OpenNIC Technical Reference Guide

OpenNIC Shell

Coding Conventions

- Align and group code in a consistent way as much as possible.
- Create a helper module only if it feels natural to define its port signals. Otherwise, implement the needed functionality inside the module you are working on and properly group the code.
- Do not worry about line wrapping unless the line is super long (e.g., more than 120 characters).
- Add the proper prefix to signals belonging to a certain interface. For example, use “axil_” for AXI-lite and “axis_” for AXI-stream.
- Clock and reset signals should not be treated as part of the interface. Try to minimize the aliases for the same clock and reset.
- For master-slave-like protocols, module ports should indicate with a “m_” or “s_” prefix. Do not use any “m_” or “s_” prefix for internal signals. For example, use “m_axis_tready” if the module drives a master AXI-stream interface, and “axis_tready” for internal usage. ***To do:*** *replace the use of the terms “master” and “slave”.*
- Use a single always block for Moore machine.
- For Mealy machine, use one always block for state transition only and another always_comb (i.e., always @*) to do the variable update.
- Do not skip the “generate” and “endgenerate” keywords when writing a generate block. Label the block properly.

Top-level Parameters

The top-level module of the shell is open_nic_shell. It can be customized by the following parameters.

- BUILD_TIMESTAMP, a 32-bit word. The value will be stored in the read-only register 0x0 at PCI-e BAR-2. Use this parameter to distinguish different builds.
- MIN_PKT_LEN, integer between 64 and 256. It specifies the minimum packet size. The packet size calculation considers the Ethernet header, the payload and the 4-byte FCS. In other words, it should be equal to MTU + 4.
- MAX_PKT_LEN, integer between 256 and 9600. It specifies the maximum packet size. Set this size to the smallest possible value as there are several FIFOs along the data path whose depths are dependent on the maximal packet size.
- USE_PHYS_FUNC, 0 or 1. Setting this parameter to 0 means that we do not intend to use the streaming interface on the QDMA side. In this case, the design terminates the QDMA H2C and C2H interfaces at the boundary of box_250mhz.
- NUM_PHYS_FUNC, integer between 1 and 4. It specifies the number of PCI-e physical functions and consequently the number of qdma_subsystem_function blocks.
- NUM_QUEUE, integer between 1 and 2048. It specifies the number of QDMA queues inside the QDMA IP.
- NUM_CMAC_PORT, 1 or 2. It specifies the number of CMAC ports and thus the number of CMAC IP instances.

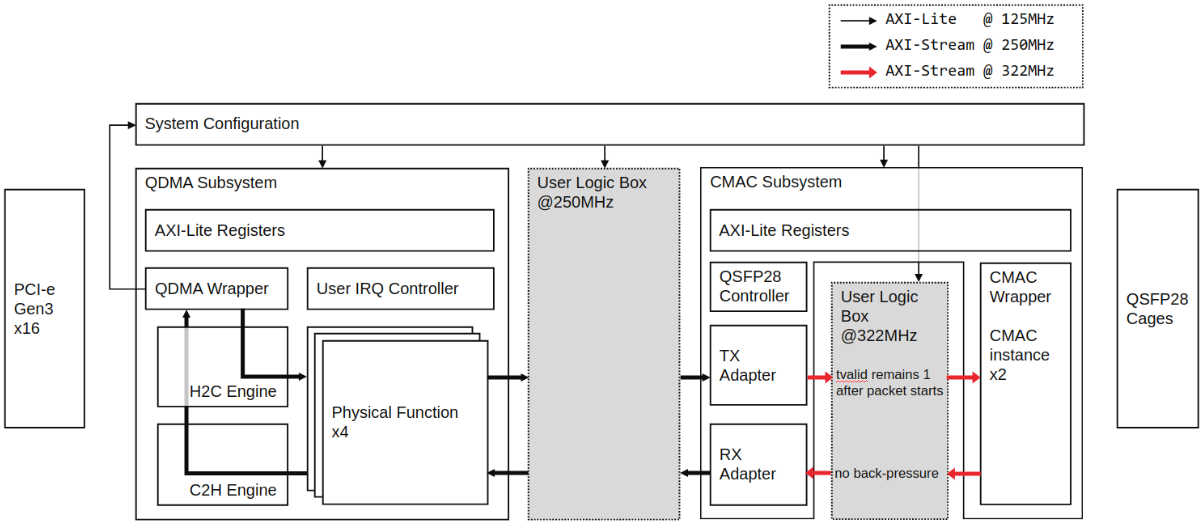
The module has its own parameter checkers and will report illegal values. If the constraints change in future releases, make sure to update the checkers as well.

