LL10GEMAC-IP with AAT QDMA Demo Reference Design Manual

LL10GEMAC-IP with AAT QDMA Demo Reference Design Manual

Rev1.00 21-Jan-2025

1 Introduction

High-Frequency Trading (HFT) is an advanced form of algorithmic trading where sophisticated computer programs execute a large number of orders at extremely high speeds, often within microseconds. This approach capitalizes on minute price discrepancies in financial markets, aiming to generate profits from rapid, short-term market movements.

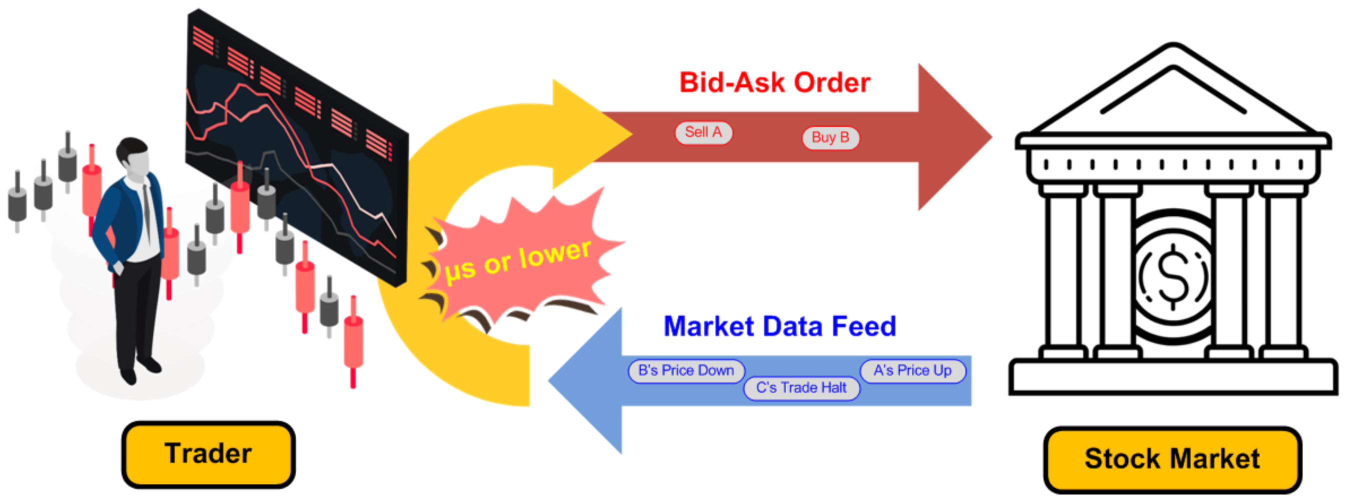


Figure 1 High-Frequency Trading (HFT) Scheme

The competitive landscape of HFT is predominantly defined by latency - the time delay between the initiation of a market event and the corresponding trading response. In this domain, even nanosecond-level advantages can significantly impact profitability. Consequently, HFT firms invest heavily in ultra-low-latency technologies to process market data and execute trades faster than their competitors.

Success in HFT hinges on the ability to swiftly analyze incoming market data and promptly submit bid-ask orders. Traders who can reduce latency in these processes are more likely to secure favorable trade positions, thereby enhancing their potential for substantial profits. This relentless pursuit of speed underscores the critical importance of low-latency systems in the HFT industry.

For maintaining a competitive edge in the trader system, various system architectures have been developed as shown in Figure 2.

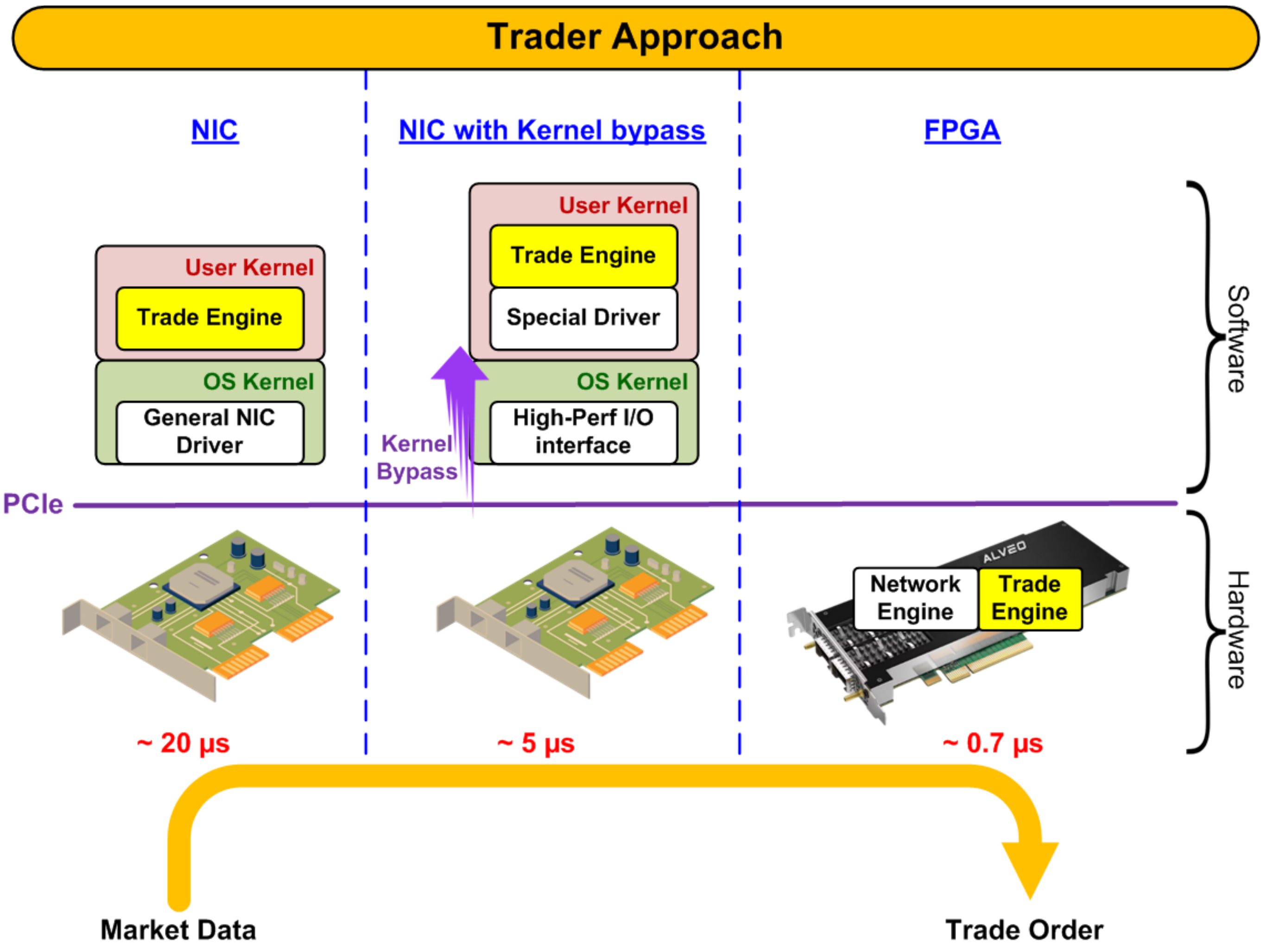


Figure 2 Trader System Approach for HFT

The first solution (most left) is Network Interface Card (NIC) with Software Trading Engine. This conventional approach utilizes a standard NIC to handle network tasks, allowing the CPU to focus on executing the trading algorithms. However, the reliance on the operating system's kernel and networking stack introduces inherent latencies. In HFT environments, this configuration typically results in response times of approximately 20 microseconds or higher.

To reduce more latency, Kernel Bypass concept (middle) is integrated to the first approach (NIC with Software Trading Engine). This method employs specialized NICs capable of bypassing the operating system's kernel, thereby minimizing the overhead associated with kernel-level processing. By leveraging user-space drivers and high-speed I/O interfaces, this architecture can achieve response latencies around 5 microseconds. Implementations often utilize technologies such as Data Plane Development Kit (DPDK) to facilitate kernel bypass and enhance performance.

The most advanced solution integrates the entire trading system within an FPGA (most right), encompassing both network processing and trading logic. By offloading critical tasks to FPGA logic and utilizing on-chip processors for complex algorithms, this architecture achieves minimal latency, often as low as 700 nanoseconds. The hardware-level processing inherent in FPGAs enables rapid data handling and decision-making, providing a significant advantage in time-sensitive trading scenarios.

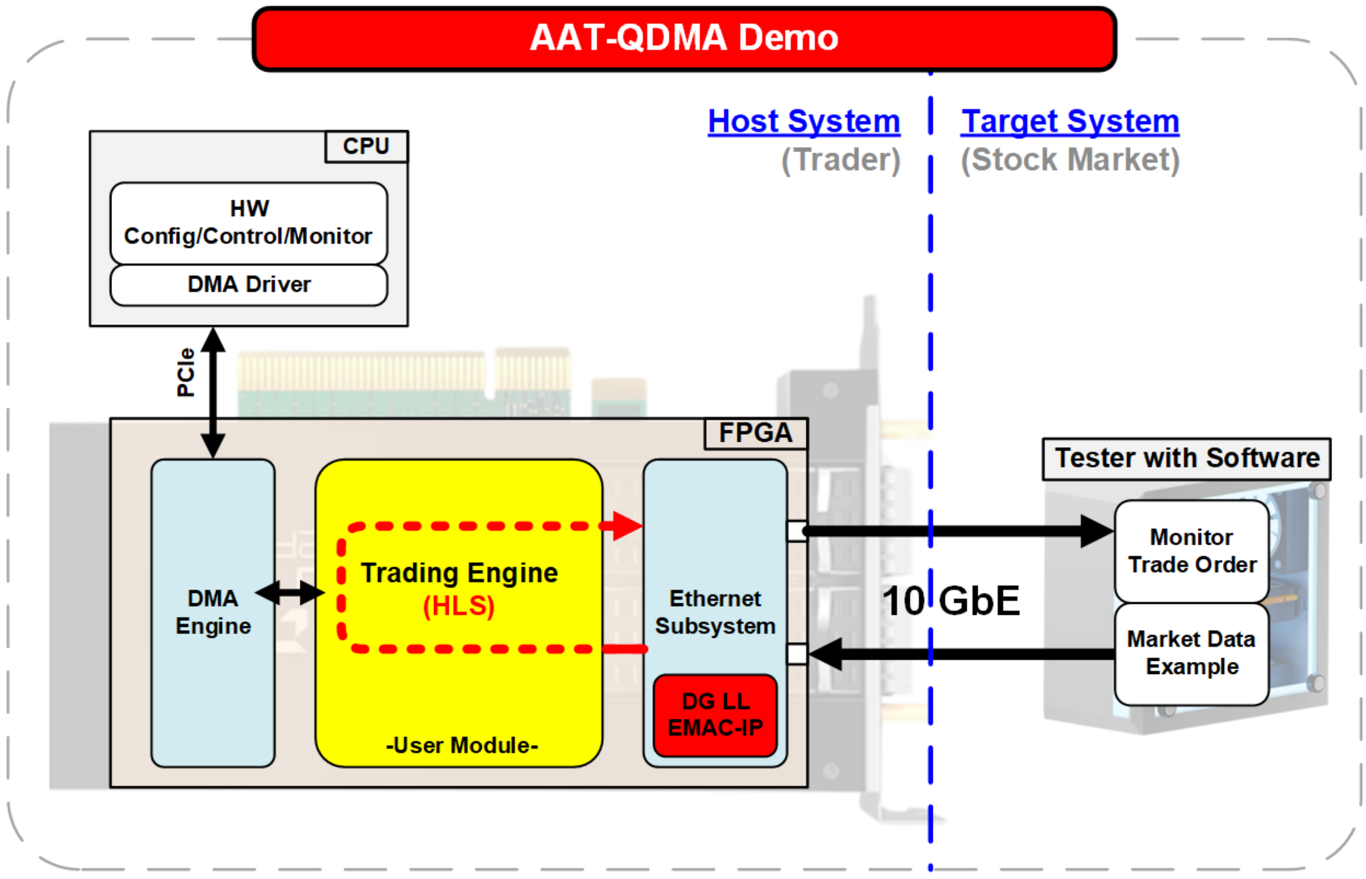


Figure 3 Overview of Accelerated Algorithmic Trading with QDMA Demo (AAT-QDMA)

Building upon the FPGA-based approach mentioned earlier, Design Gateway presents the Accelerated Algorithmic Trading with QDMA (AAT-QDMA) demo, which demonstrates a high-performance trading system over 10G Ethernet. This demonstration consists of two primary components: Host and Target.

The Host serves as the trader's platform and consists of a PC or server equipped with an FPGA card. The entire trading system is implemented within the FPGA, which is designed with three key modules:

- **Ethernet Subsystem** (Network Engine): Incorporates Design Gateway's Low-Latency 10G Ethernet MAC IP (LL10GEMAC-IP) to minimize network-layer latency, ensuring rapid data transmission.
- **Trading Engine** (User Module): Developed using High-Level Synthesis (HLS), this module implements trading algorithms and strategies tailored to specific market protocols, allowing for customization with high-level programming languages.
- **DMA Engine**: Facilitates efficient communication between the FPGA logic and the Host CPU via PCI Express (PCIe), enabling high-speed data transfer and control.

The Host CPU's role in this demo is primarily confined to configuration, status monitoring, and hardware control through the PCIe interface. The trading algorithm is fully integrated into the FPGA logic to achieve minimal latency. However, for more complex scenarios such as risk management or advanced computations, the Host CPU can process these tasks. In such cases, high-speed data transfer between the FPGA and CPU is critical, facilitated by the DMA engine over PCIe interface. This enables real-time exchange of the order book data and computational results.

The Target acts as a testing platform, simulating the role of a Stock Market. It runs dedicated software that performs two essential functions: sending sample market data to the Host and receiving Bid-Ask orders generated by the FPGA.

This interaction demonstrates a fully functional FPGA-based trading system, showcasing the high-speed order generation and response process.

The AAT-QDMA demo delivers notable benefits to trading system developers and traders:

- Reduced development time and faster time-to-market: By providing a complete framework with pre-validated DMA and Ethernet engines, the demo enables traders to focus on developing and customizing the user module within the FPGA. The use of High-Level Synthesis (HLS) simplifies the development process, allowing for algorithm implementation using high-level programming languages and facilitating integration with specific stock market protocols.
- Adaptability and portability: While the AAT-QDMA demo is initially demonstrated on AMD Alveo FPGA cards, its development using the Vivado platform ensures portability across various FPGA boards. This flexibility allows traders to seamlessly adapt the system to their preferred hardware platforms.
- Low-Latency performance: By integrating Design Gateway's LL10GEMAC-IP core, the AAT-QDMA demo delivers a robust FPGA-based trading system capable of achieving the low-latency performance essential for high-frequency trading (HFT) applications.

The subsequent sections provide a detailed description of the Host and Target components, offering further insights into the architecture and capabilities of the AAT-QDMA system.

2 Host Hardware

The AAT-QDMA framework connects the Host and Target systems through four 10G Ethernet channels. Two channels handle UDP Market Data from the Target, one channel manages TCP Bid-Ask Orders from the Host, and the remaining channel is reserved for loopback testing to measure latency. However, a minimum of two Ethernet channels is required to run the demo—one for UDP Market Data and one for TCP Orders. The Host system is primarily implemented on an FPGA card, while the Target system uses a simple PC running test software that interfaces with the Ethernet network via a standard Network Interface Card (NIC).

This section introduces two hardware configurations: Pricing Engine on Card and Pricing Engine Software. In the first configuration, the entire trading engine is implemented within the FPGA logic, with the Host CPU responsible only for system configuration and parameter control, requiring minimal data transfer. In the second configuration, the FPGA logic acts as an accelerator, working alongside the CPU software. This approach involves significant data transfer between the CPU and FPGA logic to enable real-time processing. Further details for each hardware system are described in the following sections.

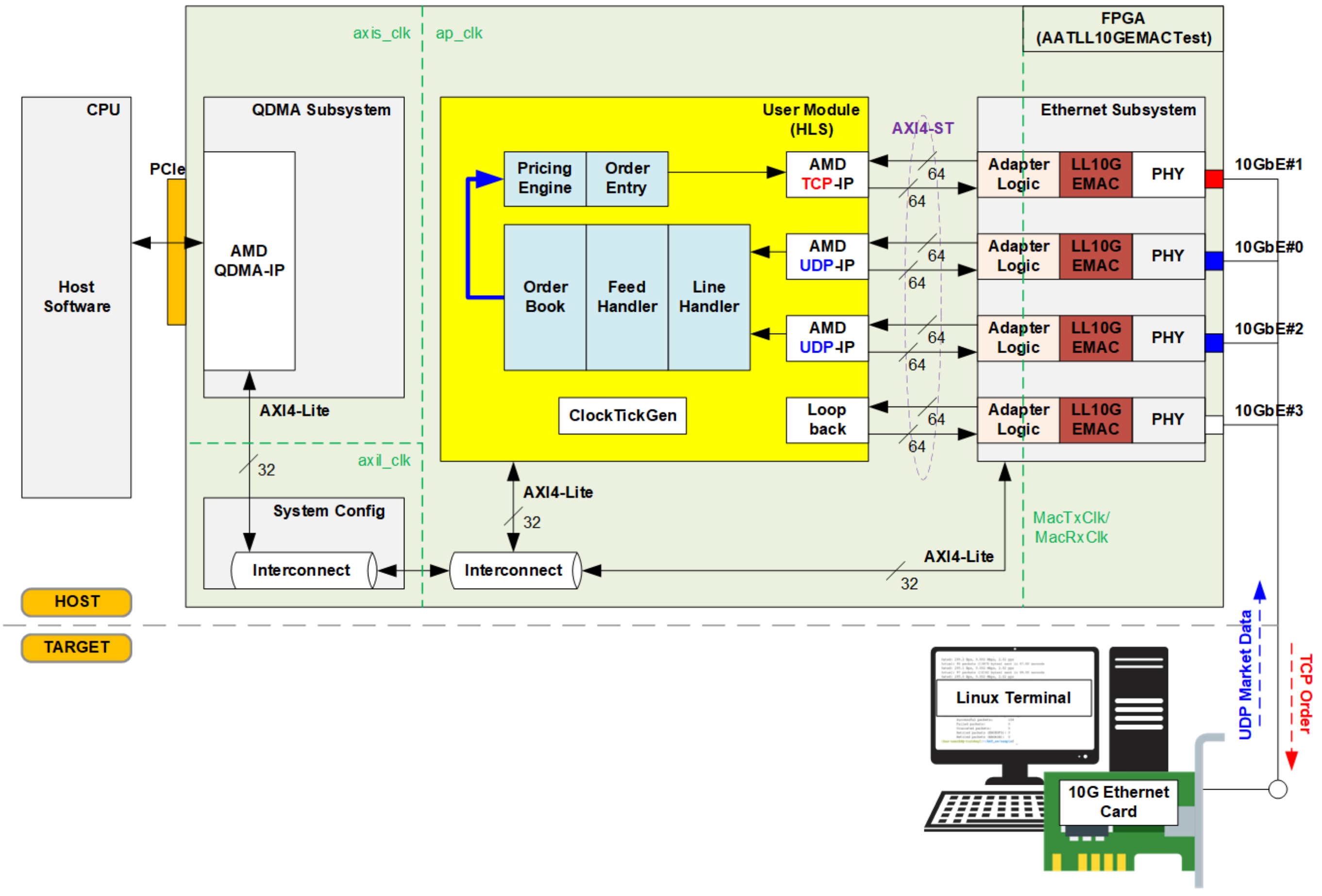


Figure 4 Host Hardware on AAT-QDMA Demo (Pricing Engine on Card)

Figure 4 illustrates the Host hardware architecture, which is entirely implemented in FPGA logic. This architecture comprises four main hardware blocks. The first block is the Ethernet Subsystem, responsible for handling low-layer networking across four Ethernet channels. It integrates Design Gateway's Low-Latency 10G Ethernet MAC IP Core (LL10GEMAC), which minimizes network latency in conjunction with AMD/Xilinx PHY. Design Gateway's adapter logic facilitates data width conversion and clock domain crossing between LL10GEMAC and the upper layers.

