WR Packet Switching Core (swcore) informal specification version 0.2

Tomasz Włostowski/CERN BE-Co-HT

Table of Contents

1.Introduction....2
2.Swcore interfaces....3
2.1.WRF and RTU inputs....3
2.2.WRF outputs....3
3.Swcore operation....3
3.1.Multiport memory....5
3.2.MMU (Page allocator)....5
3.3.Input block....6
3.4.Output block....6
3.5.Page transfer arbiter....6

1. Introduction

The White Rabbit Packet Switching Core (further abbreviated as swcore) is an HDL core implementing a multiport, non-blocking, protocol-agnostic packet switching fabric. The main application for the swcore is the WR Switch.

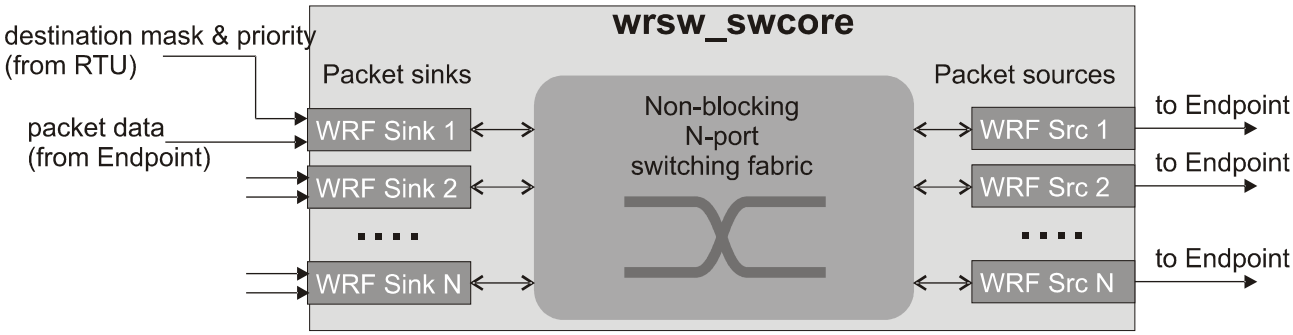

Figure 1. Function of the swcore.

As shown on **fig.1**, the swcore has the following interfaces:

- generic-configurable number (**N**) of WR Fabric Interface (WRF) sinks, receiving packets from WRF-compliant cores, such as the WR Endpoint or the WR NIC,
- **N** RTU (Routing Table Unit) interfaces. Each RTU interface is associated with a WRF sink and provides the routing destination and priority information for each incoming packet.
- **N** WRF sources, outputting the processed packets

The range of N can be restricted to 4..32 due to implementation constraints.

The swcore shall implement a gigabit switching fabric fulfilling the following requirements:

- **store-and-forward** operation - each packet shall be first fully received and stored in the fabric buffer and afterwards (if it has been received without errors – *i.e. the endpoint connected to the packet source didn't indicate an error – T.W.*) transferred to the output(s) determined by the decision of the RTU.
- **non-blocking**. That means that each port shall be capable of transmitting packets to any other port at its full data rate.
- generic-**configurable number of priority queues (1 to 4)** for handling 802.1q/IP DSCP priority tags.

2. Swcore interfaces

2.1. WRF and RTU inputs

The input side of the Swcore has N input ports, each of them consisting of a WRF packet sink (only WRF mandatory signals) and RTU decision input port. The WRF interface is described in detail in a separate document.

The RTU decision input comprises the following VHDL ports:

- `rtu_drdy_i`: RTU decision ready. When active, there is a valid RTU decision present on the 3 ports below,
- `rtu_dst_port_mask_i`: RTU destination port mask. N-bit vector, whose bits represent the Swcore outputs to which the incoming packet shall be sent (for example, if bit *x* is 1, the packet shall be sent to port *x*, if all bits are ones the packet shall be broadcast, etc.)
- `rtu_drop_i`: RTU drop packet command. When active, the swcore shall not forward the incoming packet.
- `rtu_prio_i`: Final priority of the packet, expresed as the ID of the output priority queue.
- `rtu_ack_p1_o`: Asserted for 1 clock cycle by the swcore to acknowledge the reception of the current RTU decision.