Macros

A few macros are defined in “open_nic_shell_macros.vh”. In addition, the following macros are made available through the build script.

- “__synthesis__”.
- “__BOARD__”, where BOARD is one of the support boards.
- “__zynq_family__”, if zynq_family is set to 1 in “script/board_settings/BOARD.tcl”.

Shell Architecture



System Config

The system_config block runs at 125MHz. It implements the following functions:

- Define the system-level register address map and split the input AXI-lite interface into multiple ones, each fed into a subsystem block.
- Implement a few registers related to build timestamp and subsystem-level reset.

The block has two pairs of reset/reset_done signals. The signals shell_rstn/shell_rst_done are for shell-internal usage. The signals user_rstn/user_rst_done are for user logic boxes.

The slave AXI-lite interface comes from the QDMA subsystem, in which the 250MHz AXI-lite driven by the QDMA IP is converted to the 125MHz one. One of the master AXI-lite interfaces is fed back to the QDMA subsystem to control the non-IP registers.

The system-level register map can be found at system_config/system_config_address_map.sv:L18. The register within the system_config block can be found at system_config/system_config_register.v:L21.

QDMA Subsystem

The QDMA IP is instantiated in the QDMA subsystem. The H2C and C2H interfaces of the IP requires a lot of detail on how the IP has been configured. It is desirable to decouple these details from user logic implementations. To achieve this, the QDMA subsystem implements a few additional blocks, including qdma_subsystem_h2c, qdma_subsystem_c2h and qdma_subsystem_function, and provides standard AXI-stream interfaces to user logic.

Consider the blocks qdma_subsystem_h2c, qdma_subsystem_c2h and qdma_subsystem_function:

- The H2C block takes the IP H2C interface and converts it into several internal H2C interfaces, where the number equals how many physical functions the QDMA IP is configured to have.
- Similarly, the C2H block takes the same number of internal C2H interfaces and merges them into an IP C2H and an IP C2H completion interface.
- The qdma_subsystem_function block converts between the internal H2C/C2H interface and the user visible H2C/C2H interface. Each function block takes care of one pair of H2C/C2H interfaces. The number of function blocks is the same as the number of physical functions.

The idea behind this setup is as follows. The QDMA IP is queue-based. To correctly send a packet over DMA, one must know which queue has been configured within the IP. This is too detailed information for most users, who just want to send packets back to the same device. Hence, the queue information has been abstracted by the H2C/C2H and function blocks. Instead, a pair of queue-agnostic AXI-stream interfaces for each physical function is exposed to the user. To send a packet over DMA, one simply put the packet data on one of the C2H stream. The qdma_subsystem_function blocks will find a QDMA queue for the packet and convert it into the internal interface, which is then further converted to the IP C2H interface by the qdma_subsystem_c2h block.

Below are some implementation details for blocks in QDMA subsystem.

QDMA Wrapper

The qdma_subsystem_qdma_wrapper block instantiates the QDMA IP and exposes signals that will be used in the design. The IP outputs a single 250MHz clock. Logic interfacing with the IP is expected to run at this frequency. For the AXI-lite interface used for register access, running at 250MHz is unnecessary and poses more challenges for timing closure. Therefore, a phase-aligned 125MHz clock is created from the IP output 250MHz clock and the 250MHz AXI-lite interface was converted to 125MHz. The wrapper then feeds the 125MHz AXI-lite to the system_config subsystem where multiple AXI-lite interfaces are generated.

As of version 1.0, scatter-gathering (SG) DMA is not supported. But at the wrapper level, the descriptor bypass signals, which are required for SG DMA, are already exposed. They are tied to constant values for now.

QDMA Subsystem H2C

The H2C block takes the IP H2C interface as input. As of version 1.0, some of the input signals are simply dropped. These signals include:

- s_axis_qdma_h2c_tcrc,
- s_axis_qdma_h2c_tuser_port_id,
- s_axis_qdma_h2c_tuser_err, and
- s_axis_qdma_h2c_tuser_zero_byte.

For future releases, one task is to guard the outgoing stream with CRC check and packet error detection.

