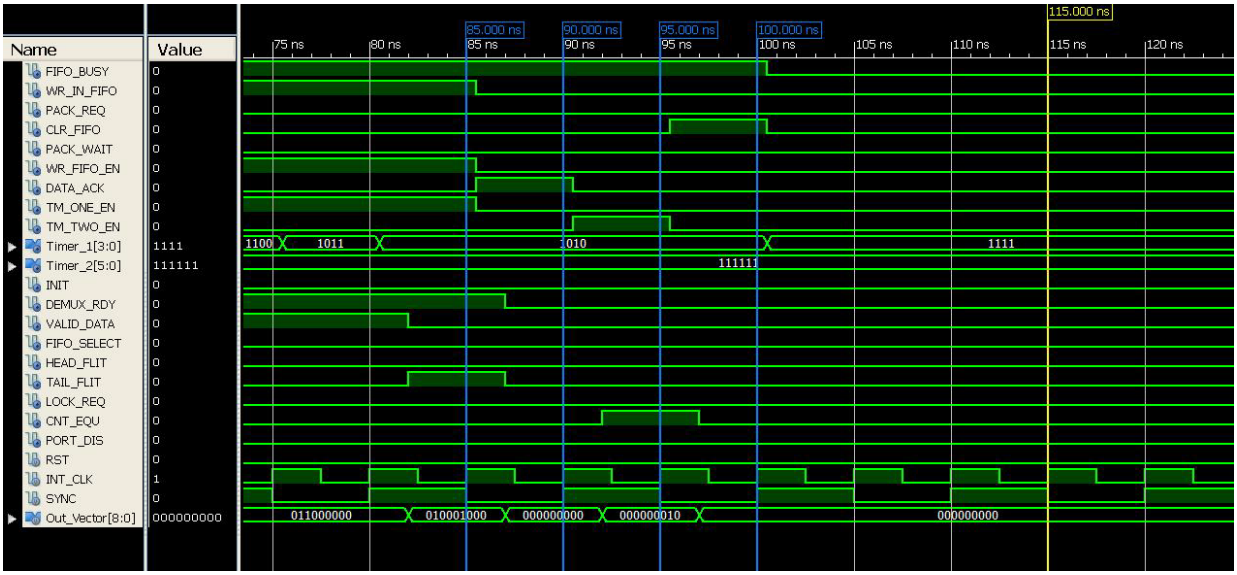


Fig. 7. End of writing a packet into a queue of the input buffer

Fig. 7 visualizes the end of writing a packet into a selected FIFO queue of the input buffer. The end of writing occurs at 82 ns, when TAIL_FLIT indicates the presence of a received tail flit. At 85 ns the state machine transitions to the *Read_Packet* state, Timer_1 stops counting, and at 90 ns it unconditionally transitions to the *End_Packet* state, in which reading of the packet from the queue or timeout exit via Timer_2 is awaited. In this case, at 92 ns the condition (CNT_EQU=1) indicates the end of reading from this queue; at 95 ns the state machine transitions to the Init state, and at 100 ns it transitions to *Idle*, where the queue waits to be selected by the input buffer for writing another packet. In this case, the time for releasing the queue from 85 ns to the *Init* state at 95 ns is 2 clock cycles.

Fig. 8 visualizes the start of writing a packet into a FIFO queue of the input buffer under conditions different from those shown in Fig. 6 (INIT and VALID_DATA are established at

different times). Here, until 127 ns the queue is in the *Idle* state. Activation of DEMUX_RDY and FIFO_SELECT by the state machine of its input buffer selects this queue at 130 ns, when it transitions to the *Ready* state. Immediately afterward, at 132 ns HEAD_FLIT is activated, indicating the presence of a header flit, but at 135 ns PACK_REQ is not generated because synchronization with the SYNC signal is required; 5 ns are waited. At 140 ns the state machine transitions to the *Write_Packet* state and generates (PACK_REQ=1); at 145 ns it unconditionally transitions to *Wr&Rd_Packet*, deasserts PACK_REQ, and starts Timer_1 for a transition to the *Init* state via timeout. At 150 ns the first change in the state of Timer_1 can be seen. In this case, the time from strobing HEAD_FLIT (135 ns) to the start of writing into the queue (150 ns) is three clock cycles due to the later delayed synchronization.

