A VLSI Self-Compacting Buffer for Priority Queue Scheduling

Suryanarayana B. Tatapudi and José G. Delgado-Frias
School of Electrical Engineering and Computer Science
Washington State University
Pullman, WA 99164-2752
jdelgado@eecs.wsu.edu

Abstract

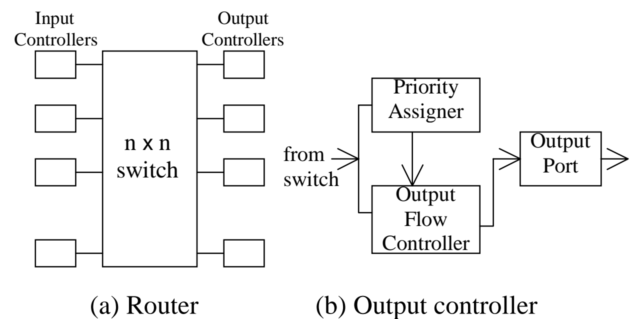
This paper describes a novel VLSI CMOS implementation of a self-compacting buffer (SCB) for the dynamically allocated multi-queue (DAMQ) switch architecture. The SCB dynamically allocates data regions within the output buffer for different priority values of the data. The proposed implementation provides high performance solution to buffered communication switches that are required to interconnect networks. This performance comes from the implementation of the DAMQ, pipelining and novel circuitry. The major components of SCB are described in detail in this paper. The system has the capability of performing a read, a write or a simultaneous read/write operation per cycle.

Keywords: Self-compacting buffer, dynamically allocated multi-queue, rotating priority queue, networks.

1 Introduction

There have been drastic changes in the computer-networking world in the past decade. The number of communicating entities on the network has grown exponentially and new networking applications have been developed. These changes brought about the need for high performance networks and in turn the need to design high performance network infrastructure and components. One of the important network infrastructures is the *router* and network performance is closely related to its architecture. This paper describes a VLSI design and implementation of a self-compacting buffer [1] at the output port of a router that has significant contribution to the router performance.

A router is composed of input controller, an nXn switch, and output controllers. The input controller receives incoming packets at the input port and determines the appropriate output port number according to a routing algorithm. The nXn switch delivers the packets from the n input controllers to the n output controllers. The output controller buffers the packets temporarily, according to the priority assigned to them and sends them to the neighboring node. Figure 1 shows an example of a block diagram for a router and an output controller. The output controller has three major responsibilities. First, it has to receive the packets from the switch and distribute the header of the packet to the priority assigner. Second, the packet and the priority assigned to it are forwarded to the packet flow controller and buffer space is allocated. Third, buffer space is de-allocated and the packet is sent to the output port.



(a) Router (b) Output controller

Figure 1. Router block diagrams.

Tamir and Frazier [2] have classified the buffered switch architectures into four major types based on how the queues are manipulated and how data is stored. The four types are: First-in first-out (FIFO), statically allocated fully connected (SAFC), statically allocated multi-queue (SAMQ), dynamically allocated multi queue (DAMQ). FIFO, SAFC, and SAMQ buffered switches do not use the buffer space efficiently [2]. Dynamically allocated multi-queue (DAMQ) is a better way of implementing the buffer as buffer space is allocated and de-allocated depending on the demand at a particular time. Research studies have reported that DAMQ achieves the best performance among the four implementations [1,2]. The self-compacting buffer is a VLSI implementation of the DAMQ.

The organization of this paper is as follows. Section 2 introduces the self-compacting buffer architecture and its properties. In section 3, the basic cell designs and implementations of the buffer, buffer controller, priority pointers, and priority pointer controller are described in detail. Section 4 explains the system timing and some concluding remarks are provided in section 5.

2 Self-Compacting Buffer

Self-compacting buffer (SCB) implementation is based on the dynamically allocated multi-queue scheme for buffer management. The organization of the SCB is as

shown in Figure 2. It consists of buffer, buffer controller, priority pointers, and priority pointer controller.