The s_axis_qdma_h2c_tuser_mdata is expected to contain the packet size (in bytes) in the lower 16 bits, which is filled in by the OpenNIC driver. The reason for doing that is because the user-visible interface contains a tuser_size signal which asks for a valid packet size being available when tvalid is asserted. If the s_axis_qdma_h2c_tuser_mty was used to compute the packet size, the entire packet would have to be buffered, causing higher latency and resource utilization. As of version 1.0, there is no check on whether the packet size indicated by tuser_mty matches the driver filled value.

There are NUM_PHYS_FUNC master interfaces. The m_axis_h2c_tready signals, each of which is driven by a qdma_subsystem_function block, reacts to m_axis_h2c_tuser_qid, which broadcasts the packet queue ID. The function block that accepts the queue ID will assert its tready. It is guaranteed that each queue ID maps to at most one function.

The H2C block reports the number of packets and bytes passed through and their function IDs.

QDMA Subsystem C2H

The C2H block takes NUM_PHYS_FUNC internal C2H interfaces as inputs. It uses a round-robin arbiter to merge the inputs into a single stream.

The QDMA IP can support several use cases by combining different modes of the C2H and C2H completion interfaces. In OpenNIC shell, only one mode is used: each C2H packet is accompanied by a C2H completion packet which contains some packet-level metadata. As a result, several signals are tied to constant values in the IP C2H interface, including

- m_axis_qdma_c2h_ctrl_marker = 1’b0,
- m_axis_qdma_c2h_ctrl_port_id = 0, and
- m_axis_qdma_c2h_ctrl_has_cmpt = 1’b1.

The C2H completion packet uses the same queue as its buddy packet does. It reports the queue ID, packet ID, which is globally increasing number for every C2H packet, and packet size. These metadata are extracted from the C2H interface and saved in the completion FIFO to guard back pressure on the C2H completion interface. The completion packet uses 8-byte data, i.e., m_axis_qdma_cpl_size = 2’b00. Thus, the upper 16 bytes of m_axis_qdma_cpl_tdata are not used and tied to 0. The mode is set to the regular mode, i.e., m_axis_qdma_cpl_ctrl_cmpt_type = 2’b11. Other signals tied to constant values in the completion interface include:

- m_axis_qdma_cpl_ctrl_no_wrb_marker = 1’b0,
- m_axis_qdma_cpl_ctrl_col_idx = 0,
- m_axis_qdma_cpl_ctrl_err_idx = 0,
- m_axis_qdma_cpl_ctrl_marker = 1’b0,
- m_axis_qdma_cpl_ctrl_user_trig = 1’b0, and

- m_axis_qdma_cpl_ctrl_port_id = 0.

Like the H2C block, the C2H block also reports the number of packets and bytes passed through and their function IDs.

Note that the acronyms “cpl”, “cmpt” and “cmpl” all refer to “completion”. The first form is used in the OpenNIC driver and brought into the shell for consistency. The second is from the QDMA IP. The last can be found in Xilinx DMA drivers (not in OpenNIC drivers).

QDMA Subsystem Function

As of version 1.0, the qdma_subsystem_function blocks have a one-one correspondence with the PCI-e physical functions configured in the QDMA IP. Since each physical function maps to a net device in Linux kernel, the function blocks become the natural candidate for the implementation of network function offloading, such as TSO, checksum offloading and RSS.

In OpenNIC, there are 3 types of queues,

1. queues used in a net device, i.e., multi-queue net device,
2. queues in QDMA IP, and
3. queues bound to a qdma_subsystem_function block.

Type 1 and 3 are similar in the sense that both are local to the net device/function and indexed from 0. Type 2 is global as there is only one QDMA instance regardless of the number of functions. Each QDMA queue is either not used or assigned to a particular physical function. As of version 1.0, there are no SRIOV-related functionalities. All three types of queues are one-one mapped.

The function block needs to be configured before sending and receiving any packet. The configuration binds several QDMA queues to the function, where the binding usually comes from the initialization of a net device in the host driver. In the implementation, when the driver initializes the software context of the QDMA queues, the same configurations are written to the qdma_subsystem_function_register block associated with the function.

The function operates in both H2C and C2H directions. In the H2C direction, it takes a packet and its queue ID and outputs the packet if the queue ID binds to the function. The statement is trivial at first glance. But recall that the queue ID triggers the assertion of tready. If a function does not own the queue, it will simply ignore the packet. As of version 1.0, TSO and checksum offloading has not been implemented.

In the C2H direction, the function takes a packet from user logic and tries to select a queue for it. The queue selection process, commonly known as receiving-side scaling (RSS), generates a hash using certain fields of the packet and does a lookup using the lower bits of the hash. The hash key and the lookup table, known as the indirection table, are software configurable and can be controlled using ethtool with straightforward driver support. As of version 1.0, we have not implemented the support.