The second hardware block is the User Module, which is allocated for the user's trading engine. This region is developed entirely using High-Level Synthesis (HLS), ensuring adaptability for different trading markets. AAT-QDMA implements this region based on AMD/Xilinx reference designs, incorporating a loopback module, TCP/UDP IP cores for upper-layer networking, and several trading sub-blocks (depicted as blue blocks), such as the Line Handler, Feed Handler, and Order Book.

Communication between the FPGA card and the Host CPU occurs via a PCI Express slot. To facilitate this interaction, the QDMA Subsystem is included in the Host hardware, primarily leveraging AMD's QDMA IP Core. This subsystem is typically used for hardware register access, enabling the Host CPU to control, configure, and monitor the FPGA sub-blocks. Each hardware submodule is equipped with an AXI4-Lite interface for efficient register access.

The final hardware block, System Config, bridges the AXI4-Lite interface between the QDMA-IP and other hardware components. In addition to this role, it manages miscellaneous tasks, e.g., card sensor management, hardware soft-reset control, and etc.

The Host hardware incorporates four distinct clock domains. The first is MacTxClk/MacRxClk, derived from AMD PHY, operating at 322.265625 MHz to support 10G Ethernet functionality. The second is axis_clk, generated by the QDMA-IP, running at a fixed frequency of 250 MHz for the PCIe interface. The third is axil_clk, dedicated to the System Config block for managing Alveo card facilities, operating at 125 MHz. Lastly, the ap_clk domain drives the User Module region and can be adjusted based on the trader's requirements to optimize resource utilization and performance. In this reference design, the ap_clk configuration varies depending on the FPGA device capabilities:

- Alveo U250 and X3522 : 312.5 MHz
- Alveo U55C : 300 MHz
- Alveo U50 : 275 MHz

As the flow shown in Figure 4, once Order Book is updated from the processed market data, the Pricing Engine then uses the updated Order Book and the trading strategy to determine the next trading action, passing this information to the Order Entry. If required, Bid-Ask orders are generated and transmitted as TCP packets. In this configuration, the entire trading engine (highlighted in blue) operates directly on the FPGA card, a structure known as "**Pricing Engine on Card**".

In scenarios where trading strategies and algorithms become more complex, FPGA logic along may not handle all computations efficiently. In such cases, the AAT-QDMA offers an alternative configuration called “**Pricing Engine Software**”, where advanced calculations are performed by the Host software. This configuration is illustrated in Figure 5.

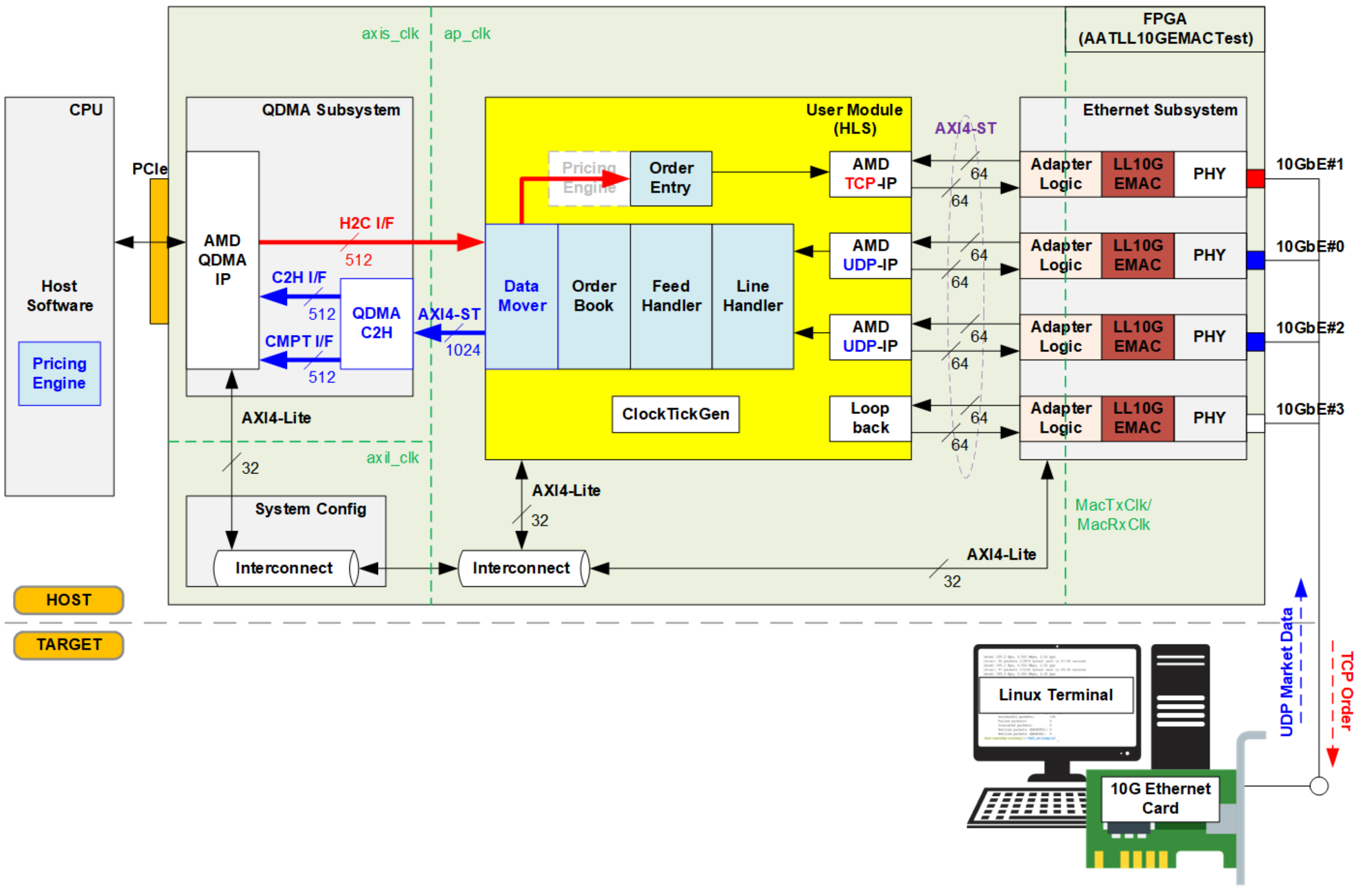


Figure 5 Host Hardware on AAT-QDMA Demo (Pricing Engine Software)

In the "Price Engine Software" structure, the Pricing Engine operates within the Host software under the control of the Host CPU. Market data is first processed on the FPGA, and the updated Order Book is transferred to the Host software via a DMA operation. The Data Mover sub-block within the FPGA facilitates this transfer by forwarding the Order Book over a 1024-bit AXI4 stream interface to the QDMA Subsystem. Unlike register access, this data transfer complies with the AMD QDMA IP Core standard. Within this process, the QDMAC2H module reformats the 1024-bit AXI4 stream into 512-bit Controller-to-Host (C2H) and Completion (CMPT) interfaces of the QDMA-IP.

Once the software-based Pricing Engine completes its computations, the results are transferred back to the FPGA trading engine via the Host-to-Controller (H2C) interface. The Data Mover then forwards these results to the Order Entry module, where Bid-Ask orders are generated.

In the AAT-QDMA reference design provided by Design Gateway, the Host hardware supports both the "Price Engine on Card" and "Price Engine Software" modes. The primary difference between these configurations lies in the data flow through specific hardware modules:

- In the "Price Engine on Card" mode, the Data Mover and QDMAC2H modules exist within the Host hardware but are inactive as no data flows through them.
- In the "Price Engine Software" mode, the Pricing Engine in the FPGA exists and is implemented, but the data flow bypasses it entirely.

Subsequent sections provide detailed descriptions of each hardware module.

2.1 Ethernet Subsystem

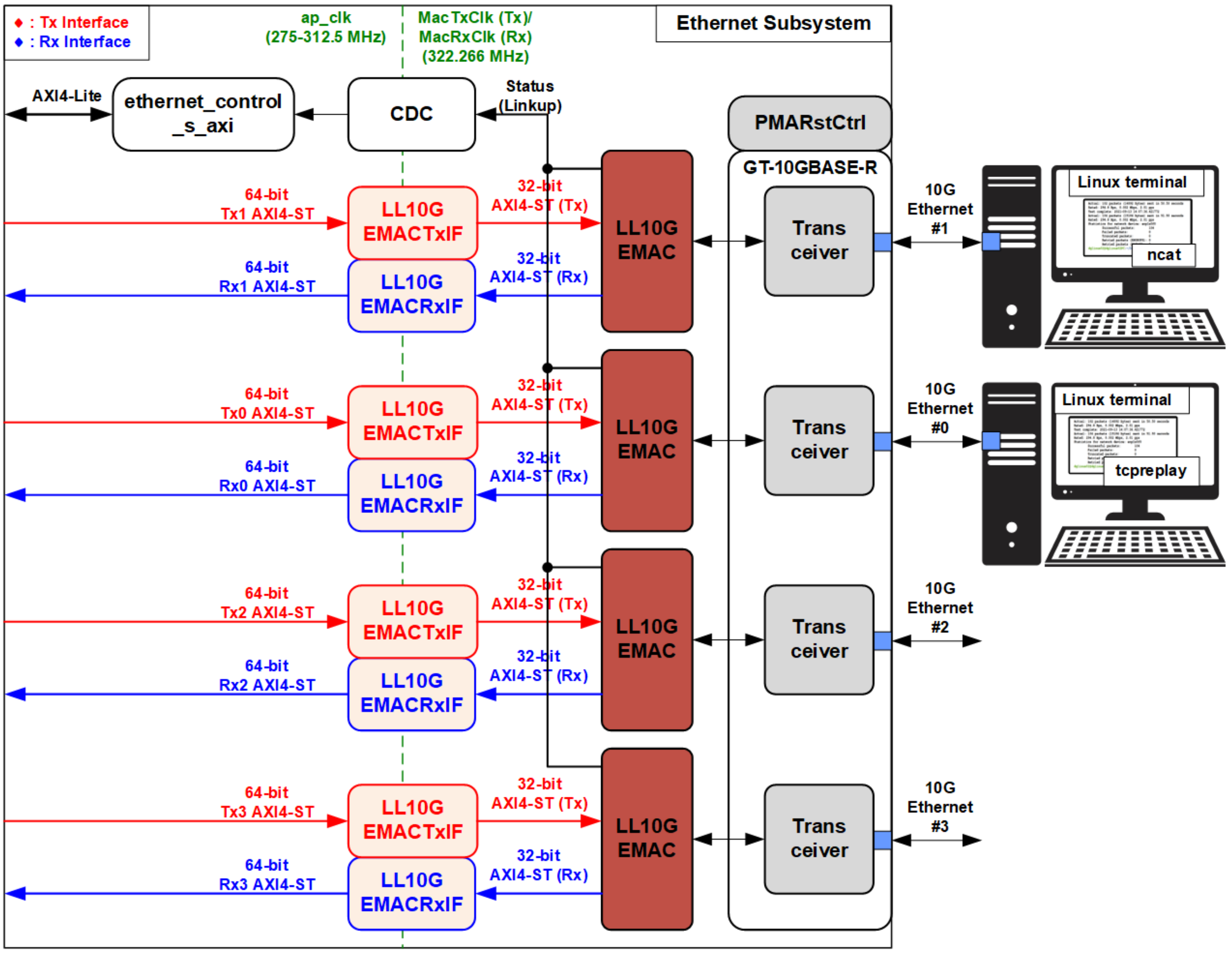


Figure 6 Ethernet Subsystem Block Diagram

To achieve 10G Ethernet connectivity with low latency, Design Gateway integrates the LL10GEMAC-IP with AMD 10G BASE-R PMA solutions. The FPGA cards used in this reference design feature either a QSFP28 connector or two DSFP28 connectors, which can be split into four independent 10G Ethernet channels. As a result, the Ethernet Subsystem incorporates four Ethernet MAC interface sets to manage these connections.

The data interface of Ethernet Subsystem operates in the ap_clk domain, where the frequency can be adjusted to meet specific system requirements. In contrast, the user interface of the transceiver for 10GBASE-R operates in distinct clock domains: MacTxClk for the Tx interface and MacRxClk for the Rx interface. To ensure reliable communication and synchronization across these different clock domains, Clock Domain Crossing (CDC) logic is implemented.

The Ethernet Subsystem provides four interfaces, all using a 64-bit AXI4-Stream interface. The LL10GEMACTxIF module adapts data from the ap_clk domain to MacTxClk, reducing the data width from 64 bits to 32 bits for the transmit path. Conversely, the LL10GEMACRxIF module adapts data from MacRxClk to ap_clk, increasing the data width from 32 bits to 64 bits for the receive path. Both modules interface with the 10G Ethernet MAC controller (LL10GEMAC-IP) using a 32-bit AXI4 stream interface (AXI4-ST) for transmit and receive operations.

The ethernet_control_s_axi module converts the AXI4-Lite interface to internal signals for write/read access and is mapped to the control and status signals of other modules. When signals cross different clock domains, CDC logic is applied to maintain synchronization. The Linkup status of all Ethernet connections is returned as a status signal by the LL10GEMAC-IP.

Further details of each module are described below.

2.1.1 Transceiver (PMA for 10GBASE-R)

The PMA IP core for 10G Ethernet (BASE-R) can be generated using Vivado IP catalog. In the AMD FPGA Transceivers Wizard, the user uses the following settings.

• Transceiver configuration preset	: GT-10GBASE-R
• Encoding/Decoding	: Raw
• Transmitter Buffer	: Bypass
• Receiver Buffer	: Bypass
• User/Internal data width	: 32
• Include transceiver COMMOM in the	: Example Design (not include in Core)

Four channels are enabled in the AAT demo for four Ethernet connections. An example of the Transceiver wizard in Ultrascale model is described in the following link.

https://www.xilinx.com/products/intellectual-property/ultrascale_transceivers_wizard.html.

2.1.2 PMARstCtrl

When bypassing the buffer inside the AMD FPGA transceiver, the user logic must manage the reset signals for both the Tx and Rx buffers. This function is implemented using a state machine that follows the steps outlined below:

1) Assert Tx reset of the transceiver to 1b for one clock cycle.
2) Wait until Tx reset done, output from the transceiver, is asserted to 1b.
3) Finish Tx reset sequence and de-assert Tx reset, user interface output, to allow the user logic beginning Tx operation.
4) Assert Rx reset to the transceiver.
5) Wait until Rx reset done, output from the transceiver, is asserted to 1b.
6) Finish Rx reset sequence and de-assert Rx reset, user interface output, to allow the user logic beginning Rx operation.

2.1.3 DG LL10GEMAC

The IP core by Design Gateway implements low-latency EMAC and PCS logic for 10Gb Ethernet (BASE-R) standard. The user interface is 32-bit AXI4-Stream bus. Please see more details from LL10GEMAC-IP datasheet on our website.

https://dgway.com/products/IP/Lowlatency-IP/dg_ll10gemacip_data_sheet_xilinx_en/

2.1.4 LL10GEMACTxIF

The LL10GEMACTxIF module is an AXI4-Stream data adapter designed to interfaces with transmitting path of LL10EMAC. It performs clock domain crossing and data width conversion from 64 bits to 32 bits. This module is optimized for low-latency performance, but its operation is subject to certain limitations:

1) The ratio of MacTxClk frequency to UserClk frequency must be less than two.
2) If UserTxValid is dropped during transmission, an error is asserted to the Ethernet MAC, prompting it to cancel the current packet transmission. Packet retransmission is then initiated, so the transmission latency increases.

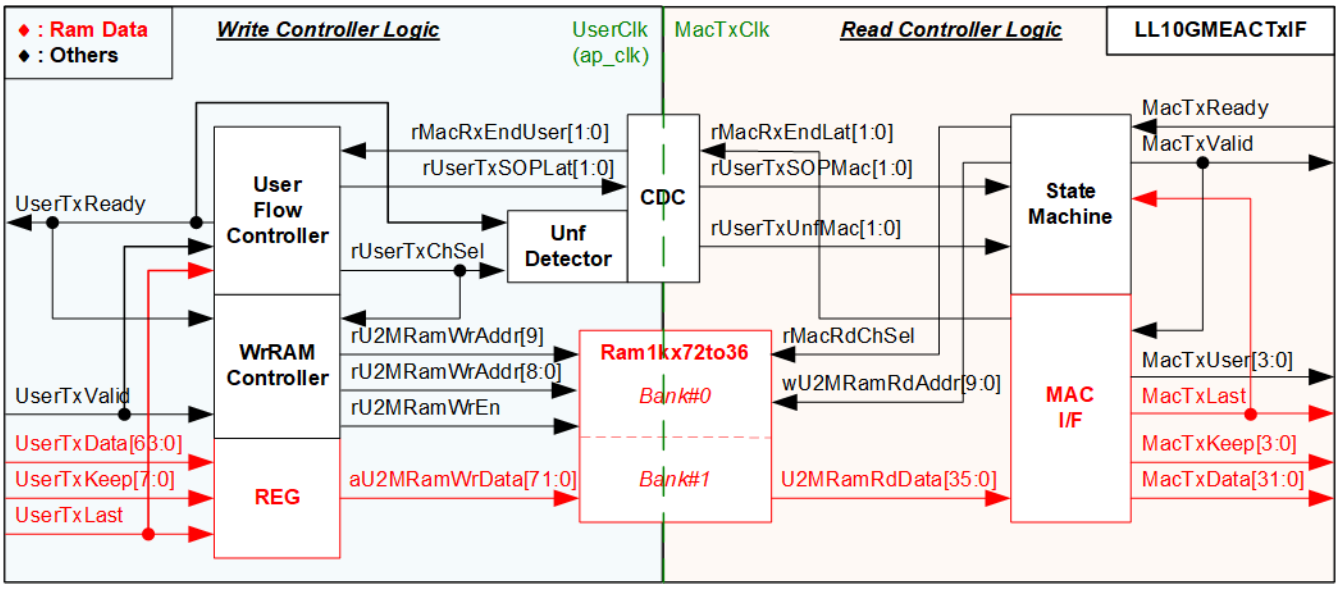


Figure 7 LL10GEMACTxIF Logic Diagram

As illustrated in Figure 7, the Ram1k72to36 buffers the data stream between the User interface to the Ethernet MAC (EMAC). The Write controller logics operates on UserClk, while the Read controller logics runs on MacTxClk. Additionally, clock domain crossing (CDC) logic is implemented to transfer flow control signals across clock domains, i.e., Start-of-packet (UserTxSOP), Underflow flag (UserTxUnf), and End-of-packet (MacRxEnd).

Ram1kx72to36

Ram1kx72to36 is a block memory module configured as a simple dual-port block RAM type with an asymmetric data width feature. It stores 72-bit data in the UserClk domain and converts it to 36-bit data in the MacTxClk domain. The 36-bit data comprises 32-bit data, 2-bit encoded keep signal, 1-bit last flag, and 1-bit reserved signal.