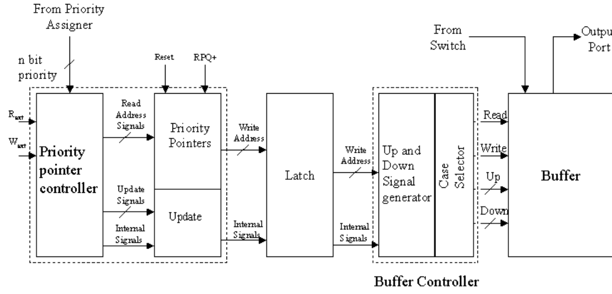


Figure 2. Self-compacting buffer organization

The SCB stores the data from the switch, and transfers data to the output port. When a write occurs, based on the priority assigned an address is generated from the priority pointers and sent to the buffer controller. The buffer controller sets the corresponding lines to shift the data in the buffer and create a vacancy at the address it received. The new data is then stored in the vacant location in the buffer. If a read occurs the read request is forwarded to the buffer controller. Data is always read from the head of the buffer and sent to the output port. The priority pointers are updated to accommodate for the changes in the buffer depending on the read and or write requests serviced.

The self-compacting buffer stores data based on the priority assigned to it, so the buffer is dynamically divided into regions storing data with a particular priority. This scheme supports the dynamically allocated buffer method introduced by Tamir and Frazier [2]. In this buffer implementation to provide Quality of Service (QoS) for transmission based on some priority assigned for data, Rotating-Priority-Queue$^+$ (RPQ$^+$) [3] data-scheduling algorithm has been implemented. The self-compacting buffer scheme has the following properties:

Property 1: For n bit priority assignment, there are 2n priority regions (queues) in the buffer indexed as 0^+, 1, 1^+, 2, 2^+, 3, 3^+… (n-1), $(n-1)^+$, n. Data always arrives with priority values 1, 2, 3 … n-1, n.

Property 2: Let j, k denote the priority levels set for the data. With j<k, the dynamically allocated region for priority j and k always have addresses A_j, A_k such that $A_j < A_k$.

Property 3: If data packets with a particular priority are not available, then no region in buffer is reserved for that priority.

Property 4: Within the space for each priority, the data is stored in a first-in first-out (FIFO) fashion.

Property 5: The data can be written into any priority level in the buffer, but the read from the buffer takes place only from the highest priority level 0^+ (the head of the buffer).

The description and properties of the self-compacting buffer suggest that data write and read requests trigger insertion and deletion of data in the buffer and there should be a mechanism to access arbitrarily regions associated with a particular priority value. When a write request occurs, vacancy must be created for insertion of data somewhere in the middle of the buffer and this requires the moving of all the data, which reside in buffer locations below the vacancy. Similarly, when a read request comes, the data is read from the head of the buffer and a vacancy is created. There is again data movement involved to temporarily fill the vacancy created. The following Figure 3 depicts the buffer space in the self-compacting buffer implementation.

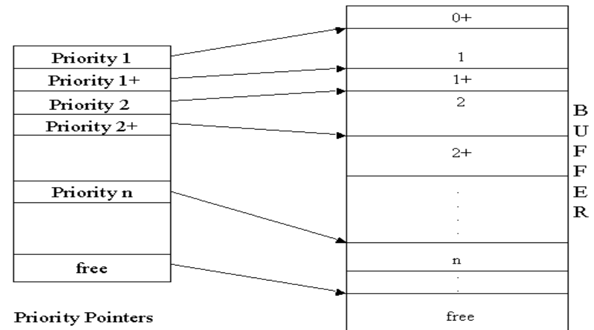


Figure 3. Buffer space

3 Buffer Implementation

This section describes the VLSI implementation of the self-compacting buffer architecture presented in Section 2. The requirements and circuitry of the buffer, buffer controller, priority pointers are presented here.

3.1 Buffer Organization and Cell Design

The buffer organization is as shown in Figure 4. Buffer consists of a finite number of storage locations. For each buffer location, the following actions can occur:

Shift up: row contents are copied into the row above it.
Shift down: row contents are copied onto the row below it.
Hold: row contents are held, no change occurs.
Write: contents of the write bus are copied into a row.
Read: contents of row at the head of buffer are pushed out to the output port.