The hashing part is implemented in qdma_subsystem_hash. With an input key, it computes the hash value for input packets using the Toeplitz function. As of version 1.0, VLAN packets are not supported.

CMAC Subsystem

Unlike the QDMA subsystem, there can be two CMAC subsystem instances in OpenNIC shell. Each instance is parameterized with a CMAC ID (either 0 or 1) and instantiates a CMAC IP accordingly. The CMAC IP is wrapped in the cmac_subsystem_cmac_wrapper block, which has a 322MHz clock for data path. The AXI-lite interface runs at 125MHz.

Another component in the CMAC subsystem is the QSFP28-related registers. Its implementation should control the I2C registers and the side-band signals for the QSFP28 cages. As of version 1.0, register space is reserved for this part, but the functionality is not implemented.

Besides running at a different frequency, the AXI-stream interfaces for CMAC subsystem are also more restrictive than those for QDMA subsystem. In the RX direction, there is no m_axis_rx_tready signal. Therefore, the data cannot be back pressured. In the TX direction, s_axis_tx_tvalid is supposed to be asserted once a packet starts. In other words, deasserting s_axis_tx_tvalid is not allowed before s_axis_tx_tlast is asserted, even if s_axis_tx_tready is pulled low. This restriction, coming from the CMAC IP, essentially requires packets to be buffered first and then sent to CMAC subsystem.

Packet Adapter

The packet adapter serves as a bridge between the 250MHz QDMA-oriented and the 322MHz CMAC-oriented clock domain. It also converts between different AXI-stream data interfaces on both sides. For each CMAC subsystem, there is a corresponding instance of the packet adapter.

In the TX direction, the packet_adapter_tx block first checks if the packet destination matches its CMAC ID and drops any mismatched packets. It then buffers an incoming packet in a packet-mode FIFO and outputs only when the packet is complete. It guarantees that the packet meets the data interface requirement of the CMAC subsystem. The TX block has a real-valued parameter PKT_CAP, which has a default of 1.5. It indicates the packet FIFO should be large enough to fit at least PKT_CAP packets of MAX_PKT_LEN bytes. See packet_adapter/packet_adapter_tx.sv:L51 for the computation.

In the RX direction, the packet_adapter_rx block also buffers packets. It drops a packet when

- the packet has its s_axis_rx_tuser_err bit asserted in the last beat, or
- the FIFO becomes full.

A difference from the TX block is that, in the RX block, the drop decision can be made anytime. It may need to discard some packet data that has been written into the FIFO. To cope with this, a utility block axi_stream_packet_buffer was implemented. Compared with an axi_stream_packet_fifo block, the packet buffer has two additional signals, drop and drop_busy. Asserting the drop signal causes the last partial packet to be discarded from the internal FIFO. Like the TX block, the RX block has a PKT_CAP parameter defaulted to 1.5 as well.

User Logic Box and User Plugins

A user logic box is where user plugins are instantiated. There are two user logic boxes in the OpenNIC shell, one running at 250MHz sitting between the QDMA subsystem and the packet adapters, and the other running at 322MHz sitting between the CMAC subsystem(s) and the packet adapters. Each box has its own control and data interfaces. The control interface uses a 125MHz AXI-lite interface generated from the system_config block. The data interfaces are variants of AXI-stream and have different constraints in different boxes, which is explained in detail later.

The number of data interfaces in each user logic box depends on top-level parameters. For the 250MHz box, there are NUM_PHYS_FUNC (up to 4) pairs of AXI-stream interfaces on the QDMA subsystem side, and NUM_CMAC_PORT (up to 2) pairs on the packet adapter (i.e., CMAC subsystem) side. For the 322MHz box, on both sides there are NUM_CMAC_PORT (up to 2) pairs of AXI-stream interfaces. Also, if USE_PHYS_FUNC is set to 0, the 250MHz box terminates the data interfaces on the QDMA subsystem side.

Each box has a predefined set of ports as listed below.