The Ram1kx72to36 module is divided into two banks, enabling it to store two packet data from the user via the AXI4-Stream interface. The maximum packet size is limited to half the size of the RAM, which is 4096 bytes (equivalent to 512 entries of 64-bit data). The packet length can be increased by extending the RAM depth.

Write controller

The User flow controller asserts UserTxReady to accept a new packet transmission when the next RAM bank is available. The WrRAM Controller generates the write address to store data of each packet in the same RAM bank during transmission from user. After the final packet data is transferred, the MSB of the address is inverted to switch the active bank. Additionally, the User flow controller asserts the Start-of-frame flag (rUserTxSOPLat) when the first data of a new packet is received. This flag is forwarded to the Read controller via CDC to initiate packet transmission to the EMAC. Once the Read controller transfers the final packet data from a RAM bank, the End-of-frame flag (rMacRxEndLat) is asserted, signaling the User flow control to free the RAM.

Due to the EMAC limitation, each packet must be transmitted entirely without pauses. To meet this requirement, the data in RAM must always be ready for reading, and the Write controller must write data to RAM continuously until the end of packet. An Unf detector is included to monitor for interruptions during packet transmission. If the user pauses the transmission by de-asserting UserTxValid to 0b before completing the final packet data transfer, the Unf detector asserts the Underflow flag (rUserTxUnfLat). This signal is also sent to the Read controller via CDC to cancel the packet transmission on the EMAC interface.

Read controller

The State machine manages the data flow, forwarding read data from the RAM to the EMAC. MacTxValid is asserted to 1b when the State machine detects the Start-of-frame flag for a new packet. The read address is created by the State machine to read RAM data, which includes 32-bit data, keep data, and the last flag.

The MAC I/F includes a decoder that converts the 2-bit encoded keep signal into a 4-bit signal. If the Underflow flag is asserted before the final data is transmitted, the State machine asserts MacTxLast and MacTxUser to cancel the current packet transmission. Following this, the State machine initiates packet retransmission by reading the first data of the same RAM bank. Once the final packet data is transmitted successfully, the State machine switches to the next RAM bank for the subsequent packet transmission.

Figure 8 and Figure 9 illustrate the timing diagram of the Write Controller, covering both normal and underflow conditions. The following information further details on the timing diagram.

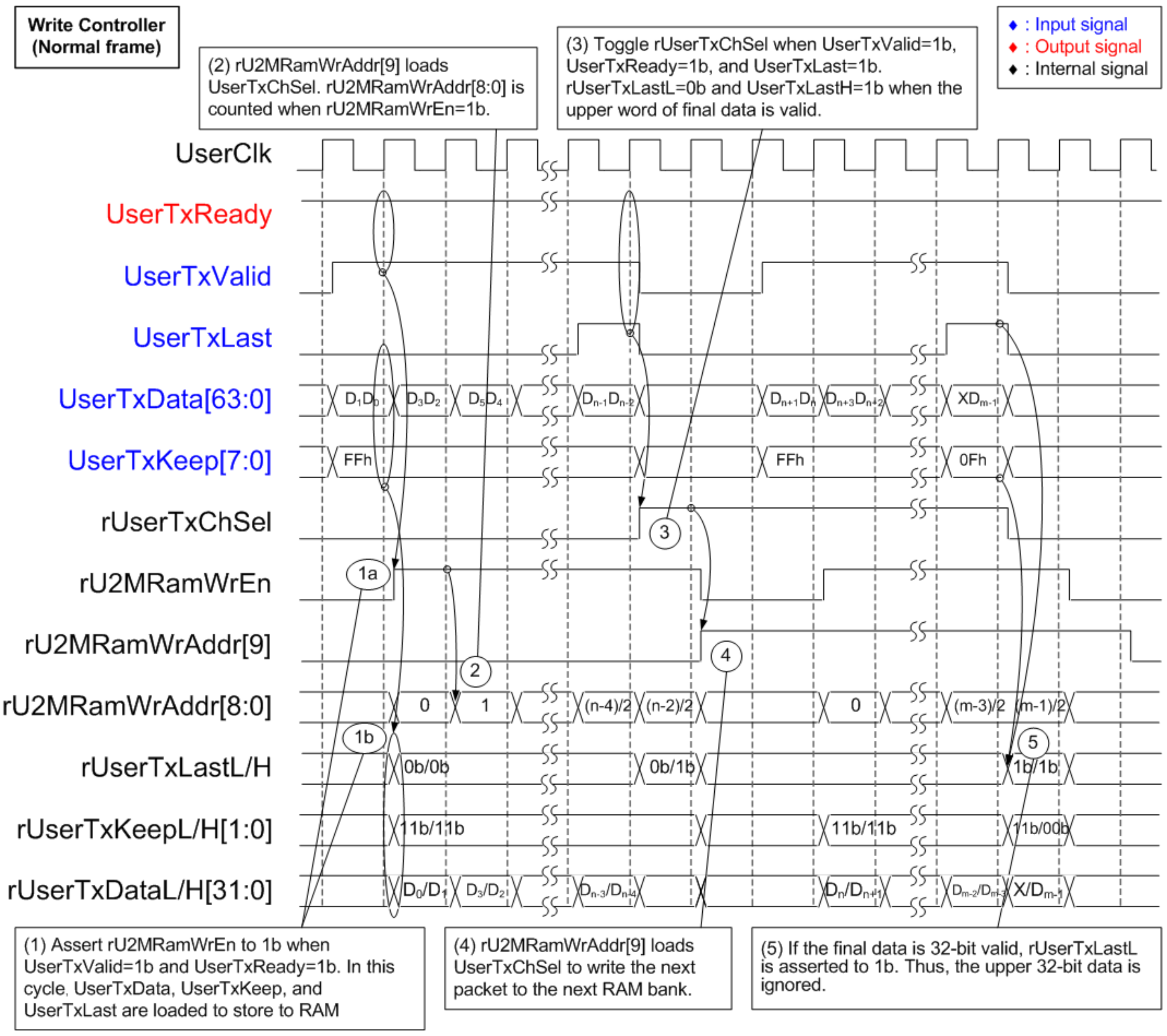

Write Controller (Normal frame)
(2) rU2MRamWrAddr[9] loads UserTxChSel. rU2MRamWrAddr[8:0] is counted when rU2MRamWrEn=1b.
(3) Toggle rUserTxChSel when UserTxValid=1b, UserTxReady=1b, and UserTxLast=1b. rUserTxLastL=0b and UserTxLastH=1b when the upper word of final data is valid.
♦ : Input signal
♦ : Output signal
♦ : Internal signal
UserClk
UserTxReady
UserTxValid
UserTxLast
UserTxData[63:0]
UserTxKeep[7:0]
rUserTxChSel
rU2MRamWrEn
rU2MRamWrAddr[9]
rU2MRamWrAddr[8:0]
rUserTxLastL/H
rUserTxKeepL/H[1:0]
rUserTxDataL/H[31:0]
D1D0
D3D2
D5D4
Dn-1Dn-2
Dn+1Dn
Dn+3Dn+2
XDm-1
FFh
FFh
0Fh
1a
1b
2
3
4
5
0
1
(n-4)/2
(n-2)/2
0
(m-3)/2
(m-1)/2
0b/0b
0b/1b
1b/1b
11b/11b
11b/11b
11b/00b
D0/D1
D3/D2
Dn-3/Dn-4
Dn/Dn+1
Dm-2/Dm-3
X/Dm-1
(1) Assert rU2MRamWrEn to 1b when UserTxValid=1b and UserTxReady=1b. In this cycle, UserTxData, UserTxKeep, and UserTxLast are loaded to store to RAM
(4) rU2MRamWrAddr[9] loads UserTxChSel to write the next packet to the next RAM bank.
(5) If the final data is 32-bit valid, rUserTxLastL is asserted to 1b. Thus, the upper 32-bit data is ignored.

Figure 8 Write Controller of LL10GEMACTxIF Timing Diagram (Normal Case)

1) When UserTxValid is set to 1b to transmit 64-bit data stream via the AXI4-Stream interface, the Write Controller asserts UserTxReady to 1b to accept data. The User data (UserTxData), User byte enable (UserTxKeep), and User last flag (UserTxLast) are loaded into rUserTxDataL/H, rUserTxKeepL/H, and rUserTxLastL/H, respectively, while the write enable signal (rU2MRamWrEn) is set to 1b. To reduce the RAM data width, the 8-bit UserTxKeep is encoded into two 2-bit value (UserTxKeepL/H) based on the following mapping.

Table 1 UserTxKeepL/H Encoding Table

UserTxKeep[7:0]	UserTxKeepL[1:0]	UserTxKeepH[1:0]	Description
01h	00b	Not use	UserTxData[7:0] is valid.
03h	01b		UserTxData[15:0] is valid.
07h	10b		UserTxData[23:0] is valid.
0Fh	11b		UserTxData[31:0] is valid.
1Fh	11b	00b	UserTxData[39:0] is valid.
3Fh	11b	01b	UserTxData[47:0] is valid.
7Fh	11b	10b	UserTxData[55:0] is valid.
FFh	11b	11b	UserTxData[63:0] is valid.

The 72-bit write data to RAM is assigned as follows.

- Bit[31:0] : rUserTxDataL
- Bit[33:32] : rUserTxKeepL
- Bit[34] : rUserTxLastL
- Bit[35] : Reserved
- Bit[67:36] : rUserTxDataH
- Bit[69:68] : rUserTxKeepH
- Bit[70] : rUserTxLastH
- Bit[71] : Reserved

The MSB of the write RAM address (rU2MRamWrAddr[9]) indicates the active RAM bank and is loaded from rUserTxChSel. rU2MRamWrAddr[8:0] is reset to 0 when storing the first packet data.

2) Once each data word is written to RAM with rU2MRamWrEn set to 1b, rU2MRamWrAddr[8:0] increments to store the incoming data at the next address in the RAM.

3) When the final packet data is received (UserTxValid=1b, UserTxLast=1b, and UserTxReady=1b), the bank indicator (rUserTxChSel) toggles to change the active bank. When UserTxLast is set to 1b to store the final packet data, rUserTxLastH is set to 1b. However, the value of rUserTxLastL depends on UserTxKeep. If UserTxKeep[7:4] ≠ 0, indicating that the final 32-bit data is stored in the upper word, rUserTxLastL is set to 0b.

4) rU2MRamWrAddr[9] toggles its value to select a different RAM bank for the next packet.

5) If the upper 32-bit data of the final packet is not valid (UserTxKeep[7:4]=0b), rUserTxLastL is asserted to 1b, indicating that the final data is positioned on the lower 32-bit data.

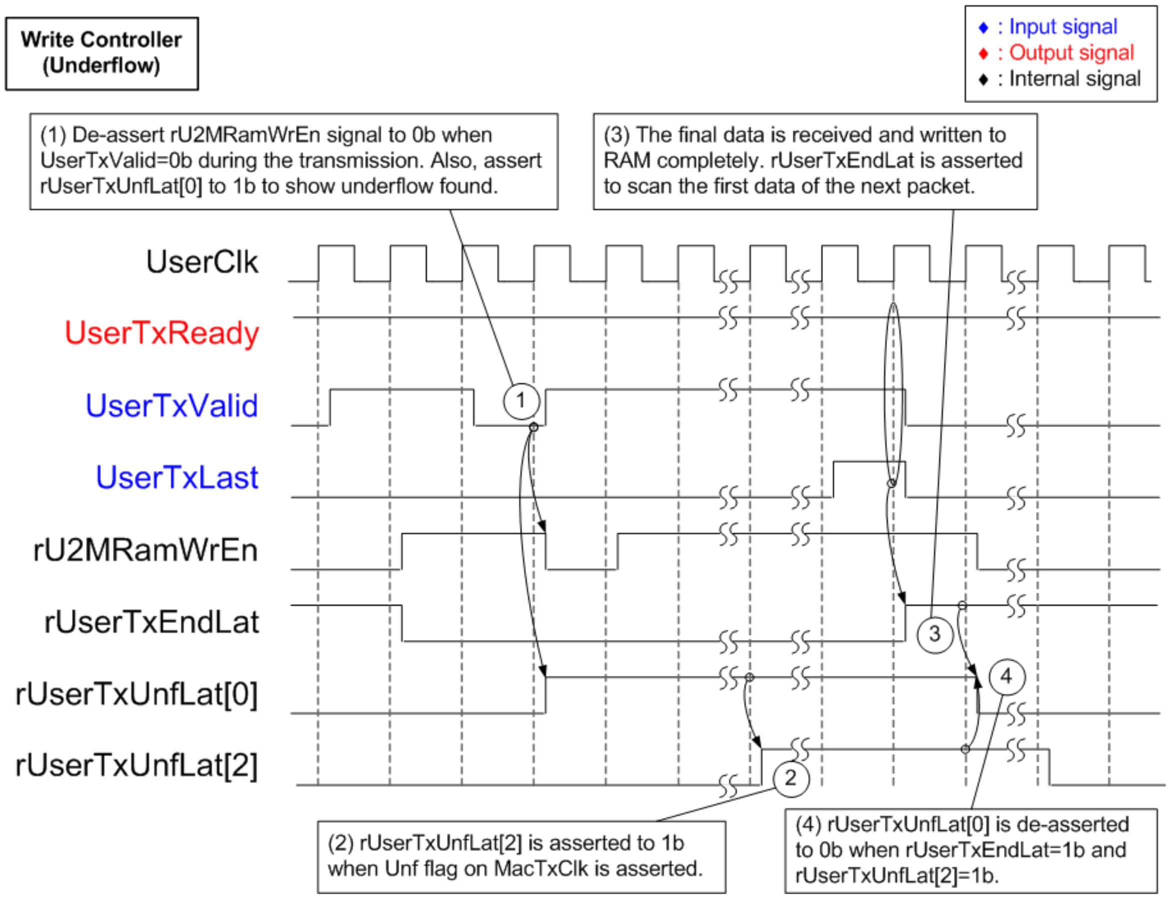

Write Controller (Underflow)
♦ : Input signal
♦ : Output signal
♦ : Internal signal
(1) De-assert rU2MRamWrEn signal to 0b when UserTxValid=0b during the transmission. Also, assert rUserTxUnfLat[0] to 1b to show underflow found.
(3) The final data is received and written to RAM completely. rUserTxEndLat is asserted to scan the first data of the next packet.
UserClk
UserTxReady
UserTxValid
UserTxLast
rU2MRamWrEn
rUserTxEndLat
rUserTxUnfLat[0]
rUserTxUnfLat[2]
(2) rUserTxUnfLat[2] is asserted to 1b when Unf flag on MacTxClk is asserted.
(4) rUserTxUnfLat[0] is de-asserted to 0b when rUserTxEndLat=1b and rUserTxUnfLat[2]=1b.

Figure 9 Write Controller of LL10GEMACTxIF Timing Diagram (Underflow Case)

Figure 9 illustrates the assertion of the Underflow flag to cancel an ongoing packet transmission when UserTxValid is de-asserted to 0b before the completion of the packet. In such cases, the packet must be retransmitted. For retransmision, the Underflow flag asserted by the Write controller is forwarded to the Read controller to halt the current packet transmission. Further details about this retransmission process are provided in Figure 11.

1) Data from the user is always stored in RAM by asserting rU2MRamWrEn to 1b when both UserTxValid and UserTxReady are set to 1b. However, if UserTxValid is set to 0b before the reception of the final packet data (UserTxEndLat=0b), the Underflow Flag (rUserTxUnfLat[0]) is asserted to 1b.

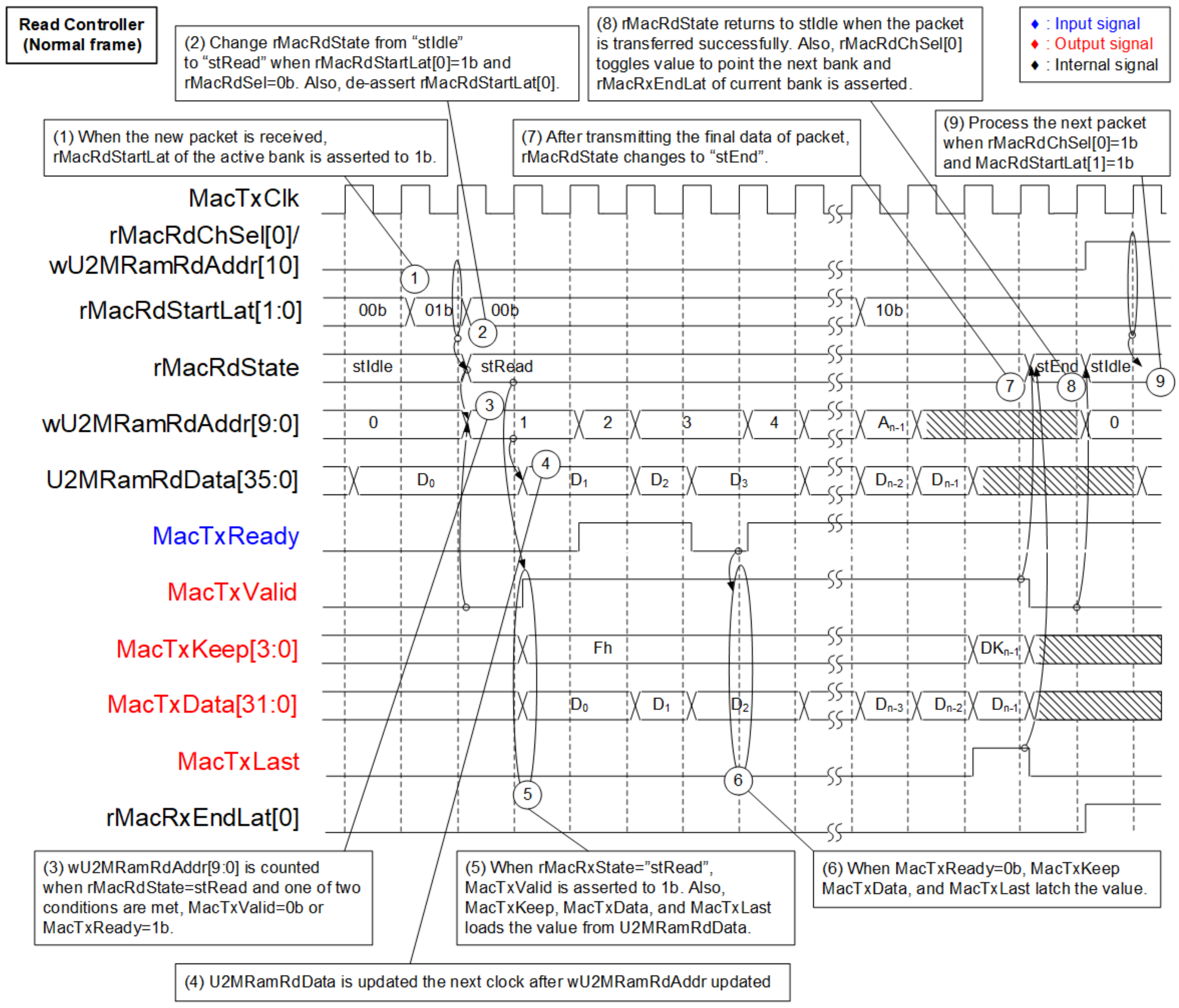
   Note: Although rUserTxUnfLat is asserted, all incoming packet data from the user continues to be stored in RAM until the operation is complete.

2) The Underflow flag (rUserTxUnfLat[0]) is forwarded via a Clock Domain Crossing (CDC) to assert the Underflow flag in the MacTxClk domain. Upon receiving this signal, the Read controller cancels the ongoing packet transmission and initiates packet retransmission. At this point, rUserTxUnfLat[2] is also asserted to 1b, indicating that the Read controller has acknowledged the cancellation request and initiated retransmission.