Each incoming packet is processed only if its associated RTU decision is present. The RTU decision is made based on the packet's MAC addresses, ethertype, VLAN ID and priority. These are provided to the RTU directly by the endpoint.

2.2. WRF outputs

The output side of the swcore has N WRF outputs (sources), supporting the following WRF optional signals:

- `terror_p1` (sink error indication, e.g. *a transmit buffer underrun in the endpoint TX path*).
- `tabort_p1` (sink-initiated transfer abort, e.g. *transfer abort initiated by the endpoint/HP routing unit due to preemption caused by an incoming HP packet*).

3. Swcore operation

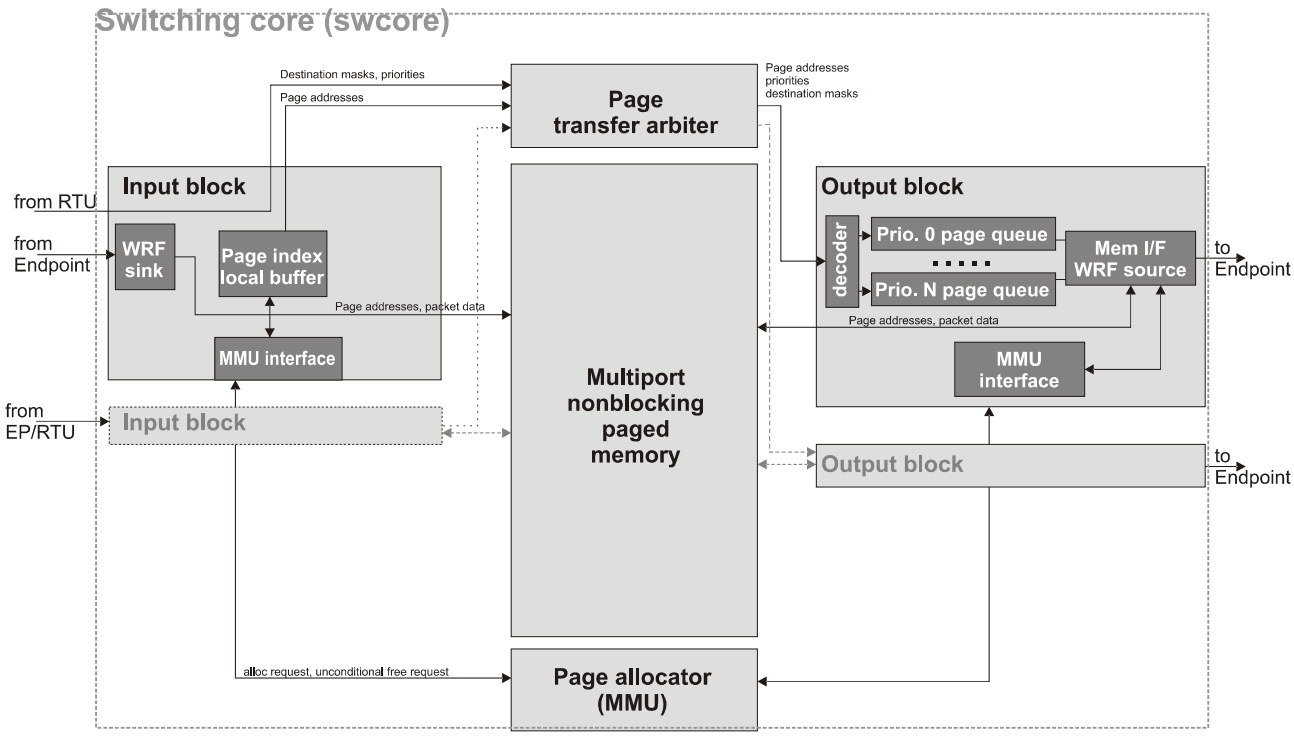

Figure 2. Swcore block diagram

The conceptual block diagram of the Swcore is shown on **fig. 2**. The design can be split into 5 sub-entities:

- N **input blocks (IBs)**, receiving the packets and their RTU decisions,
- N **output blocks (OBs)**, transmitting the routed packets with assigned priorities
- N-read, N-write paged **multiport memory (MPM)**, used as a store-and-forward packet buffer
- **page allocator (MMU)** – memory management unit, allocating and freeing the pages.
- **page transfer arbiter (PTA)** – unit transferring the page sets storing the packets from the input blocks to the output blocks.