- AXI-lite interface
- A 125MHz clock for the AXI-lite interface
- Master/slave AXI-stream interfaces for TX and RX directions
- A clock for the AXI-stream interfaces running at either 250MHz or 322MHz
- Box reset: box_rstn and box_rst_done
- Reset for individual user blocks: mod_rstn and mod_rst_done

Users can implement any logic using the provided interfaces except for the box reset, which is reserved for box-internal usage. There are two include statements in the box implementation:

- The first one is for multiplexing the input AXI-lite interface so that every user block has its own register space. See box_250mhz/box_250mhz.sv:L101 and box_322mhz/box_322mhz.sv:L92.
- The second one is for a proper instantiation of the user plugin. At the minimum, the included file should check the parameters, tie off unused mod_rst_done signals, and instantiate the user plugin top-level. See box_250mhz/box_250mhz.sv:L117 and box_322mhz/box_322mhz.sv:L93.

In each box, a single plugin top-level is expected, which should terminate the generated AXI-lite and the TX/RX AXI-stream interfaces. The included files provide the glue code to connect the plugin top-level with the box. To generate this glue code, users need to decide the register address for their plugins and the number of reset signals. As of version 1.0, the glue code generation has not been automated.

Control and Data Interfaces

OpenNIC shell and user logic boxes communicate through three types of interfaces.

- AXI-lite interface running at 125MHz for register access.
- AXI-stream interface running at either 250MHz or 322MHz for data path.
- Synchronous reset interface running at 125MHz.

The 125MHz and 250MHz clock domains are phase aligned. Signals can be sampled across each domain without clock domain crossing. On the other hand, note that different clock frequencies can lead to double sampling or missing samples.

The AXI-lite interface follows the standard AXI4-lite protocol without wstrb, awprot and arprot. At the system level, the address ranges for the 2 boxes are

- 0x10000 - 0x3FFFF for the 322MHz box, and
- 0x40000 - 0xFFFFF for the 250MHz box.

The 250MHz and 322MHz AXI-stream interfaces have slightly different semantics. The 250MHz interface has the following signals.

- tvalid, 1 bit: same as standard AXI4-stream protocol.
- tdata, 512 bits: data maps from lower to upper bytes.
- tkeep, 64 bits: null bytes are only allowed when both tvalid and tlast are asserted and cannot be followed by other data bytes for the same packet. For example, a 96B packet has a tkeep value of 0xFFFFFFFFFFFFFFFF in the first beat, and 0x00000000FFFFFFFF in the second beat.
- tlast, 1 bit: same as standard AXI4-stream protocol.
- tuser_size, 16 bits: field to specify packet size. It contains the number of bytes in the packet and must remain valid and unchanged if tvalid is asserted.
- tuser_src, 16 bits: source of the packet.
- tuser_dst, 16 bits: destination of the packet.
- tuser_user, N bits: side-band user data.
- tready, 1 bit: same as standard AXI4-stream protocol.

For the tkeep signal, the interface assumes its presence depending on the direction of a packet.

- For packets exiting the shell, it is guaranteed that tkeep is consistent with tuser_size. This means that tkeep consists of all 1s when tvalid is asserted and tlast is de-asserted and shows a bit-mask for the valid bytes in the beat when both tvalid and tlast are asserted.
- For packets entering the shell, tkeep can be optionally set to all 1s regardless of the value of tuser_size. This allows user plugins to drop tkeep in their implementation. If tkeep is not set to all 1s, it must be consistent with tuser_size.

For tuser_src and tuser_dst, the following format is used. For packets exiting the shell, tuser_src is marked accordingly and tuser_dst is all 0s. Note that although there are at most two CMAC instances, 10 bits are used for MAC ports for possible break-out extension.

15									6			3			0
P	P	P	P	P	P	P	P	P	P	R	R	F	F	F	F
MAC ports										Reserved		PCI-e PFs			

The 322MHz interface is more restrictive due to the CMAC IP constraints. It has the following signals:

- tvalid, 1 bit: no de-assertion in the middle of a packet. In other words, once tvalid is asserted to indicate the start of packet, it must remain asserted until tlast.
- tdata, 512 bits: same as in the 250MHz interface.
- tkeep, 64 bits: same as in the 250MHz interface.
- tlast, 1 bit: same as in the 250MHz interface.
- tuser_err, 1 bit: indicates if the packet contains an error.
- tready, 1 bit: same as in the 250MHz interface. For packets entering the shell, i.e., from the RX side of CMAC IPs, tready is not present and the master assumes that tready is always asserted.

Utility Blocks

There are several utility blocks shipped with OpenNIC shell.

AXI-lite Register

The axi_lite_register block converts the AXI-lite interface to a simple BRAM-like read/write interface. It can be configured to use the same or different clocks for each interface. The AXI-lite interface is assumed to have a single transaction at one time, either a read or a write transaction but not both.