3) Once the final packet data is received (UserTxLast=1b, UserTxValid=1b, and UserTxReady=1b), rUserTxEndLat is asserted to 1b.

4) The Underflow flag (rUserTxUnfLat[0]) is de-asserted to 0b after the final data is received (rUserTxEndLat=1b) and the packet cancellation request has been completely forwarded to the Read controller (rUserTxUnfLat[2]=1b).

Figure 10 presents the timing diagram of the Read Controller under normal operating conditions. The following steps explain the diagram.



Read Controller (Normal frame)
(2) Change rMacRdState from "stIdle" to "stRead" when rMacRdStartLat[0]=1b and rMacRdSel=0b. Also, de-assert rMacRdStartLat[0].
(8) rMacRdState returns to stIdle when the packet is transferred successfully. Also, rMacRdChSel[0] toggles value to point the next bank and rMacRxEndLat of current bank is asserted.
♦ : Input signal
♦ : Output signal
♦ : Internal signal
(1) When the new packet is received, rMacRdStartLat of the active bank is asserted to 1b.
(7) After transmitting the final data of packet, rMacRdState changes to "stEnd".
(9) Process the next packet when rMacRdChSel[0]=1b and MacRdStartLat[1]=1b
MacTxClk
rMacRdChSel[0]/ wU2MRamRdAddr[10]
rMacRdStartLat[1:0]
00b
01b
00b
10b
rMacRdState
stIdle
stRead
stEnd
stIdle
wU2MRamRdAddr[9:0]
0
1
2
3
4
An-1
0
U2MRamRdData[35:0]
D0
D1
D2
D3
Dn-2
Dn-1
MacTxReady
MacTxValid
MacTxKeep[3:0]
Fh
DKn-1
MacTxData[31:0]
D0
D1
D2
Dn-3
Dn-2
Dn-1
MacTxLast
rMacRxEndLat[0]
(3) wU2MRamRdAddr[9:0] is counted when rMacRdState=stRead and one of two conditions are met, MacTxValid=0b or MacTxReady=1b.
(5) When rMacRxState="stRead", MacTxValid is asserted to 1b. Also, MacTxKeep, MacTxData, and MacTxLast loads the value from U2MRamRdData.
(6) When MacTxReady=0b, MacTxKeep MacTxData, and MacTxLast latch the value.
(4) U2MRamRdData is updated the next clock after wU2MRamRdAddr updated

Figure 10 Read Controller of LL10GEMACTxIF Timing Diagram (Normal Case)

1) The Start flag (rMacRdStartLat) is asserted to 1b by the Write controller and is fed to the Read controller via CDC. This flag indicates the arrival of a new packet and prompts the Read controller to initiate processing.
2) If the start flag of the currently active bank is asserted (rMacRdStartLat[0]=1b when rMacRdChSel[0]=0b), the state transitions from Idle state ("stIdle") to Read state ("stRead").
<u>Note</u>: rMacRdChSel[0] indicates the active bank (e.g., 0b for Bank#0 and 1b for Bank#1).
3) The read address, wU2MRamRdAddr[9:0], is initialized to 0 and increments only when the main state is in Read state (rMacRdState=stRead). This occurs when either the first time is being loaded (MacTxValid=0b) or the EMAC is receiving AXI4-Stream output (MacTxReady= 1b). The address is updated using combinational logic, ensuring it updates within the same cycle when MacTxValid=0b or MacRxReady=1b. The MSB of the read address (wUMRamRdAddr[10]) corresponds to the channel indicator (rMacRdChSel), selecting RAM bank.
4) The output of the read data from RAM (U2MRamRdData) is updated after the read address (wUMRamRdAddr[10:0]) is modified.
5) When the main state (rMacRdState) is in Read state ("stRead"), the MacTxValid signal is asserted to 1b. Simultaneously, the 36-bit read data from RAM (U2MRamRdData) is loaded as the output to the MAC interface (Bit[31:0]: MacTxData, Bit[33:32]: Decoded as MacTxKeep, Bit[34]: MacTxLast). The decoding of MacTxKeep is as follows: 00b -> 0001b, 01b -> 0011b, 10b -> 0111b, and 11b -> 1111b.
6) If the current data is not accepted by the EMAC (MacTxReady=0b and MacTxValid=1b), the values of MacTxKeep, MacTxData, and MacTxLast are latched.
7) After the final data is transmitted to the EMAC (MacTxValid=1b and MacTxLast=1b), the state transitions to "stEnd".

8) In "stEnd", the state waits until MacTxValid is de-asserted to 0b, confirming that the final data has been fully accepted. The state then returns to "stIdle", awaiting the next transfer to the next RAM bank. During this cycle, the active bank (rMacRdChSel[0]) toggles, and the End transfer flag (rMacRxEndLat) is asserted. The End flag is sent back to the Write controller to release the RAM for storing newly received packets from the user.
9) The process returns to step (1), waiting for rMacRdStartLat of the active bank to be asserted. The active bank is switched to the new bank (rMacRdChSel[0]=1b), ensuring that the Read controller is prepared for the next packet transmission in the updated bank.

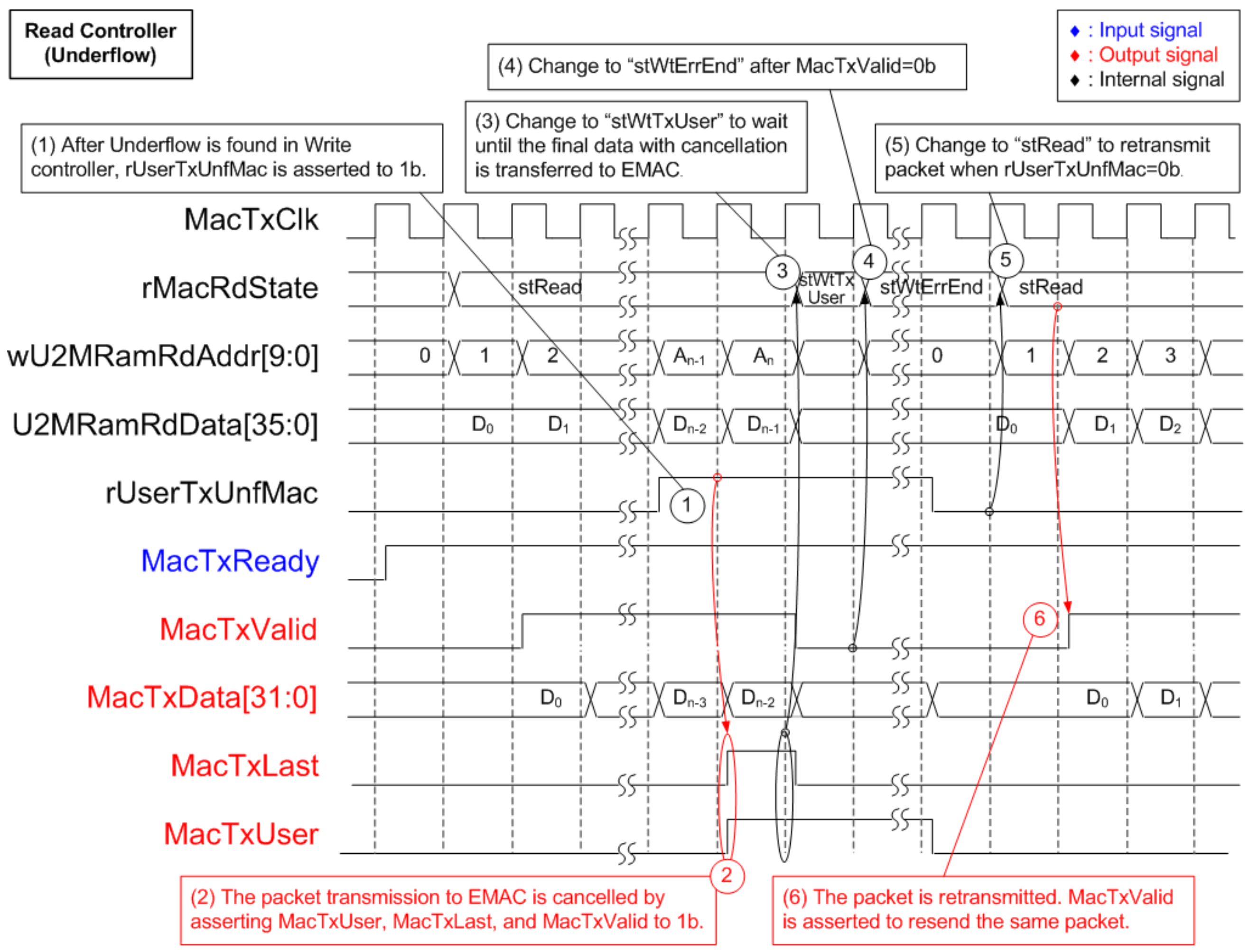

Read Controller (Underflow)
(4) Change to "stWtErrEnd" after MacTxValid=0b
♦ : Input signal
♦ : Output signal
♦ : Internal signal
(1) After Underflow is found in Write controller, rUserTxUnfMac is asserted to 1b.
(3) Change to "stWtTxUser" to wait until the final data with cancellation is transferred to EMAC.
(5) Change to "stRead" to retransmit packet when rUserTxUnfMac=0b.
MacTxClk
rMacRdState
wU2MRamRdAddr[9:0]
U2MRamRdData[35:0]
rUserTxUnfMac
MacTxReady
MacTxValid
MacTxData[31:0]
MacTxLast
MacTxUser
stRead
stWtTx User
stWtErrEnd
stRead
0
1
2
A_{n-1}
A_n
0
1
2
3
D0
D1
D_{n-2}
D_{n-1}
D0
D1
D2
D0
D_{n-3}
D_{n-2}
D0
D1
1
2
3
4
5
6
(2) The packet transmission to EMAC is cancelled by asserting MacTxUser, MacTxLast, and MacTxValid to 1b.
(6) The packet is retransmitted. MacTxValid is asserted to resend the same packet.

Figure 11 Read Controller of LL10GEMACTxIF Timing Diagram (Underflow Case)

1) When the Write controller detects that the user de-asserts UserTxValid to 0b before transmitting the final data of the packet, the Underflow flag in the MacTxClk domain (rUserTxUnfMac) is asserted to 1b.
2) If rUserTxUnfMac is asserted to 1b while the main state is "stRead", the ongoing packet transmission to the EMAC is halted. This is achieved by asserting MacTxValid, MacTxLast, and MacTxUser to 1b. When MacTxUser is asserted at the end of the packet, the EMAC concludes the packet transmission with an error condition directed at the target, causing the target to discard this packet.
3) The state machine transitions to "stWtTxUser", where it waits for the final data to be successfully transmitted to the EMAC.
4) The state machine verifies the successful transmission by detecting the de-assertion of MacTxValid to 0b. Once this is confirmed, the state changes to "stWtErrEnd".
5) In "stWtErrEnd", the state machine waits until the Underflow flag (rUserTxUnfMac) is de-asserted to 0b. Following this, the state transitions back to "stRead" to initiate packet retransmission.
6) Once retransmission begin, MacTxValid is re-asserted.

2.1.5 LL10GEMACRxIF

The LL10GEMACRxIF module serves as an AXI4-Stream data adapter interfacing with the receiving path of LL10GEMAC. It performs clock domain crossing for AXI4-stream and converts the data width from 32 bits to 64 bits. The module comprises several components: AXI4-ST 32bto64b Converter, Error Detection, AsyncFWFTFifo32x69, and AXI4-ST Read Controller.

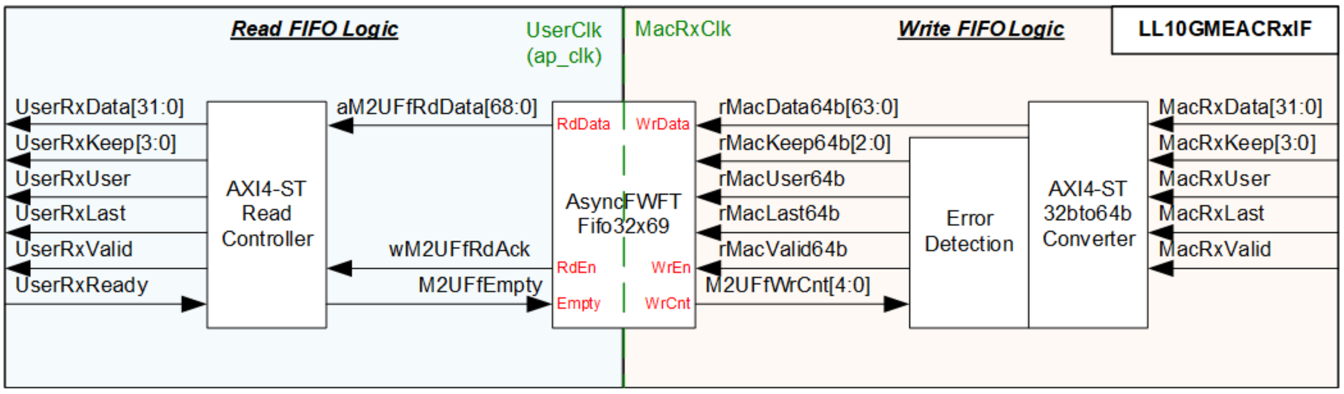


Figure 12 LL10GEMACRxIF Logic Diagram

The AXI4-ST 32bto64b Converter handles data width conversion from 32 bits to 64 bits and writes the data into the AsyncFWFTFifo32x69 module. Error Detection cancels the write operation in two cases: if the received packet size exceeds 9014 bytes, or if the available FIFO space drops below the threshold value of 8 data. When an error is detected, rMacUser64b, rMacLast64b, and rMacValid64b are asserted to 1b, marking the completion of the current packet transmission with an error status.

The AsyncFWFTFifo32x69 module functions as an asynchronous FWFT FIFO, storing data in the MacRxClk domain and transferring it to the UserClk domain. It handles 69-bit data, which includes 64 bits for data, 3 bits for encoded keep signal indicating 1-8 valid bytes, 1 bit for an error flag, and 1 bit for a last flag.

The AXI4-Stream Read Controller reads data from the AsyncFWFTFifo32x69 and forwards it to the user as AXI4-Stream data. Data is read only when the FIFO is not empty, and either the user is ready to receive data or the current data is the first one in a packet for transmission.

Figure 13 illustrates the timing diagram for writing to the FIFO inside LL10GEMACRxIF. The following details provide a description of the timing diagram.

1) The rWordHEn signal is initially set to 0b. It is asserted to 1b upon receiving the 32-bit lower data (MacRxValid=1b and rWordHEn=0b). Conversely, it is de-asserted to 0b to receive the 32-bit upper data (MacRxValid=1b and rWordHEn=1b).

   The 8-bit keep signal (MacRxKeep[3:0]), indicating byte enable, is encoded into a 3-bit register (rMacKeep64b[2:0]) to represent the validity of the 32-bit data. The mapping of MacRxKeep[3:0] to rMacKeep64b[1:0] is as follows: 0001b->00b, 0011b->01b, 0111b->10b, 1111b->11b.

   The value of rMacKeep64b[2] is determined by rWordHEn. If rWordHEn=1b, it indicates the validity of the 32-bit upper data. The 32-bit upper data (rMacData64b[64:32]) and 32-bit lower data (rMacData64b[31:0]) are loaded from MacRxData[31:0] based on the state of rWordHEn. The lower word is loaded when rWordHEn=0b, while the upper word is loaded when rWordHEn=1b.

2) Simultaneously with the loading of the 32-bit upper data (rMacData64b[64:32]), the 64-bit data (rMacData64b), 3-bit keep (rMacKeep64b), 1-bit error flag (rMacUser64b), and 1-bit last flag (rMacLast64b) are written to the FIFO by asserting rMacValid64b to 1b.

3) If the EMAC pauses data transmission by de-asserting MacRxValid to 0b, the values of rWordHEn, rMacData64b, and rMacKeep64b are latched.

4) When the end of the packet is transferred (MacRxValid=1b and MacRxLast=1b), MacRxKeep[3:0] may not be all ones. For example, if MacRxKeep[3:0]=0111b, rMacKeep64b[1:0] is encoded as 10b, and rMacLast64b is asserted to 1b. Two conditions determine how the final data is stored:

- Condition 1: MacRxLast=1b, MacRxValid=1b, and rWordHEn=1b. In this case, the upper data (rMacData64b[63:32]) is valid, and rMacKeep64b[2] is asserted to 1b.
- Condition 2: MacRxLast=1b, MacRxValid=1b, and rWordHEn=0b. In this case, only the lower data (rMacData64b[31:0]) is valid, rMacKeep64b[2] is de-asserted to 0b, and rWordHEn remains 0b after receiving the final data.

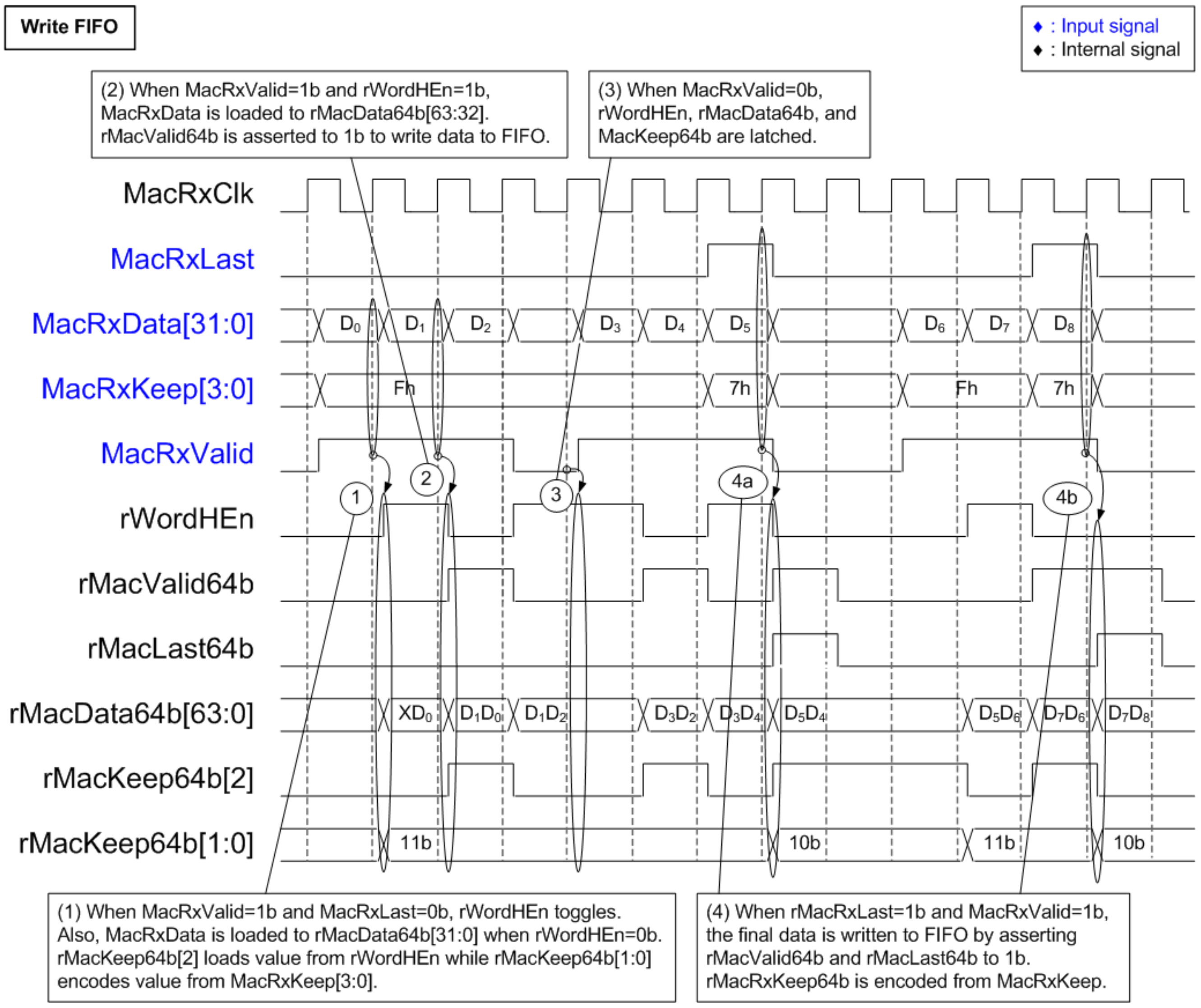

Write FIFO
♦ : Input signal
♦ : Internal signal
(2) When MacRxValid=1b and rWordHEn=1b, MacRxData is loaded to rMacData64b[63:32]. rMacValid64b is asserted to 1b to write data to FIFO.
(3) When MacRxValid=0b, rWordHEn, rMacData64b, and MacKeep64b are latched.
MacRxClk
MacRxLast
MacRxData[31:0]
MacRxKeep[3:0]
MacRxValid
rWordHEn
rMacValid64b
rMacLast64b
rMacData64b[63:0]
rMacKeep64b[2]
rMacKeep64b[1:0]
D0 D1 D2 D3 D4 D5 D6 D7 D8
Fh 7h Fh 7h
1 2 3 4a 4b
XD0 D1D0 D1D2 D3D2 D3D4 D5D4 D5D6 D7D6 D7D8
11b 10b 11b 10b
(1) When MacRxValid=1b and MacRxLast=0b, rWordHEn toggles. Also, MacRxData is loaded to rMacData64b[31:0] when rWordHEn=0b. rMacKeep64b[2] loads value from rWordHEn while rMacKeep64b[1:0] encodes value from MacRxKeep[3:0].
(4) When rMacRxLast=1b and MacRxValid=1b, the final data is written to FIFO by asserting rMacValid64b and rMacLast64b to 1b. rMacRxKeep64b is encoded from MacRxKeep.