The simplified Swcore algorithm looks like this:

1. A packet comes to one of the **input blocks** along with its RTU decision (i.e. DREQ=1 and SOF_P=1 on WRF and RTU_DRDY=1)
2. The **input block** counts the number of bits in the RTU destination port mask (DPM) and starts allocating the memory pages using the **MMU**, taking the number of bits as the "usage count" value. Pages are allocated a few clock cycles in advance (i.e. before the page currently being written is full), to sustain the full gigabit data rate. Packet data is written directly into the allocated **multiport memory** pages. Numbers (indices) of allocated pages are stored in input block's local page index buffer.
3. If a WRF transfer error occurred, the pages are immediately and unconditionally released (see 3.2)
4. If the packet was successfully written into the multiport memory, the input blocks request a time slot from the **page transfer arbiter**. Then, the page indices along with the DPM and RTU priority are transferred from the input block to all the output blocks, which have ones in their DPM bits. Note that page index array transfers must be atomic (i.e. during the assigned timeslot, page addresses for a complete packet must be transferred).

5. The output block decodes the priority and inserts the page addresses into one of the output The queues.
6. The output block chooses the non-empty queue with the highest priority and reads out the page indices. Then, it starts up a WRF packet transfer, reads the contents of the pages and outputs them on its WRF source.
7. If the transfer was aborted by the sink connected to Swcore output block, for example due to preemption or endpoint error, the transmission is repeated up to [generic-configurable] times
8. If the transfer was successful or there was an error in the sink (not an abort), the pages are released in the "normal" way (i.e. decreasing their usage counters)

Example scenario (8-port Swcore)

1. A packet arrives at input block 2 with RTU priority = 3 and DPV = "10000100"
2. The IB starts allocating pages with use count=2 and stores their indices in a local buffer. These are, for example "10, 20, 30, 40, 50". The IB immediately starts writing the packet data into the MPM pages "10, 20, 30, 40....". (so the page allocation and filling the MPM is performed simultaneously).
3. The IB detects an end-of-packet condition. The packet size was not a multiple of the page size – so the IB appends a single word with control field = 0xF after the last data word of the packet to mark the end of packet in the MPM. Since the packet was received without errors, the IB will pass it to the appropriate OBs
4. The IB requests the PTA for a timeslot. The PTA grants a timeslot.
5. The IB transfers the page indices (10, 20, 30, 40, 50) and (RTU priority = 3) to the OBs having ones in their assigned bits in the DPV. In this case, the destination OBs are 2 and 7.
6. OBs 2 and 7 receive the RTU priority and subsequent page indices from the PTA. Since the priority = 3, the indices (10..50) are written into output queue 3. When the OBs have received a complete page set from the PTA, they start reading the page contents from the MPM and outputting the read data to the WRF sources. Upon completion of each page the OBs send a release request to the MMU.
7. The pages were allocated by the IB with usage count = 2. Both OBs have requested page release (each release request decreased the usage count by 1), so now pages (10..50) are free again and can be used for storage of another packet.

3.1. Multiport memory

The multiport memory is the heart of the Swcore. Its role is to store the packets to be forwarded and provide simultaneous, nonblocking read/write access to all input/output blocks. The memory must fulfill the following requirements:

- N read ports and N write ports, all of them capable of operating simultaneously. N must be generic-configurable.
- paged access with generic-configurable page size (32..512 words)
- generic-configurable size (16k words … 256k words)
- generic-configurable data word size (16..24 bits).

The memory shall also store the out-of-band and control data (WRF ctrl_ bus), so in the default case, the data word is 20 bits-wide (16 data + 4 ctrl).
(see packet_mem.vhd, packet_mem_write_pump.vhd, packet_mem_read_pump.vhd)

3.2. MMU (Page allocator)

The MMU manages the occupation of the multiport memory pages. The pages are allocated and written by the input blocks, and are read and released by the the output blocks. The MMU therefore must support the following operations:

- **page allocation** upon request from the input block. For each allocated page, the input block provides a "usage counter" value. The address of the allocated page is outputted to the requesting input block.
- "normal" **page release**, which checks the value of the usage counter for the page at a given address and decrements it. When the usage counter reaches 0, the page is released.
- **unconditional page release** (triggered by the input block in case of an error in the incoming packet), which releases the page without checking the value of its usage counter.