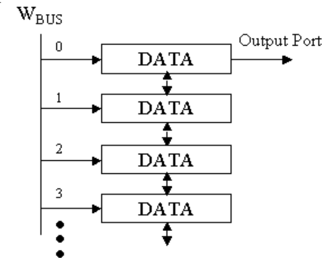


Figure 4. Buffer organization

The buffer performs these actions ones the proper signals are generated. The buffer is made up of a basic

buffer cell. All the cells in a row share the same signals to perform the above actions. The cells in a column share the share the write bus. The Figure 5 presents the CMOS implementation of the buffer cell.

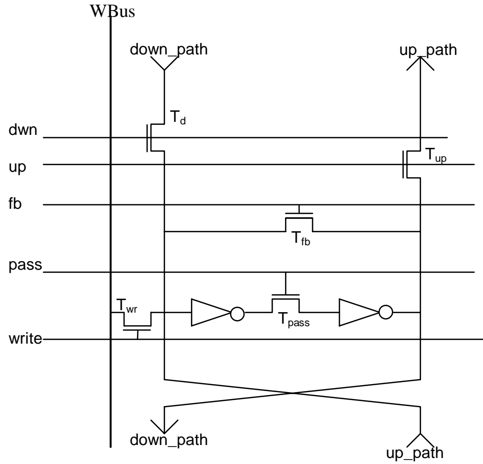


Figure 5. Buffer cell

The cell design makes it possible to perform all the actions listed above. These actions on data are implemented as follows:

Shift up/down data: When the data has to be shifted up or down by a row, the cells must be able to separate the incoming the data from the outgoing data. The feedback transistor T_{fb} and the pass transistor T_{pass} turn off, isolating the data-in and data-out. Transistors T_d and T_{up} are turned on to shift the data down or up respectively. When shifting data down from a storage cell in row k to k+1, the path is set as follows. The data stored in cell of row k, passes through transistor $T_d(k+1)$ into the first inverter of cell in row k+1. Similarly, when shifting data up, the transistor $T_{up}(k)$ turns on, passing the data from cell in row k to the input of first inverter of cell in row k-1. The data shifted up from the cells in the first row is the data to be read.

Hold data: When there are no write/read or shifting actions, the cells must retain their data. Transistors T_{fb} and T_{pass} are on during this time, acting as feedback in the cells, thus allowing the cells to hold their data.

Write data: If a write request occurs, the transistor T_w is turned on and the transistors T_{fb} and T_{pass} are turned off. The transistor T_w passes the data from the write bus to the cell, while the T_{fb} and T_{pass} transistor isolate the incoming data from the outgoing data. This allows the implementation of write and shift up/down actions on the same row simultaneously.

3.2 Buffer Controller

The self-compacting buffer has three operations, read, write and simultaneous read/write. There are three different cases, which identify the actions of the buffer cell. These three cases are:

Case 1) Single Write (Insertion): For an address generated to write the data into the buffer, all buffer locations with address less than this write address retain their data. The data in buffer locations with addresses greater than or equal to the write address must be shifted down by one buffer location, to create a vacancy for incoming data.

Case 2) Single Read (Deletion): All the buffer locations shift their data up by one buffer location. The data shifted up from the first buffer location is sent to the output port.

Case 3) Simultaneous Read/Write: All buffer locations with address greater than the write address retain their data, while the buffer locations with addresses less than or equal to the write address, shift their data contents up by one location, creating a vacancy at the buffer location with write address.

The buffer controller bases its operations on the case of the current request and generates control signals for; data movement within the buffer, from the write bus (W_{BUS}) to the buffer locations, from the first buffer location to the output port.

The case selector in the buffer controller does a selection between up and down signals generated by the internal read signal and decoder. The decoder implementation here is different from the conventional address decoder. Instead of setting a single line to logic 1 depending on the address, all the lines below this are also set to logic 1, while the lines above are set to logic 0. When a single write occurs, the buffer controller decodes the address and generates a write signal for the corresponding buffer location. The rest of the locations below the write locations are set to shift down their stored data. This case corresponds to case 1. When a single read occurs, the buffer controller generates up signals for all the buffer locations to shift up the data stored. This one corresponds to case 2. When a simultaneous write and read occur, the write address is decoded and based on it, the location of write and all locations above it are set to shift their data up.