Figure 13 Write FIFO Logic of LL10GEMACRxIF Timing Diagram

Figure 14 presents the timing diagram of the AXI4-Stream Read Controller. The following details provide a description of the timing diagram.

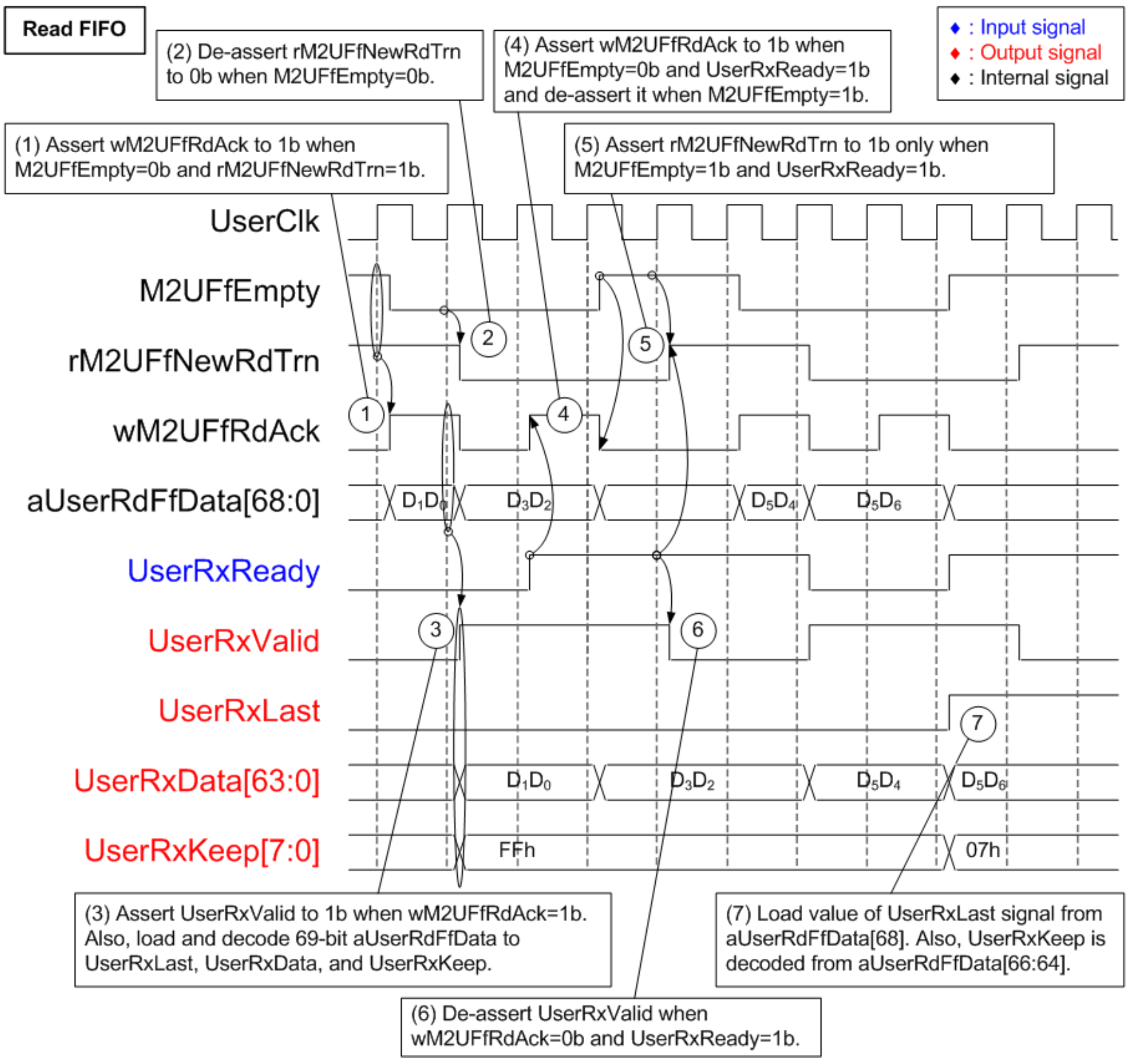

Read FIFO
(2) De-assert rM2UFfNewRdTrn to 0b when M2UFfEmpty=0b.
(4) Assert wM2UFfRdAck to 1b when M2UFfEmpty=0b and UserRxReady=1b and de-assert it when M2UFfEmpty=1b.
♦ : Input signal
♦ : Output signal
♦ : Internal signal
(1) Assert wM2UFfRdAck to 1b when M2UFfEmpty=0b and rM2UFfNewRdTrn=1b.
(5) Assert rM2UFfNewRdTrn to 1b only when M2UFfEmpty=1b and UserRxReady=1b.
UserClk
M2UFfEmpty
rM2UFfNewRdTrn
wM2UFfRdAck
aUserRdFfData[68:0]
UserRxReady
UserRxValid
UserRxLast
UserRxData[63:0]
UserRxKeep[7:0]
D1D0
D3D2
D5D4
D5D6
FFh
07h
(3) Assert UserRxValid to 1b when wM2UFfRdAck=1b. Also, load and decode 69-bit aUserRdFfData to UserRxLast, UserRxData, and UserRxKeep.
(7) Load value of UserRxLast signal from aUserRdFfData[68]. Also, UserRxKeep is decoded from aUserRdFfData[66:64].
(6) De-assert UserRxValid when wM2UFfRdAck=0b and UserRxReady=1b.

Figure 14 Read FIFO Logic of LL10GEMACRxIF Timing Diagram

1) Once the data is written to the FIFO, M2UFfEmpty is de-asserted to 0b. Since the FIFO is the FWFT type, the read FIFO data (aUserRdFfData) becomes valid when the FIFO read enable (wM2UFfRdAck) is asserted. wM2UFfRdAck is immediately set to 1b when the FIFO is not empty (M2UFfEmpty=0b) and there is no remaining latched data in the Read Controller (rM2UFfNewRdTrn=1b). Subsequently, the data is transferred to the user.
2) rM2UFfNewRdTrn is de-asserted to pause reading data from the FIFO when latched data remains, as indicated by M2UFfEmpty=0b. After that, wM2UFfRdAck is controlled by UserRxReady.
3) When the read FIFO enable signal (wM2UFfRdAck) is asserted to 1b, the AXI4-Stream valid output (UserRxValid) is also set to 1b. Furthermore, the AXI4-Stream data (UserRxData[63:0]) and the last flag (UserRxLast) are loaded from aUserRdFfData[63:0] and aUserRdFfData[68], respectively. While the keep signal (UserRxKeep[7:0]) is decoded from aUserRdFfData[66:64] using the following mapping: 000b -> 01h, 001b -> 03h, 010b -> 07h, 011b -> 0Fh, 100b -> 1Fh, 101b -> 3Fh, 110b -> 7Fh, and 111b -> FFh.
4) When rM2UFfNewRdTrn=0b and M2UFfEmpty=0b, wM2UFfRdAck is controlled by UserRxReady. It is asserted to 1b to read the next data when UserRxReady is also asserted to 1b to accept the current data.
5) Once the latched data is completely transferred to the user (UserRxReady=1b) and there is no remaining data in the FIFO (M2UFfEmpty=1b), rM2UFfNewRdTrn is re-asserted to 1b. The system now returns to the state described in step (1), waiting for new data in the FIFO.
6) At this point, UserRxValid is de-asserted to 0b, pausing the data transmission.
7) The last flag (UserRxLast) is asserted to 1b when aUserRdFfData[68]=1b. When transmitting the last data, UserRxKeep may be equal to 01h, 03h, 07h, …, FFh, indicating the valid bytes of the final data.

2.2 User Module

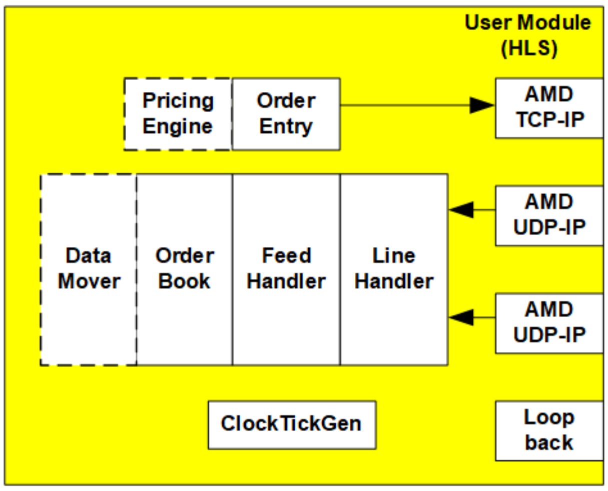


Figure 15 User Module Component

The User Module consists of various submodules developed using High-Level Synthesis (HLS) to minimize development time. These submodules are derived from the AMD Accelerated Algorithmic Trading Demo. However, the DataMover submodule has been specifically modified to interface with the AMD QDMA-IP instead of XDMA-IP, ensuring compatibility with the system's PCIe data transfer infrastructure.

Market data is generally transmitted over two Ethernet channels using UDP/IP protocols. Duplicated market data on both channels is processed by Line Handler, which merged the data into a single stream. This merged stream is then decoded by the Feed Handler and subsequent submodules, which follow the CME MDP 3.0 Market Data Protocol. This protocol uses the Financial Information Exchange (FIX) format with Simple Binary Encoding (SBE).

By leveraging HLS, users can easily develop, replace, and modify these submodules to suit specific trading strategies and market requirements. Detailed descriptions of each submodule in the User Module are provided in the following sections.

2.2.1 UDP

This UDP submodule implements a UDP stack to enable data transmission over a UDP/IP connection, providing hardware-based networking capabilities tailored for trading systems. During initialization, the Host Software configures key network parameters, including the IP address, subnet mask, and UDP listening ports, using the AXI4-Lite interface.

Packets with a matching target UDP port are processed, and the payload data is forwarded to the Line Handler submodule for further processing. The UDP submodule supports features such as multicast transmission, ARP and ICMP packet transmission for efficient data handling in real-time trading environments.

2.2.2 Line Handler

The Line Hadler submodule is designed to arbitrate between two UDP streams, referred to as the A Line and B line which are linked to UDP#0 and UDP#1, to ensure reliable data delivery. Many stock exchanges deliver data feeds via UDP multicast, a high-speed but unreliable method where packets can be lost or delivered out of order. By subscribing to two identical multicast streams, the Line Handler ensures data integrity through line arbitration.

The Line Handler uses sequence number embedded in the packet headers to track and process incoming packets. This module outputs a single, reliable stream of market data for subsequent processing by the Feed Handler submodule.

2.2.3 Feed Handler

The Feed Handler submodule processed arbitrated UDP market data received from the Line Handler and decodes the messages according to the CME Market Data Protocol (MDP). This module specifically handles the feed for reporting the highest bid and lowest ask prices for up to five levels in the order book.

Messages received by the Feed Handler are formatted using the FIX protocol and encoded in Simple Binary Encoding (SBE). The submodule decodes these messages, and extracts key trading information such as price levels and bid/ask quantities. After updating securities information, the results are then passed to the Order Book submodule to maintain an up-to-date market view.

2.2.4 OrderBook

The OrderBook submodule maintains a five-level order book for both bid and ask sides. Each level contains aggregated quantities, prices, and the number of orders at specific price points. This submodule processes updates from the exchange feed, recalculates the order book, and ensures accurate market representation.

The submodule receives update instructions such as add, modify, or delete operations. It retrieves the current state of the order book from memory, applies the specified update, and reorders entries if necessary. Updated order book is then forwarded to either the Pricing Engine Submodule for on-card bid-ask order generation or the DataMover Submodule for transfer to the host memory. This dual-output capability supports both On-Card mode and Software mode configurations for the Pricing Engine.

2.2.5 DataMover

The DataMover submodule handles data transfer in both uplink and downlink of QDMA Subsystem. The uplink transfers the updated OrderBook data while the downlink transfers the trade information to the OrderEntry submodule. The DataMover is enabled when the Pricing Engine is implemented by Host software.

Two components are implemented in DataMover for handling data transfer in each direction: responseMove for uplink transfer and operationMove for downlink transfer, as shown in Figure 16.

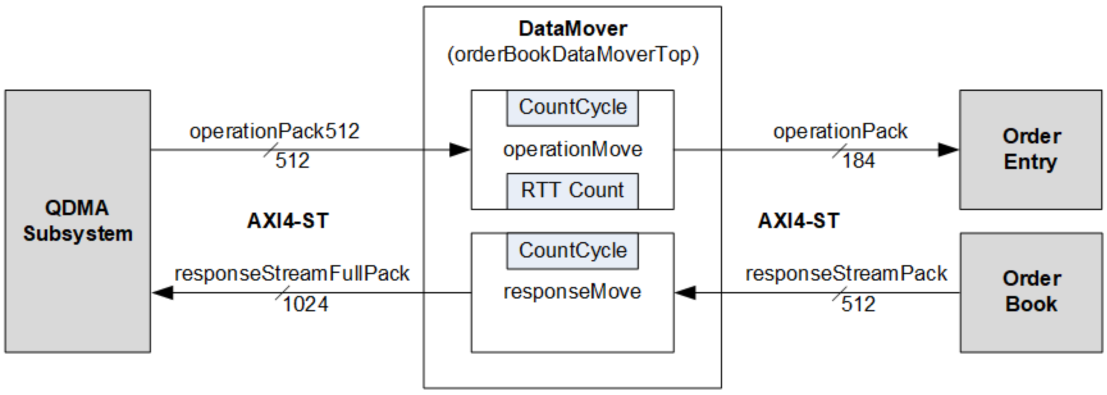


Figure 16 DataMover Submodule

responseMove

The responseMove manages data flow between two AXI4-Stream interfaces:

- **responseStreamPack**: A 256-bit interface that receives metadata and level data from the OrderBook.
- **responseStreamFullPack**: A 1024-bit interface that transmits packed data to the QDMA subsystem.

The structure of the packets involves in this process is illustrated in Figure 17.

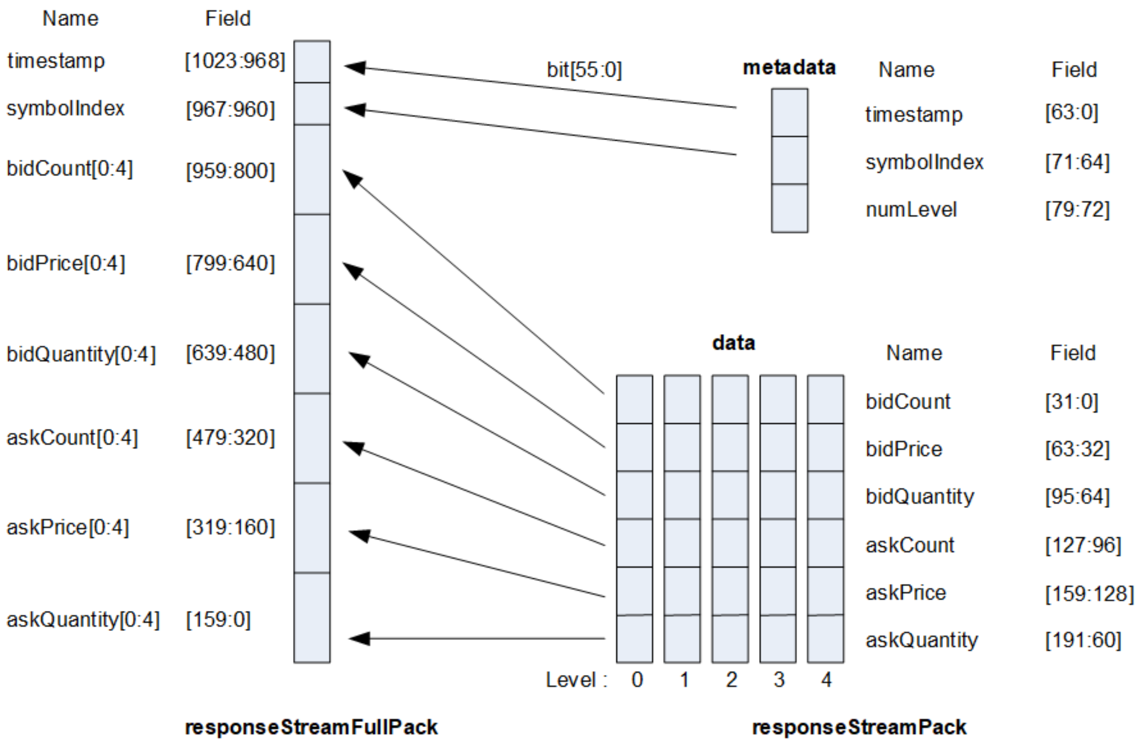


Figure 17 responseMove Packet Layout

The operation of responseMove is managed by three sequential state machines: IDLE, METADATA, and DATA.

- IDLE: This state waits for new metadata. Upon receiving a new metadata packet, the state transitions to METADATA.
- METADATA: This state decodes the metadata packet, which includes the timestamp, symbol index, and the number of levels. Once the metadata is processed, the state transitions to DATA.
- DATA: This state receives data from the OrderBook for five fixed levels (levels 0 to 4). Each level packet contains details such as bidCount, bidPrice, bidQuantity, askCount, askPrice, and askQuantity. After processing all five levels, the metadata and level data are combined into a single 1024-bit packet, which includes the lower 56 bits of timestamp. The finalized packet is then sent to QDMA Subsystem as input to the Pricing Engine in the Host system. Once the transmission is complete, the state transitions back to IDLE to await the next metadata packet.

operationMove

The operationMove manages data flow between tow AXI4-Stream interfaces:

- operationPack512: A 512-bit interface that receives trade operation data from the Pricing Engine in the Host system via the QDMA Subsystem.
- operationPack: A cropped version of operationPack512, with a width of 184 bits, containing only the required fields.