AXI-lite Slave

The axi_lite_slave block is a sink for unused AXI-lite interfaces. A common use case is when multiple AXI-lite interfaces are derived from an AXI-lite crossbar, we may not implement all of them at the same time. To make sure that those unimplemented ones do not get stuck, they can be terminated with axi_lite_slave blocks. Additionally, the axi_lite_slave block accepts two parameters: register address width and a prefix which fills the upper 16 bits of covered registers.

AXI-stream Register Slice

The axi_stream_register_slice block is used to decouple the slave and master AXI-stream interfaces. It supports two modes: forward and full mode. The forward mode guarantees that the output ports of the master interface are directly driven by flip-flops. The master tready signal, on the contrary, feeds backwards to the slave side using a combinational path. It is recommended to use the forward mode only for providing one-cycle delay from the slave interface. The forward mode block should not be cascaded, which could lead to nasty timing issues.

The full mode guarantees that all the output ports, including the slave tready signal, are directly driven by flip-flops. Therefore, it can be used to fully decouple the slave and master interfaces. Note that the delay between master and slave interfaces can be more than one cycle.

AXI-stream Packet FIFO

The axi_stream_packet_fifo is a replacement for xpm_fifo_axis in the packet mode, which seems to have certain bugs. Use this block only when a packet-mode FIFO is required.

AXI-stream Packet Buffer

The axi_stream_packet_buffer enhances axi_stream_packet_fifo with a packet dropping feature. It has two signals, drop (input) and drop_busy (output). When drop is asserted, the input packet is completely dropped and will not be observed from the master interface. To initiate it, the drop signal should be asserted for a single cycle. The drop_busy signal shows the status of the packet buffer.

Another feature of this block is that it reports the packet size in bytes using m_axis_tuser_size. The m_axis_tuser_size signal is valid as long as m_axis_tvalid is asserted.

The axi_stream_packet_buffer block supports both common_clock and independent_clock mode. It also accepts a PKT_CAP parameter, which can be used to tune the depth of the internal FIFO.

AXI-stream Size Counter

The axi_stream_size_counter block calculates the size of the input packets. It does so by passively monitoring the input packets (notice the “p_” prefix of the AXI-stream signals). The size_valid signal is asserted one cycle after the last word of the packet.

Generic Reset

The generic_reset block takes a reset pair (i.e., the mod_rstn and mod_rst_done signals) and generates a configurable number of reset signals that are asserted for multiple cycles in different clock domains. There are three parameters.

- NUM_INPUT_CLK. Specifies the number of input clocks and long-asserted output reset pins. When larger than 1, the first clock should be the slowest one.
- SAMPLE_CLK_INDEX. Specifies which input clock is used to sample and drive the reset pair when NUM_INPUT_CLK is larger than 1.
- RESET_DURATION. Specifies how long the reset pins are asserted. The duration is in terms of clock cycles with respect to the first input clock, which is assumed to the slowest one when NUM_INPUT_CLK is larger than 1.

This block is used to convert a reset pair into long-asserted reset signals. If a custom reset sequence is desirable, one can either modify the generic reset block or roll out a specific implementation.

Level-trigger CDC

The level_trigger_cdc block synchronizes a single-bit level trigger and optionally a set of data across two different clock domains. It is guaranteed that the pulse generated in the source clock domain is propagated to the destination clock domain with the same width. A common use case is to synchronize a counter increment signal from a different clock domain.

The implementation uses an asynchronous FIFO. The depth should be large enough that no event in the source domain will be missed. When src_valid is asserted while the FIFO is full, src_miss will be asserted indicating the event is missed.

Round-robin Arbiter

The rr_arbiter block implements a round-robin arbiter. A request from channel k comes when req[k] is asserted. It should keep asserted until the requested transaction is done. The transaction starts when grant[k] is asserted, which could happen in the same cycle as the assertion of req[k].

Once both req[k] and grant[k] are asserted, the arbiter allocates the resource to channel k, and grant[k] remains high until a single-cycle fin[k] is received. The arbiter then releases the resource and starts another round of allocation in the next cycle.

A request cannot be canceled once its req bit is asserted.

Resource is granted to a particular channel based on two criteria: 1) first-come-first-served (FCFS), and 2) least recently used (LRU). When a request arrives, it will be queued if the resource is temporarily unavailable. When multiple requests arrive at the same time, the LRU channel gets the resource if available, and others are pushed into the queue. The queue tracks the relative arrival time, so that queued requests that arrived at the same time compete solely based on the LRU principle.