When the buffer is full, no write request and write address are sent to the buffer controller, unless a simultaneous write and read occur. This prevents any data writing to the buffer unless in the later case, where in a free buffer location is created due to the read operation. In case of an empty buffer no read request is forwarded to the buffer controller preventing any reading since the buffer has no valid data stored in it.

Once the down and up signals have been generated, they pass to the buffer through the case selector, which selects a suitable case. For a single write, all of the down lines below the selected location are set to logic 1 and all the down lines for locations above the selected location are set to logic 0. All up lines are set to logic 0. In case of a single

read, all the down lines are set to logic 0 and all the up lines are set to logic 1. In these two cases, the outputs of the case selector are identical to its inputs, i.e. they pass unaltered. When a simultaneous write and read occur, all the down lines are set to logic 0. For the location where the write occurs and the locations above it, the up lines are set to logic 1, while all other locations have their up lines set to logic 0.

The CMOS circuit for case selector is shown in Figure 6. Figure 7 illustrates the outputs of case selector for different cases.

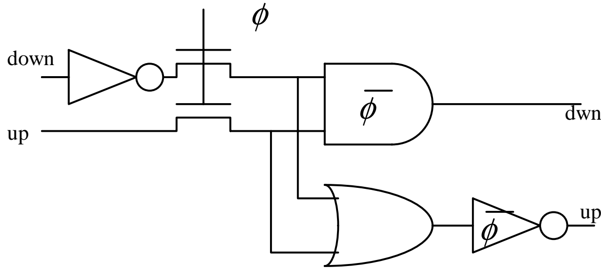


Figure 6. Case Selector in Buffer controller

Address	down	up	dwn	up
0 (read)	0	0	0	0
1	0	0	0	0
2	0	0	0	0
3	0	0	0	0
	0	0	0	0
j-1	0	0	0	0
j (write)	1	0	1	0
j+1	1	0	1	0
	1	0	1	0
p	1	0	1	0

Address	down	up	dwn	up
0 (read)	0	1	0	1
1	0	1	0	1
2	0	1	0	1
3	0	1	0	1
	0	1	0	1
j-1	0	1	0	0
j (write)	1	1	0	0
j+1	1	1	0	0
	1	1	0	0
p	1	1	0	0

Figure 7. Down and Up lines from Case Selector

3.3 Priority Pointers

The self-compacting buffer implementation is for output port in the router system. Data for a particular output port is dynamically allocated buffer space on a priority basis in the output port's buffer. For each priority value of data there is a priority pointer (address) which points to the buffer location corresponding to the beginning of the data for that priority. The data in that particular priority is stored in a FIFO fashion. It can be said that each priority is FIFO queue implementation. Each priority data space is dynamically updated depending on the write or read operations that occur. This dynamic updating makes the buffer to expand or contract, thus changing the location of data. So, the priority pointers must keep up with these changes to point to the beginning of the data corresponding to that priority.

When a single read occurs, data is always read from the top of the highest priority 0^+, decreasing the buffer space by one, and all priority pointers update their addresses, decrementing by 1. When a single write occurs, data is written to the bottom of the selected priority increasing its space. All buffer locations starting from the location where the write takes place are updated. Priority pointers for lower priority values than the current data's priority, update the addresses they store, incrementing by 1. When a simultaneous write and read occur, the buffer locations belonging to the same priority where the write is to take place, and the locations above these are updated. The pointer for priority where the write is to occur and all higher priorities are updated, decremented by 1 and the other priority pointers are unchanged. If the buffer is empty or full, the priority pointers update is canceled when a read or write request respectively arrive at the input.

In this buffer implementation to provide Quality of Service (QoS) for transmission based on some priority assigned for data, a data-scheduling algorithm has been implemented. The scheduler implemented is the Rotating-Priority-Queue$^+$ (RPQ$^+$) scheduler [3] an approximation of Earliest-Deadline First (EDF) with rotating FIFO queues.