The structure of the packets processed by the operationMove is illustrated in Figure 18.

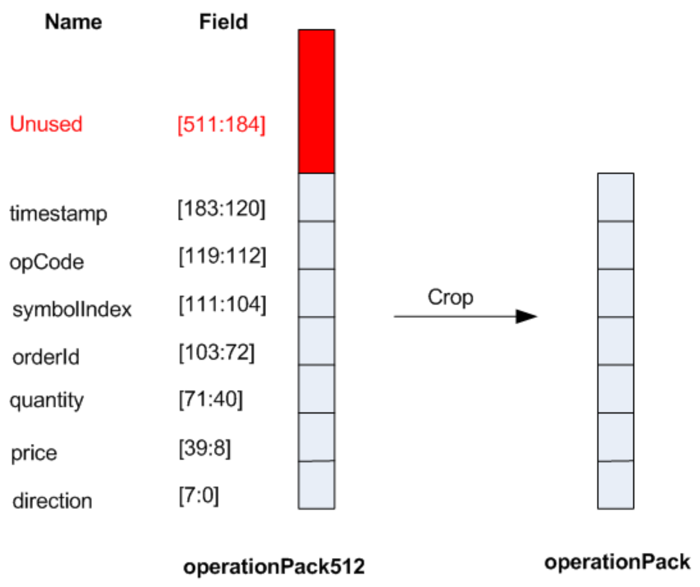


Figure 18 operationMove Packet Layout

This submodule handles downlink operations, transferring trade operation data from the Host system to the Order Entry. Due to the fixed data width of the AMD QDMA IP at 512 bits, only the lower 184 bits are relevant and transferred to the OrderEntry, while the upper bits [511:184] contain dummy data. The lower 184 bits include the following fields: timestamp, opcode, symbol index, order ID, quantity, price, and direction.

The operationMove also includes a feature to calculate the round-trip time for data transferred between the responseMove and operatonMove. This round-trip time is measured as follows.

1) The packet outputs from the responseMove contains a timestamp field, which is replaced by the current value of the internal clock counter, referred to as CountCycle (shown in Figure 16_.
2) The packet is processed by the Pricing Engine in the Host system and then sent back as a corresponding trade operation, including the updated timestamp.
3) Upon receiving the trade operation, the operationMove calculates the round-trip time by subtracting the timestamp in the packet from the current value of the internal clock (CountCycle in Figure 16).

The CountCycle counters in the responseMove and operationMove are individually designed but are synchronized by being reset to zero simultaneously at the start of operations.

2.2.6 Pricing Engine

The Pricing Engine submodule is responsible for generating bid-ask orders for specific ticker symbols based on the current state of the market and configured trading rules. It operates on updates received from the OrderBook submodule, recalculating trading actions whenever changes occur.

The Pricing Engine submodule evaluates trading conditions and generates bid/ask requests, which are then forwarded to the OrderEntry submodule for execution. It also logs generated requests, providing visibility into executed strategies for analysis.

The submodule is configurable, allowing real-time updates to trading strategies, rules, and parameters without requiring a reset. This flexibility supports rapid adaptation to changing market conditions.

2.2.7 OrderEntry

The OrderEntry submodule processes order placement requests received from the Pricing Engine submodule or DataMover submodule. It constructs order messages in the appropriate in the FIX4.2 protocol format, which includes tags for price, quantity, order ID, and active type.

After constructing the order message, the OrderEntry submodule calculates and appends a partial checksum. The resulting message is passed to the TCP submodule, which completes the encapsulation with IP and TCP headers before transmitting it over the Ethernet interface.

2.2.8 TCP

The TCP submodule implements TCP stack to establish and maintain TCP connections for transmitting order messages to a stock trading exchange. It supports up to 32 concurrent TCP connections using on-chip memory, with each connection allocated a 64 KB buffer for transmit and receive operations.

This submodule is closely integrated with the OrderEntry submodule, which generates order messages to be encapsulated with TCP headers and transmitted over the network. The TCP submodule also provides packet retransmission capabilities, ensuring reliable delivery and optimizing performance in high-frequency trading environments.

During initialization, the Host system configures key network parameters, such as the MAC address and IPv4 address, via the AXI4-Lite interface. For data transfer, the TCP submodule receives commands and data from the OrderEntry submodule to send order packets to the target system.

2.2.9 ClockTick Generator

The ClockTick Generator submodule generates periodic pulses to synchronize and trigger operations across various submodules. During initialization, the Host system configures the generator via the AXI4-Lite interface, specifying time intervals in microseconds for each submodule.

The ClockTick Generator connects to multiple components, including the Feed Handler, OrderBOok, Pricing Engine, OrderEntry, and Line Handler. By providing timing signals, it ensures coordinated processing and supports time-based execution of trading algorithms and system tasks, enabling synchronization within the hardware pipeline.

2.2.10 Loopback

The Loopback submodule provides a pass-through mechanism for Ethernet packets, enabling the verification of data paths and measurement of system latency. This module reads Ethernet packets from the Rx AXI4-Stream interface (10GbE#3) and conditionally forwards them to the Tx AXI4-Stream interface (10GbE#3) based on a control register setting.

2.3 QDMA Subsystem

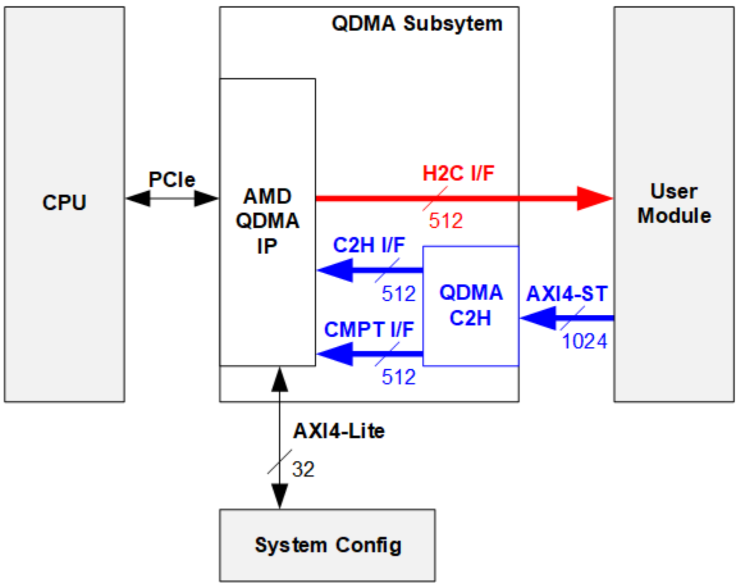


Figure 19 QDMA Subsystem Component

The QDMA subsystem facilitates high-throughput and low-latency data transfer between the FPGA and the Host system via PCIe. It integrates the AMD QDMA IP and QDMA C2H submodule, ensuring compatibility with the trading system's AXI4 Stream interfaces.

The AMD QDMA IP supports advanced features such as queue-based abstractions, which optimize the handling of small packets in dynamic trading environments. The QDMA C2H submodule adapts the trading system's data interface to the AMD QDMA IP interface, enabling efficient data flow between the hardware and the host. Further details about each submodule are provided below.

2.3.1 AMD QDMA IP

The AMD Queue Direct Memory Access (QDMA) IP provides a DMA solution optimized for PCIe. Its queue-based architecture abstracts DMA operations into software-controlled queues, simplifying the management and ensuring efficient data movement. The user configured the AMD QDMA IP with the following settings:

Basic

• Mode	: Advance
• PCIe Lane Width and Maximum Link Speed	: 16.0 GT/s x8 or 8.0 GT/s x16
• AXI Data Width	: 512 bit
• AXI Clock Frequency	: 250 MHz
• DMA Interface Selection	: AXI Stream with Completion

Capabilities

• Class Code PF0/Base Class	: Network controller
• Class Code PF0/Subclass	: Other network controller

For more information about the AMD QDMA IP, refer to the following link:

https://www.xilinx.com/products/intellectual-property/pcie-qdma.html#overview

2.3.2 QDMAC2H

The QDMAC2H bridges the interface between the DataMover and AMD QDMA-IP by converting a 1024-bit AXI4-Stream input to a 512-bit AXI4-Stream with Completion output. The DataMover provides packets with a fixed length of 128 bytes (1024 bits), sending data in one clock cycle followed by a pause of 5 clock cycles.

The QDMAC2H processes each 128-byte packet by splitting it into two 512-bit segments for transmission. During this conversion, the submodule calculates a CRC32 for 512-bit segment and transfers it to the QDMA alongside the data. After the entire 128-byte packet is transmitted to the QDMA, a completion packet is sent via the CMPT interface, signaling the end of the packet transfer.

The total processing time for each packet is 4 clock cycles, providing sufficient bandwidth to handle packet reception from the DataMover. Since the design uses a fixed packet length of 128 bytes, several output signals, such as packet length and error correction code, are assigned constant value.

Further details about the QDMAC2H are described below.

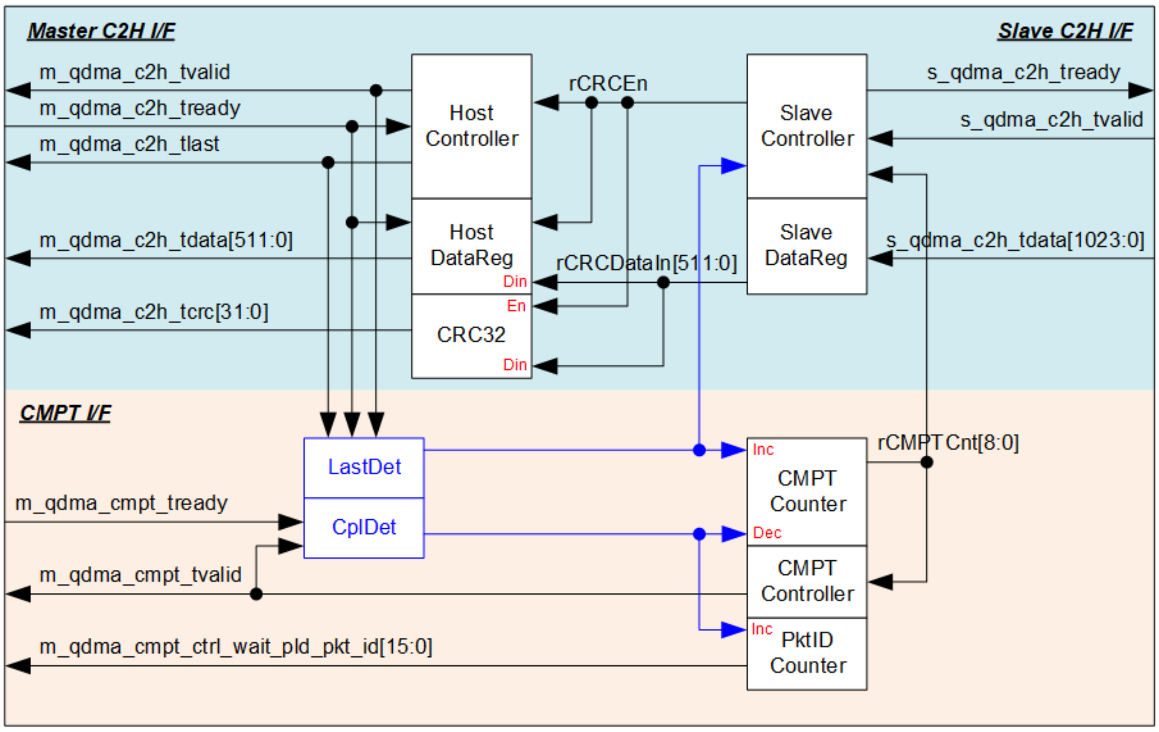


Figure 20 QDMAC2H Block Diagram

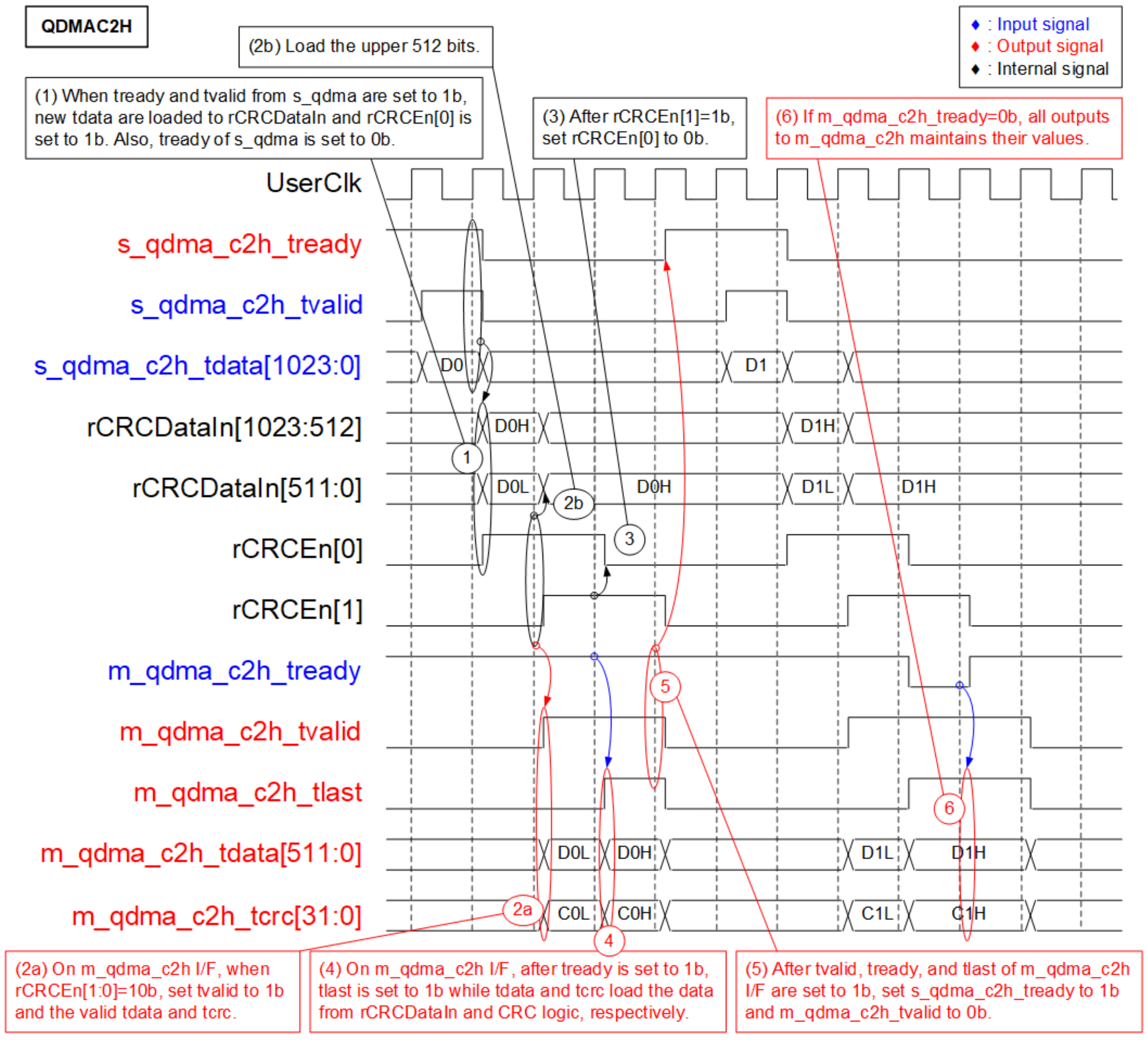
When new data is received from the DataMover via the Slave C2H I/F, the Slave controller de-asserts the ready signal to pause the transmission of the next packet. The received 1024-bit data is latched into the SlaveDataReg, where it is sliced into two 512-bit data segments. These segments are then processed in parallel for transmission to the QDMA via the HostDataReg and for CRC32 calculation via the CRC32 logic.

Once the data slicing and CRC32 calculation are completed, the two 512-bit data segments and their corresponding CRC32 values are transmitted to the QDMA. Upon transmitting the last packet of data to the QDMA, the process triggers the CMPT Logic, which generates a completion packet for the QDMA.

The CMPT Counter (9-bit) tracks the number of packets transmitted to the QDMA without generating a completion packet. It increments each time the last packet data is successfully transmitted to the QDMA and decrements when a completion packet is successfully sent. Since the AMD QDMA-IP in this design configures the number of Queues to 512, the value of rCMPTCnt should not exceed 511. To maintain this limit, if rCMPTCnt exceeds 507, the ready signal of the Slave C2H I/F is de-asserted to pause the transmission of the next packet.

The CMPT Controller generates a completion packet and transmitted it to the QDMA when the CMPTCnt output of the counter is non-zero. Additionally, the PktID Counter generates a unique packet ID for each completion packet, starting from 1 and incrementing sequentially.

Further details of the 512-bit data and CRC32 transmission process to the QDMA are illustrated in Figure 21, which provides the corresponding timing diagram.



QDMAC2H
(2b) Load the upper 512 bits.
(1) When tready and tvalid from s_qdma are set to 1b, new tdata are loaded to rCRCDataIn and rCRCEn[0] is set to 1b. Also, tready of s_qdma is set to 0b.
(3) After rCRCEn[1]=1b, set rCRCEn[0] to 0b.
(6) If m_qdma_c2h_tready=0b, all outputs to m_qdma_c2h maintains their values.
♦ : Input signal
♦ : Output signal
♦ : Internal signal
UserClk
s_qdma_c2h_tready
s_qdma_c2h_tvalid
s_qdma_c2h_tdata[1023:0]
rCRCDataIn[1023:512]
rCRCDataIn[511:0]
rCRCEn[0]
rCRCEn[1]
m_qdma_c2h_tready
m_qdma_c2h_tvalid
m_qdma_c2h_tlast
m_qdma_c2h_tdata[511:0]
m_qdma_c2h_tcrc[31:0]
D0
D1
D0H
D1H
D0L
D0H
D1L
D1H
D0L
D0H
D1L
D1H
C0L
C0H
C1L
C1H
1
2a
2b
3
4
5
6
(2a) On m_qdma_c2h I/F, when rCRCEn[1:0]=10b, set tvalid to 1b and the valid tdata and tcrc.
(4) On m_qdma_c2h I/F, after tready is set to 1b, tlast is set to 1b while tdata and tcrc load the data from rCRCDataIn and CRC logic, respectively.
(5) After tvalid, tready, and tlast of m_qdma_c2h I/F are set to 1b, set s_qdma_c2h_tready to 1b and m_qdma_c2h_tvalid to 0b.

Figure 21 QDMAC2H Timing Diagram

1) To enable data reception, s_qdma_c2h_tvalid is set to 1b. The DataMover asserts s_qdma_c2h_tvalid along with the 1024-bit data on s_qdma_c2h_tdata to send one packet to QDMAC2H. At this point, the data is loaded into rCRCDataIn, and rCRCEn[0] is asserted to 1b, initiating the CRC32 calculation for the lower 512 bits of rCRCDataIn. Simultaneously, s_qdma_c2h_tready is set to 0b, pausing the reception of new packets.

2) The result of the CRC32 calculation is forwarded to m_qdma_c2h_tcrc, while the lower 512 bits of rCRCDataIn are forwarded to m_qdma_c2h_tdata. At the same time, m_qdma_c2h_tvalid is set to 1b to start transmitting the first 512 bits of data to the AMD QDMA-IP. Simultaneously, the lower 512 bits of rCRCDataIn are updated with the upper 512 bits to prepare for the next CRC calculation and data transmission.