Resource grant is guarded by the ready signal. When ready de-asserts, no grant will be given.

Register Layout

Those marked with in red are not yet implemented in version 1.0.

System-level Address	**Local offset**	**Mode**	**Description**
System Config: 0x0000 - 0x00FF			
0x0000	0x0000	RO	Build timestamp This register uses the value of the parameter BUILD_TIMESTAMP, which can be customized from the build command line.
0x0004	0x0004	WO	System reset Writing to this register initiates a system-level reset, which lasts until, all the submodules are out of reset. A system-level reset is also done on power up.
0x0008	0x0008	RO	System status 31:1 - reserved 0 - system reset done
0x000C	0x000C	WO	Shell reset 31:3 - reserved, 2 - reset for CMAC subsystem 1 1 - reset for CMAC subsystem 0 0 - reset for QDMA subsystem Writing 1 to a bit of this register initiates a submodule-level reset, which lasts until the submodule is out of reset.
0x0010	0x0010	RO	Shell status 31:3 - reserved, 2 - CMAC subsystem 1 reset done 1 - CMAC subsystem 0 reset done 0 - QDMA subsystem reset done

0x0014	0x0014	WO	User logic reset Writing 1 to a bit of this register initiates a reset on the corresponding reset pin, which lasts until the reset done pin is asserted. It is up to the user on how to connect the reset pairs to the user logic blocks.
0x0018	0x0018	RO	User logic status Each bit, if used, shows the reset status of the corresponding user logic block.
QDMA Subsystem: 0x1000 - 0x5FFF			
	Function 0: 0x0000 – 0x0FFF Function 1: 0x1000 – 0x1FFF Function 2: 0x2000 – 0x2FFF Function 3: 0x3000 – 0x3FFF System-level address is only shown for Function 0. Add an offset of 0x1000, 0x2000 and 0x3000 for Function 1, 2 and 3 registers respectively.		
0x1000	0x0000	RW	Queue configuration 31:16 - base queue ID 15:0 - number of queues
0x1400 - 0x15FF	0x0400 - 0x05FF	RW	RSS indirection table 31:16 - reserved 15:0 - queue ID at index
0x1600 - 0x1627	0x0600 - 0x0627	RW	RSS hash key
0x1800 0x1804	0x0800 0x0804	RO	TX packets from function
0x1808 0x180C	0x0808 0x080C	RO	TX bytes from function
0x1900 0x1904	0x0900 0x0904	RO	RX packets into function
0x1908 0x190C	0x0908 0x090C	RO	RX bytes into function
	QDMA subsystem registers: 0x4000 - 0x4FFF		
0x5000 0x5004	0x0000 0x0004	RO	TX packets from QDMA
0x5008 0x500C	0x0008 0x000C	RO	TX bytes from QDMA
0x5100 0x5104	0x0100 0x0104	RO	RX packets into QDMA
0x5108 0x510C	0x0108 0x010C	RO	RX bytes into QDMA
CMAC Subsystem - **CMAC 0: 0x8000 – 0xAFFF**			

- CMAC 1: 0xC000 – 0xEFFF **System-level addresses are only shown for CMAC 0. Add an offset of 0x4000 for CMAC 1 registers.**			
0x8000 - 0x9FFF	0x0000 - 0x1FFF		CMAC IP registers
0xA000 - 0xAFFF	0x2000 - 0x2FFF		QSFP28 registers
Packet Adapter **- Packet adapter 0: 0xB000 – 0xBFFF** **- Packet adapter 1: 0xF000 – 0xFFFF** **System-level addresses are only shown for packet adapter 0. Add an offset of 0x4000 for packet adapter 1 registers.**			
0xB000 0xB004	0x0000 0x0004	RO	TX packets sent
0xB008 0xB00C	0x0008 0x000C	RO	TX bytes sent
0xB010 0xB014	0x0010 0x0014	RO	TX packets dropped
0xB018 0xB01C	0x0018 0x001C	RO	TX bytes dropped
0xB020 0xB024	0x0020 0x0024	RO	RX packets received
0xB028 0xB02C	0x0028 0x002C	RO	RX bytes received
0xB030 0xB034	0x0030 0x0034	RO	RX packets dropped
0xB038 0xB03C	0x0038 0x003C	RO	RX bytes dropped
0xB040 0xB044	0x0040 0x0044	RO	RX packets with error
0xB048 0xB04C	0x0048 0x004C	RO	RX bytes with error
User Logic Box @ 322MHz: 0x10000 - 0x3FFFF			
User Logic Box @ 250MHz: 0x40000 - 0xFFFFF			

Working with OpenNIC Shell

Here a few techniques and common issues when working with OpenNIC shell are discussed.