The MMU shall provide N independent allocation and unconditional page release request lines for the input blocks and N "normal" page release request lines for the output blocks. Since the rates of page alloc/release requests are low compared to the system clock frequency, the MMU can be wrapped inside a time multiplexing arbiter.

The multiport MMU shall also inform the input block each time a page (previously allocated by the same input block) is released by providing an appropriate strobe signal.

(see page_alloc.vhd, multiport_page_alloc.vhd)

3.3. Input block

The input block(s) perform the following tasks:

- handling WRF packet input (sink)
- allocating memory pages using the MMU and writing packet data into the multiport memory
- initiating page transfers to the output blocks via the page transfer arbiter

Each input block shall also track the number of the pages allocated by itself, to manage the MPM buffer occupation and prevent buffer overflows and packet storm attacks. When the allocated page count is bigger than *[generic-configurable]* threshold, the IB shall:

- finish writing the current packet into the multiport memory (eventually allocating a few more pages),
- wait until the page count gets below the threshold,
- continue receiving and processing packets.

3.4. Output block

The role of the the output block is to:

- receive the page indices, DPMs and priorities from the page transfer arbiter,
- decode packet priority and put the page indices into the matching output queue,

- choose the queue with the highest priority containing at least one complete packet,
- read out the packet data from the multiport memory using the page indices from the output queue,
- output the packet using the WRF source,
- retransmit the packet in case of a WRF abort,
- drop the packet in case of a WRF error,
- release the pages used by the packet.

Note: It's not necessary for the first release of the Swcore to have multiple output queues.

3.5. Page transfer arbiter

The page transfer arbiter:

- waits for the requests from the input blocks and grants the transfers
- takes the page index sets for complete packets, associated with their DPMs and priorities
- passes the latter to the output blocks having ones in the DPM.

Appendix A – input block operation pseudocode

Tell me if this is useful – I will produce the pseudocodes (or even a full SV model) for all Swcore blocks.

```
module input_block(
    WRF_sink in,
    RTU_if rtu,
    MMU_if mmu,
    PTA_if pta,
    MEM_if pmem);

int allocated_page_count = 0;

bit [N-1:0] rtu_dpm;
bit3 rtu_prio;
bit rtu_drop;

int page_queue[MAX_PAGES];
int npages;


// check if there is a new packet on the WRF with its RTU decision and if we have enough pages
if(in.packet_present() && allocated_page_count < THRESHOLD && rtu.decision_present())
{
  // start the reception of the packet
  in.start_receiving_packet();

  // store the RTU decision and ack the RTU
  rtu_dpm = rtu.dpm;
  rtu_prio = rtu.prio;
  rtu_drop = rtu.drop;

  rtu.acknowledge();

  if(rtu_drop) // RTU decided to drop this packet?
  {
    in.drop_the_packet();
  }
```

```
  npages = 0;

} else { // no pages available? wait
  in.wait();
}

// MMU told us that one of the pages previously allocated by us was released by all output blocks
if(mmu.page_freed_strobe)
{
  allocated_page_count--;
}

// We are in the middle of packet reception
if(in.packet_in_progress())
{

  // request a page
  int page_idx = mmu.allocate_page();
  int n_received;

  bit16 data[PAGE_SIZE];
  bit4 ctrl[PAGE_SIZE];
 allocated_page_count++;

// set the write address and receive PAGE_SIZE packet words from the WRF
  pmem.set_write_page(page_idx);
  n_received = in.receive_words(data, ctrl, PAGE_SIZE);

// indicate the end of the packet if its size is not a multiple of PAGE_SIZE
  if(n_received < PAGE_SIZE)
    ctrl[n_received++] = TERMINATION_CODE; // indicate the end-of-packet by an unique value of ctrl code

  pmem.write({data,ctrl}, n_received);

// store the page index
  page_queue[npages++] = page_idx;
}

// we've reached the end of current packet?
if(in.end_of_packet())
{
// request the timeslot for the Page Tranfer Arbiter
  pta.request_timeslot();

// transfer the page index array, DPM and priority to the output blocks
  pta.transfer(page_queue, npages, rtu_dpm, rtu_prio);

// terminate the reception of the packet.
  in.next_packet();
}

endmodule
```