The RPQ$^+$ scheduling implementation has 2n ordered FIFO queues, indexed as 0^+, 1, 1^+, 2, 2^+, 3, 3^+… (n-1), $(n-1)^+$, n. Data is always read from the highest priority FIFO queue 0^+ of the buffer. Data arrives with one of the priorities 1,2,3 … n, assigned to them. Depending on the priority data is added at the end of the FIFO queue of that priority. After every Δ time units the FIFO queues are rotated in a two-step process; the first step is the *concatenation step*, the second the *promotion step*. In the concatenation step, the current FIFO queue p and FIFO queue p^+ are merged to form FIFO p and in the promotion step this FIFO queue p is relabeled as FIFO queue $(p-1)^+$. These two steps result in the data in a priority p moving to higher priority. The Figure 8 illustrates this queue rotation operation of the RPQ$^+$ scheduler.

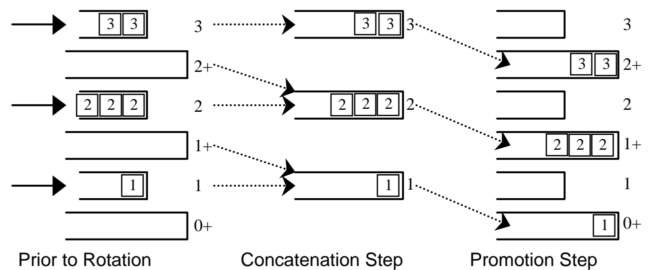


Figure 8. Example for RPQ$^+$ Scheduler's queue rotation operation.

The storage of data at different buffer locations is already based on priority and FIFO queue implementation. The implementation of the RPQ$^+$ is clear from following

explanation about the priority pointer cell. The CMOS implementation of the priority pointer cell is shown in Figure 9.

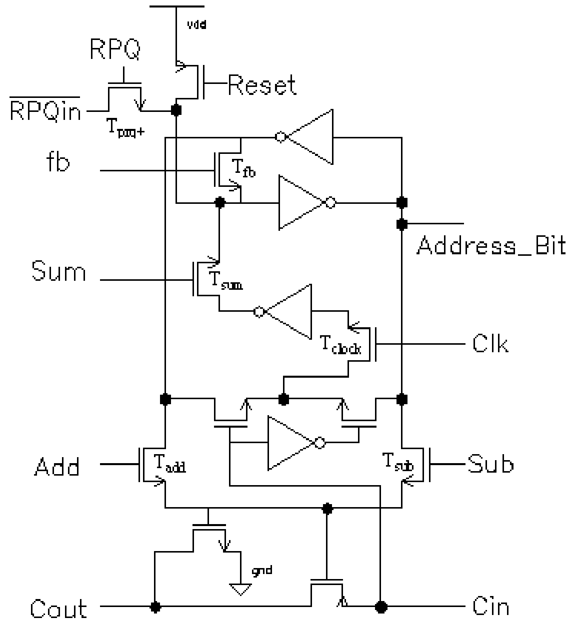


Figure 9. CMOS implementation of Priority Pointer cell

The priority pointers can be reset to address 0, (indicating an empty buffer) by enabling the transistor T_{reset}, and now all the pointers point to the head of the buffer. The priority pointer controller generates the signals Add, Sub when read and/or write requests arrive. The sum (or difference) when incrementing (or decrementing) a priority's starting address is generated through an XOR of the stored address bit and the carry in C_{in}. The sum signal controlling the T_{sum} transistor is set to logic 1 when add or subtract occurs. Through the transistors, T_{clock} and T_{sum} the result of the above XOR operation is passed to the pointer memory cell (the inverters). The two transistors T_{clock} and T_{sum} are not turned on at the same time, as it would result in a closed loop through the inverters and the output of the XOR. Adding or subtracting determines whether C_{in} is propagated to C_{out} or killed (set to logic 0). If the add signal is set to logic 1, transistor T_{add} is turned on allowing the complement of address bit to pass. Therefore C_{out} is equal to C_{in} when the address bit is set to logic 1 otherwise C_{out} is killed. Setting the sub signal to logic 1, allows the address bit to pass through transistor T_{sub}. In this case, if the address bit is set to logic 0, C_{out} is equal to C_{in}. Finally, while no operations are being performed transistor T_{fb} is turned on to provide feedback. The transistor T_{rpq+} is used to implement the rotation step in the RPQ^+ scheduling described above. There are two RPQ^+ signals; RPQ^+_1, RPQ^+_2, generated from the external RPQ^+ signal and the RPQ^+_1 signal always precedes the RPQ^+_2 signal. The RPQ^+_1 signal is given to the T_{rpq+} transistors of the priority pointers with index 1^+, 2^+, …$(n-1)^+$. When these transistors are enabled, the addresses stored in the pointers 2, 3, ….n, are copied into these pointers, completing the concatenation step of the RPQ^+ queue rotation. Then the RPQ^+_2 signal enables the T_{rpq+} of the priority pointers with index 1, 2, …n-1 and the new addresses stored in the pointers 1^+, 2^+, …$(n-1)^+$, are copied into these pointers, completing the promotion step of the RPQ^+ queue rotation.