Timing Closure

With the default user plugins (i.e., the p2p plugins), OpenNIC shell has been tuned so that it can achieve timing closure using the Vivado default implementation strategy. With custom plugins, however, it can be challenging to close timing. Here are a few common techniques to analyze and fix timing issues.

Whenever there is a timing violation, one needs first to understand what might be the root cause. A timing violation can be

- inter-clock, or
- intra-clock.

For the former case, mostly it is because there are not good enough constraints between the two clock domains. Setting false paths between the two clock domains is usually not the best solution, as it basically asks the timing engine not to balance the latency of such paths at all. It is generally preferable to set a reasonable maximum delay between the two clock domains. This ensures that any related path has a bounded latency.

When the maximum delay constraints are already in place, it is still possible to have inter-clock violations. It can be due to a too small delay value, or that the timing budget for intra-clock logic is too tight. In the latter case, usually a bunch of intra-clock violations occur at the same time, which should be fixed before trying to fix the inter-clock violations.

The intra-clock violations have two major types. The first type is due to long combinational paths. For this type of failing path, one can observe a large number of logic levels. In this case, what to do is to reduce the logic levels as much as possible, by either inserting registers or changing logic implementations. Note that there is no specific rule on how many levels is considered "large". It depends on the frequency and the inherent logic delay of LUTs. Generally speaking, for Alveo-family boards running at 250MHz, 10 levels or more is considered reasonably large.

The second type of intra-clock violation features large net latency. Long wires can be observed in the device chart. It happens because the timing budget of the failed path has been squeezed by the rest of the logic. It implies that the root cause could be in the neighboring logic even though there is no timing issue in there. One needs to analyze our implementation around the failed path and see which part might be too aggressive.

A good tool for timing closure is pblocks. From an OpenNIC shell perspective, it is very desirable to minimize SLR crossings. Using pblocks to constrain certain RTL blocks inside a single SLR can be critical to efficient timing closure.

Trouble-shooting Runtime Issues

Runtime issues can be debugged with the help of ILA. To use ILA, add the attribute

```
(* MARK_DEBUG = “true” *)
```

to the desired signals and setup debug cores after synthesis. Generally speaking, good places to mark debug signals include the QDMA IP interfaces, the boundaries of user logic boxes, and the CMAC IP interfaces. In many cases, it is not necessary to debug pure data signals, such as axis_tdata.

The built-in registers are also helpful. For example, registers in the packet adapters can be used to identify whether there is any packet drop in the RX path.

Simulating OpenNIC Shell

Older version of OpenNIC shell (prior to 1.0) came with a simulation library for everything excluding the QDMA and CMAC IPs. It exposed AXI-stream and AXI-lite interfaces and connected the simulator into the Linux kernel network stack. This library was temporarily removed from version 1.0 due to some incompatibilities with the new user plugin scheme. In detail, the simulation flow does not know how to handle IPs in a user plugin since OpenNIC shell does not manage the directory structure of the plugin. Consequently, the simulation code for the IP needs to be manually generated, which could introduce many sorts of compatibility issues. The plan is to further refine the newly introduced plugin scheme in a future version and bring back the simulation capabilities.

OpenNIC Driver

OpenNIC driver implements a lightweight Linux kernel driver for OpenNIC shell. It is different from the Xilinx QDMA Linux kernel driver in the following two respects:

- OpenNIC driver is a net device driver which connects the low-level device to Linux kernel network stack. It manages not only QDMA, but also CMAC and other logic inside the shell. The Xilinx QDMA driver works like a data mover between host and FPGA memory.
- OpenNIC driver implements a small subset of QDMA capabilities. For example, it only supports queues in streaming mode and has fixed arrangement for interrupts. By contrast, the Xilinx QDMA driver provides a lot of QDMA configuration options to users.

OpenNIC driver is divided into:

- a hardware access layer which implements register access to QDMA and other hardware blocks,
- a network stack layer which implements callbacks requried by the stack, and
- an application layer which deals with non-packet data generated from user logic.

To a large extent, OpenNIC driver shares the same hardware access layer for the QDMA part as the Xilinx QDMA driver except for some stylistic differences.