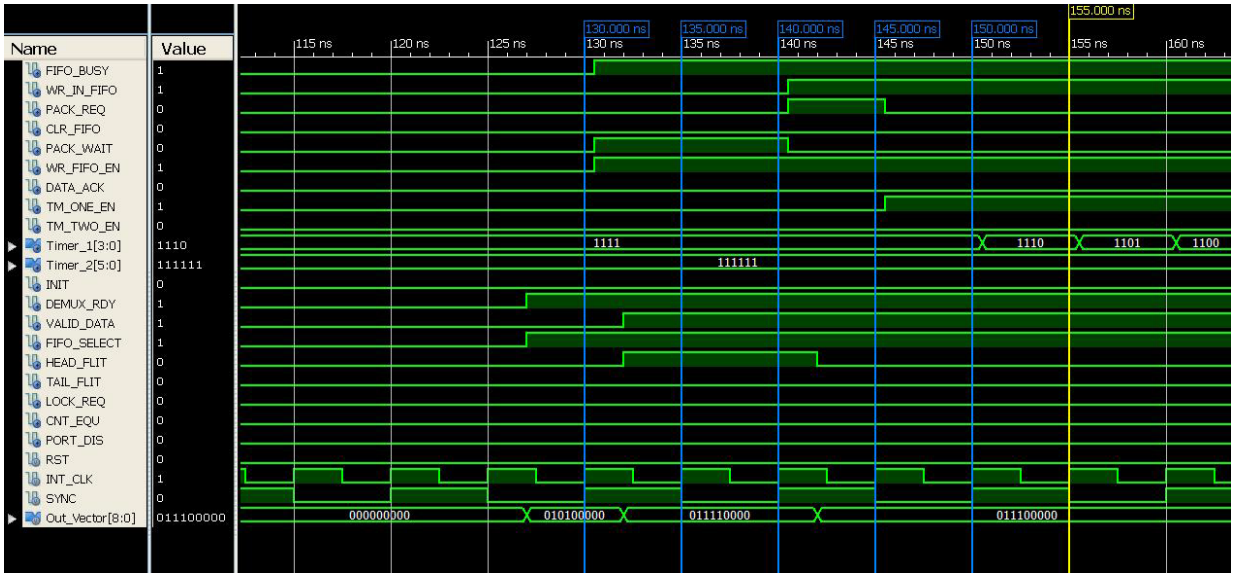


Fig. 8. Second variant for the start of writing a packet into an input FIFO queue

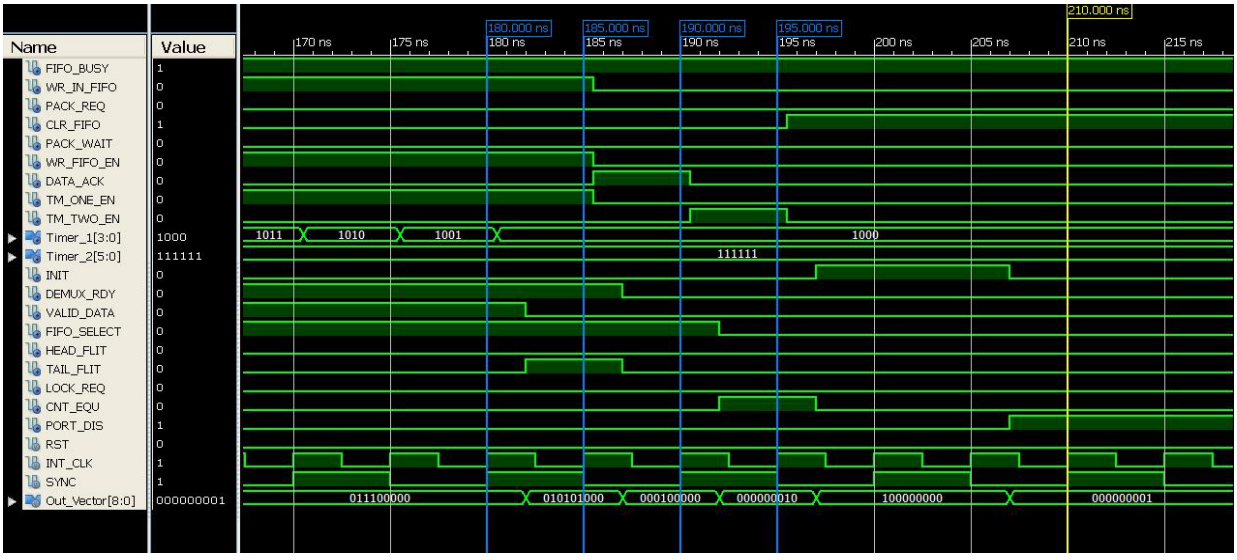


Fig. 9. Second variant for the end of writing a packet into a queue of the input buffer