The priority pointer controller generates the signals needed to update the priority pointers according to the tables shown in Figure 10 (example shown here for 3 priority levels 1, 2, 3. Free corresponds to the unused buffer space) and passes them through a case selector shown if Figure 11. The case selector [4] has a different implementation from the one in Figure 6, but achieves a similar function with add and sub signals passed to the pointers.

Writing into		Generate these signals					
P_1	**P_o**	**A_1**	**A_{1+}**	**A_2**	**A_{2+}**	**A_3**	**A_{free}**
0	1	0	1	1	1	1	1
1	0	0	0	0	1	1	1
1	1	0	0	0	0	0	1

Inputs to case selector when write occurs

S_1, S_{1+}, S_2, S_{2+}, S_3, S_{3+}, S_{FREE} = 1

Inputs to case selector when read occurs

Figure 10. Inputs to Case selector for write and read requests

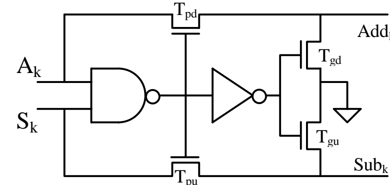


Figure 11. Case Selector in Priority Pointer Controller.

For a given priority pointer, if the S_k or A_k is a 1 or both are 0, the circuit lets the input to pass on; transistors T_{pd} and T_{pu} are on and transistors T_{gd} and T_{gu} are off. However, when the S_k and A_k lines are both 1, T_{pd} and T_{pu} are off and T_{gd}, T_{gu} are on. Thus, both lines are set to logic 0 preventing any update of priority pointers. This occurs during a simultaneous read/write

4 System Timing

The current implementation of self-compacting buffer uses a single-phase clock scheme and pipelining has been introduced for performing the operations on the priority pointers controller, priority pointers, buffer controller and buffer; thus the operations can overlap.