3) After processing the entire 1024-bit packet, the CRC32 calculation completes. This is indicated by rCRCEn[1], which is one-clock latency version of rCRCEn[0], being set to 1b. At this point, rCRCEn[0] is de-asserted to 0b.

4) Once the AMD QDMA-IP acknowledges the first 512 bits by setting m_qdma_c2h_tready to 1b, the remaining 512 bits and their corresponding CRC32 value are forwarded via m_qdma_c2h_tdata and m_qdma_c2h_tcrc, respectively. Simultaneously, m_qdma_c2h_tlast is set to 1b, indicating that the last packet data is being transmitted.

5) After the final packet data is fully transmitted, indicated by m_qdma_c2h_tvalid, m_qdma_c2h_tlast, and m_qdma_c2h_tready all being set to 1b, m_qdma_c2h_tvalid is de-asserted to 0b, completing the packet transmission. At the same time, s_qdma_c2h_tready is set to 1b to enable the reception of a new packet.

6) During the transmission of the 512-bit data to the AXI QDMA-IP, if the QDMA IP cannot accept the data (indicated by m_qdma_c2h_tready being set to 0b), all data outputs, including m_qdma_c2h_tdata, m_qdma_c2h_tcrc, m_qdma_c2h_tlast, must retain their values until the AMD QDMA-IP is ready to receive the data.

Note: The polynomial used for CRC32 calculation is as follows:

$P(X) = X^{32} + X^{26} + X^{23} + X^{22} + X^{16} + X^{12} + X^{11} + X^{10} + X^{8} + X^{7} + X^{5} + X^{4} + X^{2} + X + 1$

2.4 System Config

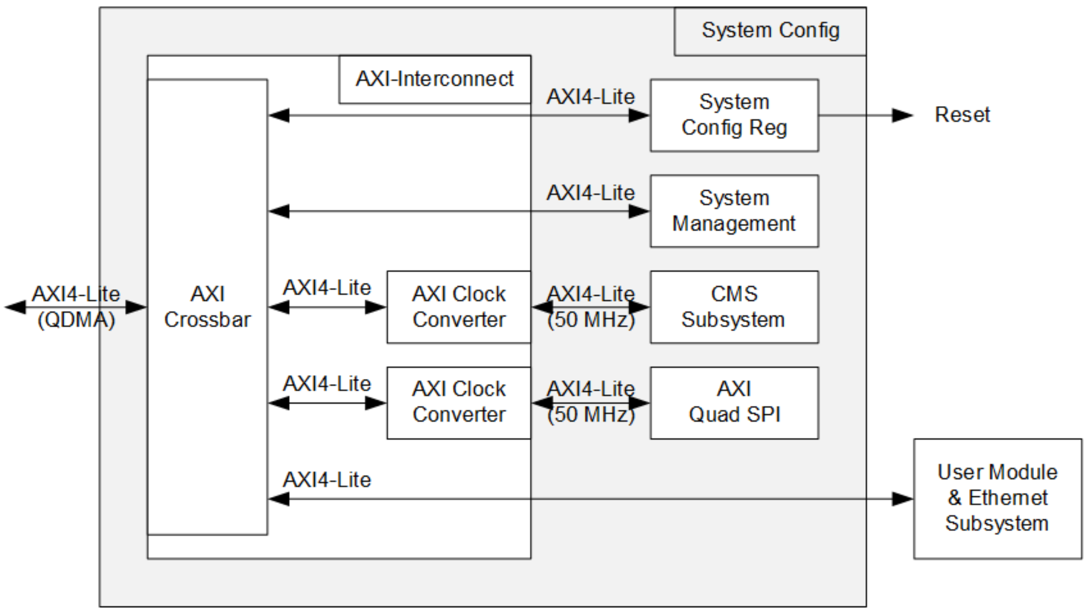


Figure 22 System Config Block Diagram

The System Config module manages system configuration signals using the AXI4-Lite bus interface, which is optimized for single transfers and is not performance-sensitive. Consequently, this module operates on axil_clk, running at a lower frequency than axis_clk and ap_clk. The module comprises several components, including the AXI-Interconnect, System Config Reg, System Management, CMS Subsystem, and AXI Quad SPI. Detailed descriptions of each component are provided below.

2.4.1 AXI-Interconnect

The AXI-Interconnect submodule serves as the interface hub for the AXI4-Lite connections. It facilitates communication between the AMD QMDA-IP (master port) and various submodules (slave ports).

Some submodules require operation at a fixed clock frequency of 50 MHz, which differs from axil_clk. To handle this, the AXI Clock Converter is included to bridge the clock domains between axil_clk and the 50 MHz clock.

More information about the AXI-Interconnect is available on the AMD website.

https://www.xilinx.com/products/intellectual-property/axi_interconnect.html#overview

The address range assigned to each slave port for accessing submodules is as follows:

1) 0x000000 – 0x00FFFF : System Config Reg
2) 0x010000 – 0x01FFFF : System Management
3) 0x040000 – 0x07FFFF : CMS Subsystem
4) 0x080000 – 0x080FFF : AXI Quad SPI
5) 0x100000 – 0x1FFFFF : User Module and Ethernet Subsystem

2.4.2 System Management Wizard

The System Management Wizard provides a solution for monitoring system parameters in AMD UltraScale devices, such as on-chip voltage and temperature. The Host system can access and monitor these values via the AXI4-Lite interface. For more information, refer to the AMD website:

https://www.xilinx.com/products/intellectual-property/system-management-wiz.html#overview

2.4.3 CMS Subsystem

The AMD Alveo Card Management Solution Subsystem (CMS Subsystem) is specifically designed for compatibility with Alveo acceleration cards. It reads sensor information from the satellite controller, such as voltages, currents, and fan speed. For U250 card, the CMD Subsystem is additionally used to control the QSFP28 reference clock input to the FPGA. For more information, refer to the AMD website:

https://www.xilinx.com/products/intellectual-property/cms-subsystem.html

2.4.4 AXI Quad SPI

In this design, the FPGA configuration file is stored in flash memory, which can be updated via the SPI interface. Using the AXI Quad SPI submodule allows the Host system to update the hardware through the AXI4-Lite interface.

The parameters of the AXI Quad SPI are configured as follows:

• Mode	: Quad
• Slave Device	: Micron (Numonyx)
• FIFIO Depth	: 256
• Flash (Memory) access through STARTUP Primitive	: Enable
• STARTUP Primitive Usage	: User STARTUP Primitive Internal to IP

For additional details, refer to the AMD website:

https://www.xilinx.com/products/intellectual-property/axi_quadspi.html.

2.4.5 System Config Reg

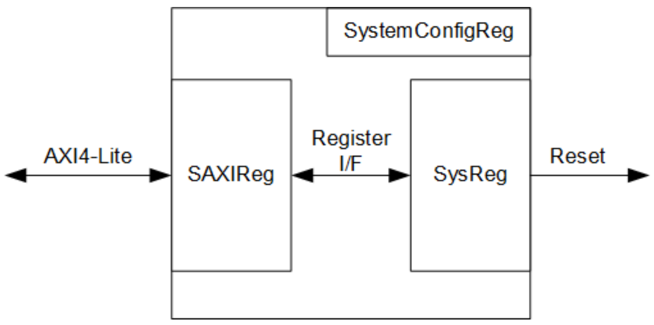


Figure 23 System Config Reg Block Diagram

The SystemConfigReg module manages reset signals for the data path of AMD QDMA-IP and the User Module. It consists of two components:

- SAXIReg: Converts the AXI4-Lite interface into a Register interface.
- SysReg: Generates reset signals for the system.

Further details about SAXIReg and SysReg are provided in the following sections.

SAXIReg

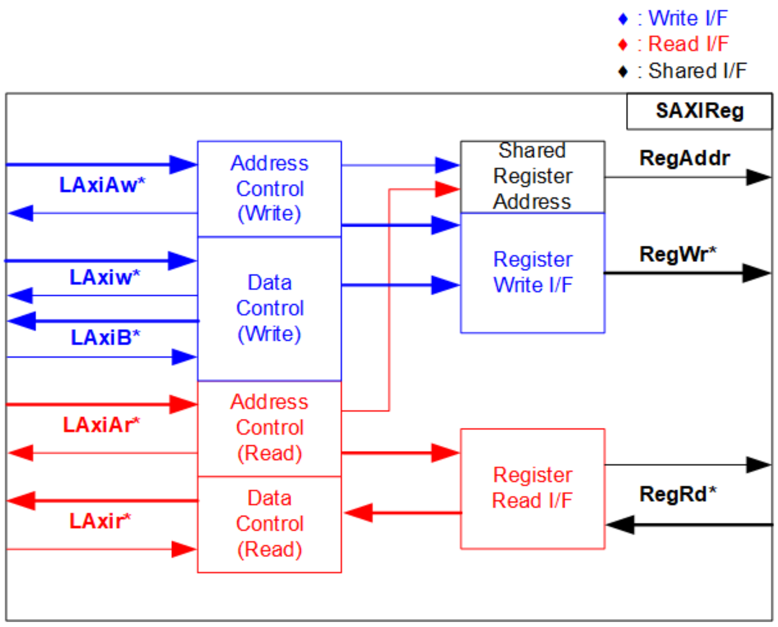


Figure 24 SAXIReg Interface

The signal on AXI4-Lite bus interface can be grouped into five groups, i.e., LAxiAw* (Write address channel), LAxiw* (Write data channel), LAxiB* (Write response channel), LAxiAr* (Read address channel), and LAxir* (Read data channel). More details to build custom logic for AXI4-Lite bus is described in following document.

https://github.com/Architech-Silica/Designing-a-Custom-AXI-Slave-Peripheral/blob/master/designing_a_custom_axi_slave_rev1.pdf

According to AXI4-Lite standard, the write channel and the read channel are operated independently for both control and data interfaces. Therefore, the logic within AXI4-Lite bus is divided into four groups, i.e., Write control logic, Write data logic, Read control logic, and Read data logic, as shown in the left side of Figure 24. The Write control I/F and Write data I/F of AXI4-Lite bus are latched and transferred to become the Write register interface. Similarly, the Read control I/F of AXI4-Lite bus is latched and transferred to the Read register interface, while Read data is returned from Register interface to AXI4-Lite. In the Register interface, RegAddr is a shared signal for write and read access, so it loads the value from LAxiAw for write access or LAxiAr for read access.

The Register interface is compatible with single-port RAM interface for write transaction. The read transaction of the Register interface has been slightly modified from RAM interface by adding the RdReq and RdValid signals to control read latency time. The address of Register interface is shared for both write and read transactions, so user cannot write and read the register at the same time. The timing diagram of the Register interface is shown in Figure 25.

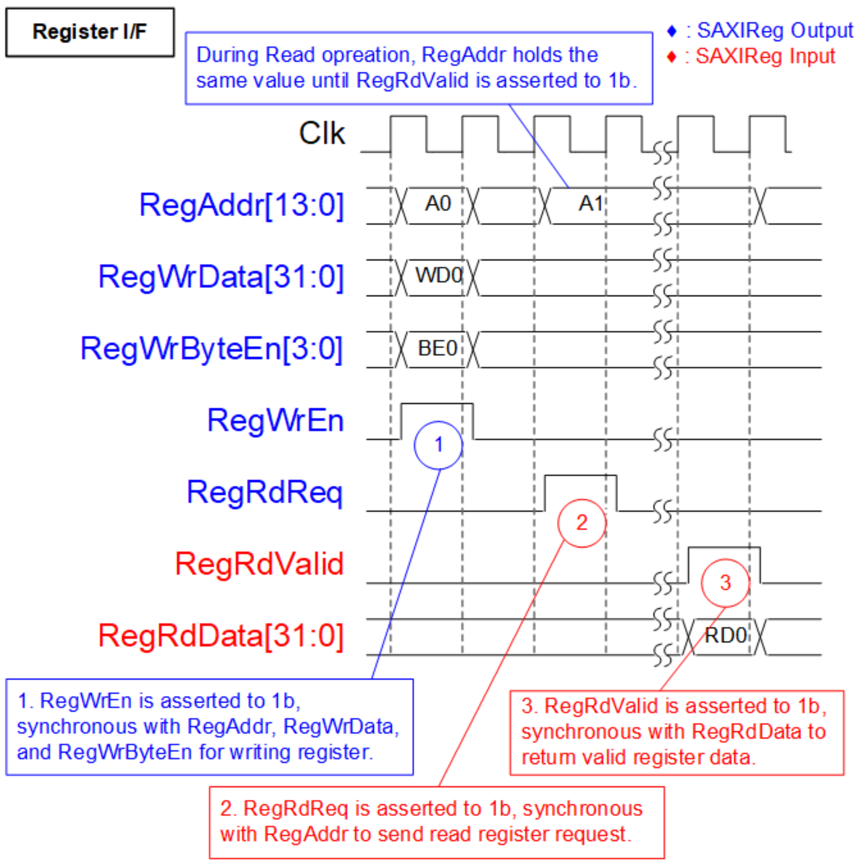

Register I/F
During Read opreation, RegAddr holds the same value until RegRdValid is asserted to 1b.
♦ : SAXIReg Output
♦ : SAXIReg Input
Clk
RegAddr[13:0]
A0
A1
RegWrData[31:0]
WD0
RegWrByteEn[3:0]
BE0
RegWrEn
RegRdReq
RegRdValid
RegRdData[31:0]
RD0
1
2
3
1. RegWrEn is asserted to 1b, synchronous with RegAddr, RegWrData, and RegWrByteEn for writing register.
3. RegValid is asserted to 1b, synchronous with RegRdData to return valid register data.
2. RegRdReq is asserted to 1b, synchronous with RegAddr to send read register request.

Figure 25 Register Interface Timing Diagram

1) Timing diagram to write register is similar to that of a single-port RAM. The RegWrEn signal is set to 1b, along with a valid value for RegAddr (Register address in 32-bit units), RegWrData (write data for the register), and RegWrByteEn (write byte enable). The byte enable consists of four bits that indicate the validity of the byte data. For example, bit[0], [1], [2], and [3] are set to 1b when RegWrData[7:0], [15:8], [23:16], and [31:24] are valid, respectively.

2) To read register, SAXIReg sets the RegRdReq signal to 1b with a valid value for RegAddr. The 32-bit data is returned after the read request is received. The slave detects the RegRdReq signal being set to start the read transaction. In the read operation, the address value (RegAddr) remains unchanged until RegRdValid is set to 1b. The address can then be used to select the returned data using multiple layers of multiplexers.

3) The slave returns the read data on RegRdData bus by setting the RegRdValid signal to 1b. After that, SAXIReg forwards the read value to the LAxir* interface.

SysReg

The SysReg module includes two registers for controlling the soft reset from the Host system. In this design, the write byte enable (RegWrByteEn) is not implemented, so the Host system must use a 32-bit pointer to access both reset registers. A read latency of this module is set to one clock cycle, achieved by asserting a D Flip-Flop triggered by the RegRdReq signal to generate the RegRdValid signal. Further details about the address mapping within the SysReg module are provided as follows.

Table 2 System Register Map

Address	Register Name	Description
0x0000 Wr/Rd	QDMA reset (QDMARST_INTREG)	[0]: Set to 1b to soft reset QDMA IP. *Note: This register is cleared by Hardware*
0x0004 Wr/Rd	AAT reset (AATRST_INTREG)	[0]: Set to 1b to soft reset User module and Ethernet Subsystem. *Note: This register is cleared by Hardware*

3 Host Software

In the AAT-QDMA demo, Host software is divided into two main regions – DPDK and AAT-QDMA application as illustrated in Figure 26.

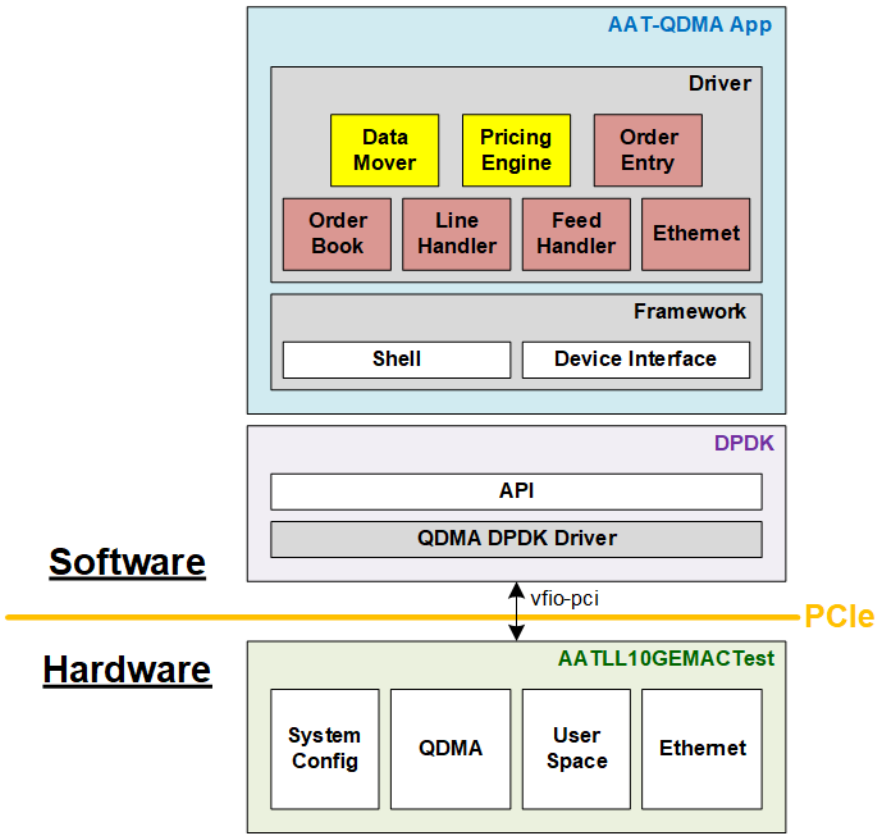


Figure 26 Host Software Structure on AAT-QDMA Demo

To enable seamless communication between hardware and software, the Data Plane Development Kit (DPDK) is employed. Using DPDK, the FPGA card (hardware) is directly accessible from the software, bypassing the traditional OS kernel procedures via high-performance I/O virtualization (vfio-pci). In this reference design, DPDK is tailored to work with the AMD QDMA IP Core on the FPGA. As a result, the QDMA DPDK driver is utilized to fulfill the Direct Memory Access (DMA) requirements. DPDK also provides an API that acts as an interface for its driver, offering a set of predefined functions or commands that allow the upper-layer software (AAT-QDMA application) to leverage DPDK's capabilities.

Built on top of DPDK, the AAT-QDMA application is responsible for managing the demo's initialization, configuration, hardware register access, and monitoring. This application operates through two key sub-blocks. The first is the Framework, which comprises a set of shared software functions typically used by other software components. The Framework includes functions for interfacing with the user terminal (Shell) and functions for interacting with DPDK for DMA operations and hardware access (Device Interface).

The second sub-block is the Driver, which serves as the interface for controlling each hardware submodule within the FPGA. When managing hardware submodules, the Driver initiates register access to configure or clear specific hardware ports. For data transfer, the Driver requests DMA operations to move data between hardware and software. To effectively manage multiple hardware blocks, the Driver is further divided into sub-drivers, each dedicated to a specific hardware module. For example, the Ethernet driver is responsible for controlling the Ethernet Subsystem in hardware, while the Feed Handler driver manages the Feed Handler within the User Module.