Fig. 9 shows a second variant for the end of writing a packet into a selected FIFO queue. TAIL_FLIT is activated at 182 ns, indicating the presence of a received tail flit. At 185 ns the state machine transitions to the *Read_Packet* state, Timer_1 stops counting, and at 190 ns it unconditionally transitions to the *End_Packet* state, in which reading of the packet from the queue (verified by the CNT_EQU=1 signal) or timeout exit via Timer_2 is awaited. In this case, at 192 ns the CNT_EQU signal indicates the end of reading from this queue, and at 195 ns the state machine transitions to the *Init* state, where conditions for transition to *Idle* are awaited. In this case, the time for releasing the queue from 185 ns to the *Init* state at 195 ns is 2 clock cycles, but the transition to Idle is not determined.

**VIII Conclusion**

In this paper, one of the main elements of a router for a DLH network with VCT flow control is presented – the input buffer containing a pool of FIFO queues controlled by their finite state machines. The experimental results from the studies conducted through simulations prove the correct operation of this functional unit in accordance with the set goal – minimum latency during packet transfer. The timing diagrams show optimal management – fast allocation and release of the selected queue for operation. The measurements visualized by the timing diagrams were performed on the basis of the number of clock cycles for control (allocation and release) of the FIFO queues at the beginning and end of packet writing. The obtained results of 2 or 3 clock cycles for starting to write into a queue (depending on the specific case) and 2 clock cycles for releasing a queue on the internal clock signal INT_CLK prove the high speed of switching between the queues of the input buffer and minimum latency during packet reception. This, in turn, determines the high throughput and frequency bandwidth of the input channels in the router.

**References**


[1]. Ping-Jing Lu, Ming-Che Lai, Jun-Sheng Chang, A Survey of High-Performance Interconnection Networks in High-Performance Computer Systems, Electronics 2022, 11, 1369, https://doi.org/10.3390/electronics11091369

[2]. E. Fard et al, An Efficient NoC Router by Optimal Management of Buffer Read and Write Mechanism, Microprocessors and Microsystems 89, 11 January 2022, https://doi.org/10.1016/j.micpro.2022.104440

[3]. James Aweya, Switch/Router Architectures Systems with Crossbar Switch Fabrics, Published June 25, 2024 by CRC Press, ISBN 9781032654218

[4]. M. Rezaei-Zare, M. Fathy, A. Rezaei-Zare, Design of a high-performance router with distributed shared-buffer for load balancing for on-chip networks, Microelectronics Journal, Volume 132, February 2023, https:// doi.org/10.1016/ j.mejo. 2022.105647

[5]. Dejun Shi, Xiaohu Han, Weijian Chen, et al. Overview of Router Architecture in High Performance Computing, Proc. SPIE 12717, 3rd International Conference on Artificial Intelligence, Automation, and High-Performance Computing (AIAHPC 2023), 127171R (21 Jul 2023); https://doi.org/10.1117/12.2686172

[6]. Ruoyao Xiao, Yiming Cui, Yuanyong Liang. Efficient Routing Algorithms for HPC Interconnect Networks, 2024, ⟨hal-04595791⟩
[7]. M. Besta, J. Domke et al., High-Performance Routing with Multipathing and Path Diversity in Ethernet and HPC Networks, IEEE Transactions on Parallel and Distributed Systems (Vol. 32, Issue: 4, 01 April 2021) Page(s): 943–959, DOI: 10.1109/ TPDS.2020.3035761
[8]. S. Salunke1, R. Kinagi, Design and Performance Evaluation of a High-Speed VLSI Router Using Efficient Buffering Techniques, IJRASET, Volume 14 Issue Jan 2026, ISSN: 2321-9653, ww.ijraset.com
[9]. M. Katta, T.K. Ramesh, J. Plosila, AS-Router: A novel allocation service for efficient Network-on-Chip, Engineering Science and Technology, an International Journal, 50 (2024), https://doi.org/10.1016/j.jestch.2023.101607
[10]. M. T. Balakrishnan, T. G. Venkatesh, A. V. Bhaskar, Design and Implementation of Congestion Aware Router for Network-on-Chip, Integration, the VLSI journal, 88 (2023) 43–57, https://doi.org/10.1016/j.vlsi.2022.08.012