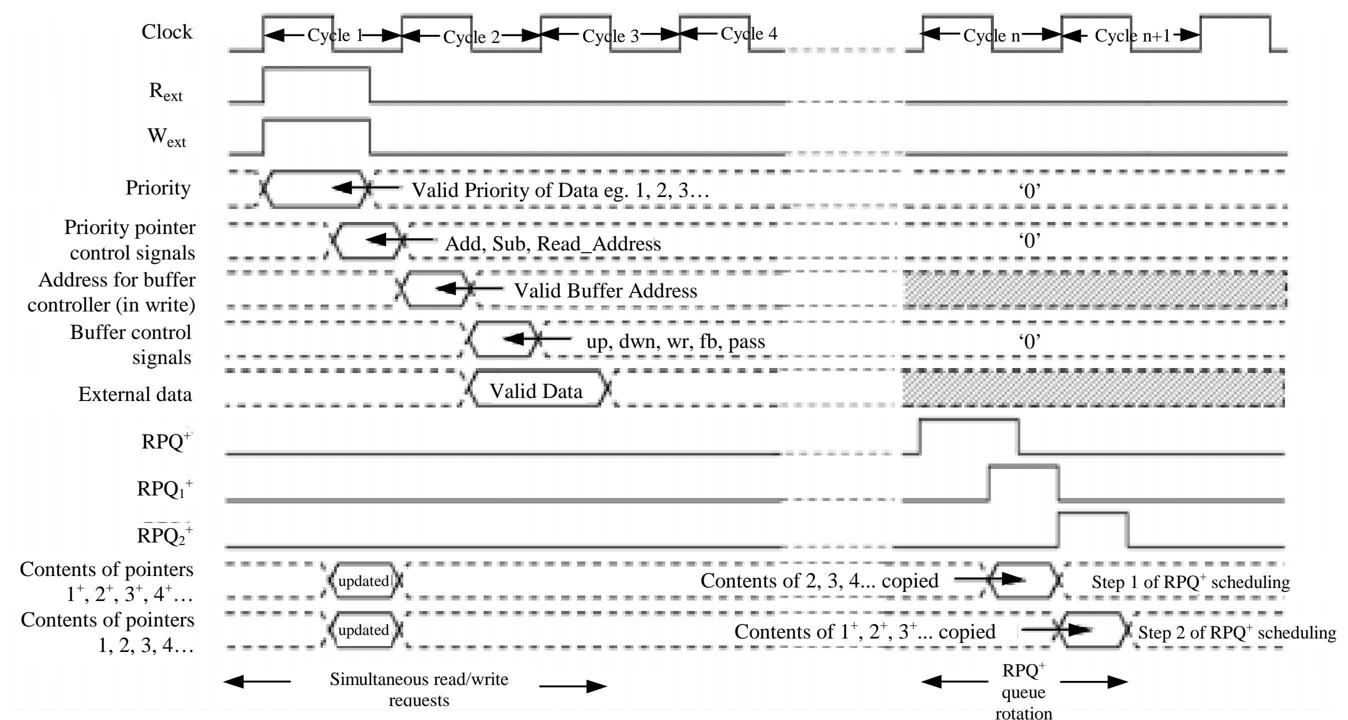


Figure 12. System Timing Diagram

The above Figure 12 is the timing diagram for the operations that can occur. The operations shown in the diagram are simultaneous read/write and RPQ^+ queue rotation. The priority pointer controller receives the signals R_{ext}, W_{ext} (external read and write signals), P_n (n-bit external priority value for the data to be stored) at the rising edge of the clock ϕ. At the falling edge of the first clock period, the pointer controller generates add, sub, and read_address signals for the priority pointer to be updated. The update of pointers takes place during the first clock period, when clock is low. If there is write request for priority *p*, before the update of pointers is done, the address is read from the priority pointer p^+ and is latched during the clock high period. In the second clock period at the falling edge, buffer controller generates the control signals for the buffer update. The buffer update takes one clock period.

5 Concluding Remarks

A novel VLSI CMOS implementation of a self-compacting buffer (SCB) for the dynamically allocated multi-queue (DAMQ) switch architecture has been presented in this paper. The DAMQ switch has been shown to provide the best performance among the buffered switch architectures [2]. The self-compacting buffer presented here is a novel implementation of the DAMQ switch architecture. The SCB allocates buffer space on demand and so there is optimum use of buffer space and data storage demands can be met efficiently. To provide Quality of Service, the Rotating Priority Queue (RPQ^+) scheduling scheme has been implemented.

This paper presents the SCB architecture and the VLSI CMOS implementations of the basic cells for each block of the system. The SCB consists of the buffer, buffer controller, priority pointers, and priority pointer controller. Novel circuitry has been used in the implementation of all the blocks. The SCB is capable of performing a read, a write or a simultaneous read/write operation.

References

[1] J. Park, B.W. O'Krafka, S. Vassiliadis, J.G. Delgado-Frias, "Design and Evaluation of a DAMQ Multiprocessor Network With Self-Compacting Buffers," *IEEE Supercomputing '94*, pp. 713-722, Washington D.C., November 1994.

[2] Y. Tamir and G.L. Frazier, "Dynamically-Allocated Multi-Queue Buffers for VLSI Communication Switches," *IEEE Transactions on Computers*, Vol. 14, No. 6, pp. 725-737, 1992.

[3] J. Liebeherr and D. E. Wrege, "Priority Queue Schedulers with Approximate Sorting in Output Buffered Switches," *IEEE Journal on Selected Areas in Communications. Special Issue on Next Generation IP Switches and Routers*, vol. 17, no. 6, pp. 1127-1145, June 1999.

[4] J. G. Delgado-Frias and R. Diaz, "A VLSI Self-Compacting Buffer for DAMQ Communication Switches," *IEEE 8th Great Lakes Symp. on VLSI*, pp.128-133, Feb. 1998.