Similar to the hardware implementation, the Host software is pre-built to support both "Price Engine on Card" and "Price Engine Software" modes. In this design, Data Mover and Pricing Engine Drivers are always visible to the software. However, when a Driver is tasked with an operation incompatible with the configured mode ("Price Engine on Card" or "Price Engine Software"), the software either notifies the user with a failed status or returns an empty response from the hardware. These conditions do not disrupt the overall operation of the AAT-QDMA demo.

The detailed information of the Host software is described in the following topics.

3.1 DPDK

The Data Plane Development Kit (DPDK) is a collection of libraries and drivers designed to enable high-performance packet processing on CPUs. It allows user-space applications to bypass the kernel, reducing latency and overhead for faster data handling. DPDK provides direct access to network hardware through Poll Mode Drivers (PMDs) and optimizes performance with features like zero-copy mechanisms and core affinity. It is particularly effective on FPGA-powered acceleration platforms, facilitating rapid data transfer between CPUs and FPGA accelerators. For more information about DPDK, visit the DPDK website. https://www.dpdk.org/

In this reference design, the QDMA DPDK PMD is utilized, a specialize driver within the DPDK framework designed for QDMA IP, with support for Virtual Function I/O (vfio-pci).

To perform DMA operations, this design follows the workflow outlined in the "QDMA DPDK Driver Use Cases" documentation: xilinx.github.io/dma_ip_drivers/master/QDMA/DPDK/html/qdma_usecases.html#. The API functions are grouped by their functionalities, including Initialization, Queue Setup, Device Start, Device Transfer, and Clean Up. Proper execution of the Clean Up step is crucial for clearing the queue context and freeing resource.

Below are the API functions used in this demo:

Initialization

int rte_eal_init(int argc, char** argv)	
Parameters	argc: The number of command-line arguments passed to the application, including the program name. argv: An array of strings containing the command-line arguments, used to configure the DPDK Environment Abstraction Layer (EAL) initialization.
Return value	≥0: Success, indicating the EAL has been successfully initialized. -1: Failure, indicating an error occurred during initialization.
Description	Initializes the Environment Abstraction Layer (EAL), which sets up the DPDK runtime environment. This function prepares resources such as hugepages, memory, and CPU core bindings required for high-performance packet process. It must be called before any other DPDK API functions are used.

Queue Setup

int rte_eth_tx_queue_setup(uint16_t pord_id, uint16_t tx_queue_id, uint16_t nb_tx_desc, unsigned int socket_id, const struct rte_eth_txconf* tx_conf)	
Parameters	port_id: The port identifier of the Ethernet device for which the transmit queue is being set up. tx_queue_id: The index of the transmit queue to configure. nb_tx_desc: The number of transmit descriptors to allocate for the transmit ring. socket_id: The socket identifier for NUMA (Non-Uniform Memory Access) configuration. tx_conf: A pointer to the configuration structure ("rte_eth_txconf") for the transmit queue.
Return value	0: Success, indicating the transmit queue has been correctly set up. Other: Error, indicating an issue occurred during queue setup.
Description	Allocates and sets up a transmit queue for an Ethernet device. This function configures the transmit ring and ensures the queue is ready to send packets. It must be called before starting the Ethernet device using "rte_eth_dev_start".

int rte_eth_rx_queue_setup(uint16_t pord_id, uint16_t rx_queue_id, uint16_t nb_rx_desc, unsigned int socket_id, const struct rte_eth_rxconf* rx_conf, struct rte_mempool* mb_pool)	
Parameters	port_id: The port identifier of the Ethernet device for which the receive queue is being set up. rx_queue_id: The index of the receive queue to configure. nb_rx_desc: The number of receive descriptors to allocate for the receive ring. socket_id: The socket identifier for NUMA configuration. rx_conf: A pointer to the configuration structure ("rte_eth_rxconf") for the receive queue. mb_pool: A pointer to the memory pool ("rte_mempool") used for allocating buffers for received packets.
Return value	0: Success, indicating the receive queue has been correctly set up. Other: Error, indicating an issue occurred during queue setup.
Description	Allocates and sets up a receive queue for an Ethernet device. This function configures the receive ring and associated it with a memory pool for packet buffers. The queue must be set up before starting the Ethernet device with "rte_eth_dev_start".

Device Start

int rte_eth_dev_start(uint16_t port_id)	
Parameters	port_id: The port identifier of the Ethernet device to start.
Return value	0: Success, the device has been successfully started. Others: Error code of the driver device start function.
Description	Activates the Ethernet device. This function enables the RX and TX queues for packet processing and activates the data path. It must be called after the queues have been set up using "rte_eth_tx_queue_setup" and "rte_eth_rx_queue_setup".

Data Transfer

uint16_t rte_eth_tx_burst(uint16_t port_id, uint16_t queue_id, struct rte_mbuf** tx_pkts, uint16_t nb_pkts)	
Parameters	port_id: The port identifier of the Ethernet device. queue_id: The index of the transmit queue to use. tx_pkts: An array of nb_pkts pointers to "rte_mbuf" structures containing the packets to transmit. nb_pkts: the maximum number of packets to transmit.
Return value	The number of packets successfully transmitted. This can be less than "nb_pkts" if the transmit queue is full or the device cannot handle more packets.
Description	Sends a burst of packets on a specific transmit queue of an Ethernet device. The function processes as many as possible, up to the specified "nb_pkts", and returns the number of packets actually sent.

uint16_t rte_eth_rx_burst(uint16_t port_id, uint16_t queue_id, struct rte_mbuf** tx_pkts, uint16_t nb_pkts)	
Parameters	port_id: The port identifier of the Ethernet device. queue_id: The index of the receive queue to use. rx_pkts: An array of pointers to "rte_mbuf" structures where the received packets will be stored. nb_pkts: The maximum number of packets to retrieve from the Rx queue.
Return value	The number of packets successfully received and stored in "rx_pkts".
Description	Retrieve a burst of received packets from a specific RX queue of an Ethernet device. The function processes as many packets as available, up to the specified "nb_pkts", and returns the number of packets received. Ensure that "rx_pkts" is large enough to store all received packets, as defined by "nb_pkts".

Clean Up

uint16_t rte_eth_dev_stop(uint16_t port_id)	
Parameters	port_id: The port identifier of the Ethernet device to stop.
Return value	0: Success, the Ethernet device has been successfully stopped. Other: Error code of the driver device stop function
Description	Stop the Ethernet device. This function disables the TX and RX queues and stops packet processing. It does not free the resource allocated during initialization or queue setup.

int rte_eth_dev_close(uint16_t port_id)	
Parameters	port_id: The port identifier of the ethernet device to close.
Return value	0: Success, the port has been successfully closed. <0: Error.
Description	Close the Ethernet device. This function releases all resources allocated to the device, including memory buffers and queue contexts. It should be called after "rte_eth_dev_stop" to clean up and free resources properly.

3.2 AAT-QDMA Framework

The AAT-QDMA Framework gathers common software functions, providing a structured foundation for efficient and reusable operations across the AAT system. It is divided into two main sections: Shell, for command-line interactions, and Device Interface, for hardware operations.

3.2.1 Shell

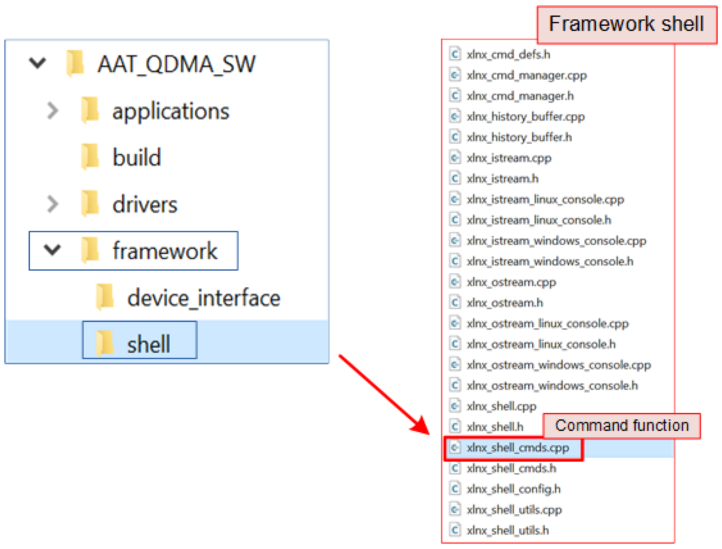


Figure 27 AAT_QDMA Shell

The Shell is a collection of source code linked to the AAT-QDMA shell commands, offering 10 executable functions within the AAT-QDMA console. These functions enable users to interact directly with the hardware, allowing for control, configuration, and monitoring operations. The functions are implemented in the file "xlnx_shell_cmds.cpp", as illustrated in Figure 27.

Table 3 lists the available shee commands and their corresponsed functions:

Table 3 AAT-QDMA Framework Shell Commands

Command	Argument	Description
help	<object>	Prints a list of objects and commands
exit	none	Exits the shell instance
quit	none	Exits shell instance
history	none	Lists previously executed commands
regrd	<addr>	Reads a single 32-bit register
regwr	<addr> <value>	Writes to a single 32-bit register
regwrm	<addr> <value> <mask>	Writes to a register with a mask
reghexdump	<addr>	Performs a hex dump of registers starting at <addr>
run	<filepath>	Executes a script containing shell commands
touch	<filepath>	Change file timestamps

3.2.2 Device Interface

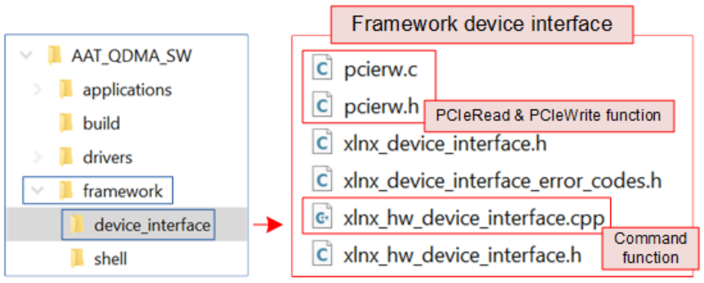

AAT_QDMA_SW
applications
build
drivers
framework
device_interface
shell
Framework device interface
pcierw.c
pcierw.h
PCIeRead & PCIeWrite function
xlnx_device_interface.h
xlnx_device_interface_error_codes.h
xlnx_hw_device_interface.cpp
xlnx_hw_device_interface.h
Command function

Figure 28 AAT-QDMA Device Interface

The Device Interface facilitates hardware communication by providing functions to write and read registers, which are extensively utilized throughout the AAT-QDMA application. It defines seven primary functions for hardware interaction, all declared in the file “xlnx_hw_device_interface.cpp”. The directory structure for this file is shown in Figure 28

Below is the list of functions provided by the Device Interface:

HWDeviceInterface::HWDeviceInterface()	
Parameters	None
Return value	None
Description	Constructor that initializes the class instance and sets default values.

HWDeviceInterface::~HWDeviceInterface()	
Parameters	None
Return value	None
Description	Destructor that cleans up dynamically allocated resources.

uint32_t HWDeviceInterface::ReadReg32(uint64_t address, uint32_t* value)	
Parameters	address: Address of the register value: Pointer to store the read value
Return value	XLNX_OK (defined as 0x00000000) on success, or an error code
Description	Reads a 32-bit register value from the specified hardware address.

uint32_t HWDeviceInterface::WriteReg32(uint64_t address, uint32_t value)	
Parameters	address: Address of the register value: Value to be written
Return value	XLNX_OK (defined as 0x00000000) on success, or an error code
Description	Writes a 32-bit value to the specified hardware address.

uint32_t HWDeviceInterface::WriteRegWithMask32(uint64_t address, uint32_t value, uint32_t mask)	
Parameters	address: Address of the register value: Value to be written mask: Bitmask specifying the bits to modify
Return value	XLNX_OK (defined as 0x00000000) on success, or an error code
Description	Writes a 32-bit value to a register, applying a bitmask to modify only specific bits.

uint32_t HWDeviceInterface::BlockReadReg32(uint64_t address, uint32_t* buffer, uint32_t numWords)	
Parameters	address: Starting address buffer: Pointer to buffer for storing read values numWords: Number of 32-bit words to read
Return value	XLNX_OK (defined as 0x00000000) on success, or an error code
Description	Reads multiple 32-bit values from consecutive hardware registers into a buffer.

uint32_t HWDeviceInterface::BlockWriteReg32(uint64_t address, uint32_t* buffer, uint32_t numWords)	
Parameters	address: Starting address buffer: Pointer to buffer containing values to write numWords: Number of 32-bit words to write
Return value	XLNX_OK (defined as 0x00000000) on success, or an error code
Description	Writes multiple 32-bit values from a buffer to consecutive hardware registers.

The above functions rely on the “PciRead” and “PciWrite” operations from the QDMA DPDK driver, enabling direct access to hardware registers. These internal functions, excluded from the standard API, are implemented in “pciew.c” and declared in “pciew.h” files. The directory structure for these files is also shown in Figure 28.

unsigned int PciRead(unsigned int bar, unsigned int offset, int port_id)	
Parameters	bar: Base address register (starting address) offset: Offset within the BAR port_id: Port ID for the PCIe device
Return value	The value read from the specified BAR and offset on the PCIe device.
Description	Reads a 32-bit register value from the given BAR and offset on the PCIe device.

void PciWrite(unsigned int bar, unsigned int offset, unsigned int reg_val, int port_id)	
Parameters	bar: Base address register (starting address) offset: Offset within the BAR reg_val: Value to write to the register port_id: Port ID for the PCIe device
Return value	None
Description	Writes a 32-bit value to the specified BAR and offset on the PCIe device.

3.3 AAT-QDMA Driver

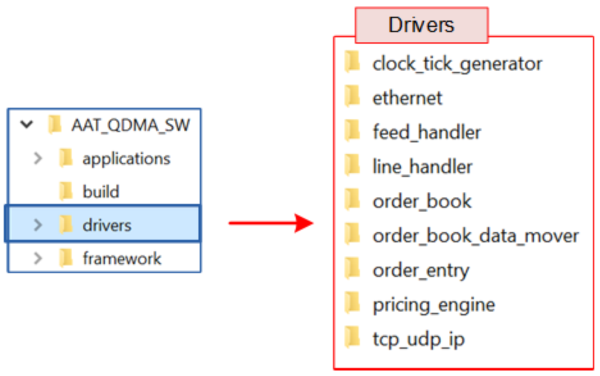


Figure 29 AAT-QDMA Driver

The AAT-QDMA Driver is a software layer designed to control and monitor hardware components of the AAT-QDMA system. It performs write and read operations to the target hardware registers, enabling configuration, status retrieval, and operational control.

The driver is organized into sub-drivers, each dedicated to managing a specific hardware block, such as Clock Tick Generator, DataMover, or OrderEntry modules. Each sub-driver comprises a driver implementation and a corresponding shell. The driver provides low-level functions to interact with hardware registers. The shell offers a high-level interface for user commands, simplifying configuration and debugging.

Detailed descriptions of each sub-driver and its shell are provided in the following sections.

3.3.1 ClockTick Generator

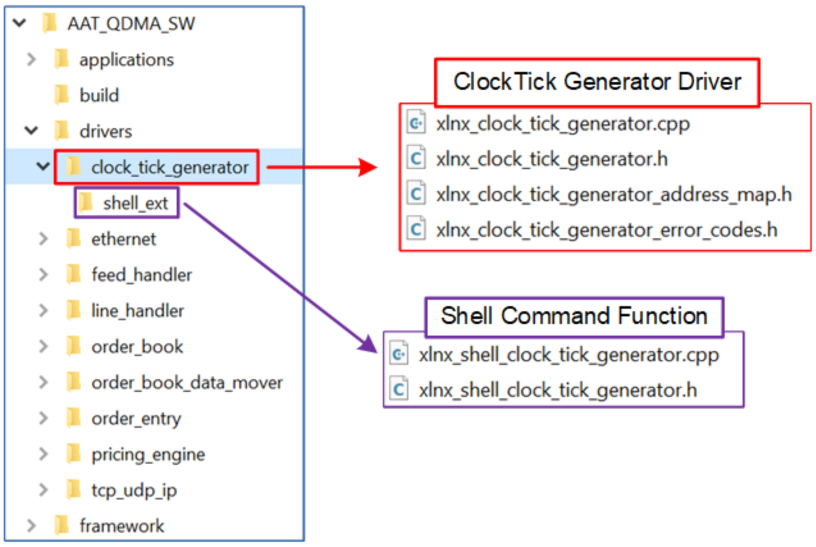

AAT_QDMA_SW
applications
build
drivers
clock_tick_generator
shell_ext
ethernet
feed_handler
line_handler
order_book
order_book_data_mover
order_entry
pricing_engine
tcp_udp_ip
framework
ClockTick Generator Driver
xlnx_clock_tick_generator.cpp
xlnx_clock_tick_generator.h
xlnx_clock_tick_generator_address_map.h
xlnx_clock_tick_generator_error_codes.h
Shell Command Function
xlnx_shell_clock_tick_generator.cpp
xlnx_shell_clock_tick_generator.h

Figure 30 ClockTick Generator Driver

The ClockTick Generator driver enables control and monitoring of programmable tick events for the AAT-QDMA system. It supports configuring tick intervals, enabling or disabling specific tick streams, resetting internal counters without triggering output, and retrieving block status. The driver interacts with hardware registers implemented in "xlnx_clock_tick_generator.cpp", while its shell commands for user interaction are defined in "xlnx_shell_clock_tick_generator.cpp".

The commands available in the shell for managing the ClockTick Generator are listed in Table 4.

Table 4 ClockTick Generator shell command

Command	Argument	Description
setenable	<index> <bool>	Enable or disable a specific tick stream
setinterval	<index> <usecs>	Set the tick event interval in microseconds
getstatus	None	Retrieve the block status

3.3.2 Ethernet

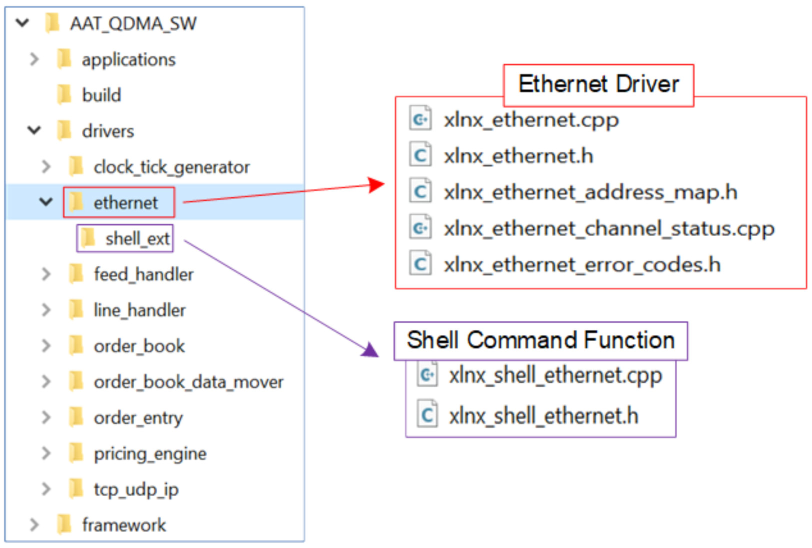


Figure 31 Ethernet Driver

The Ethernet driver manages the Ethernet hardware by performing register read and write operations. It enables monitoring, configuration, and control of Ethernet functionality, such as retrieving status, printing diagnostics, and resetting latches for each channel. The driver interacts with the hardware via registers defined in “xlnx_ethernet_address_map.h”, while error codes are specified in “xlnx_ethernet_error_codes.h”. The core driver is implemented in “xlnx_ethernet.cpp” and “xlnx_ethernet_channel_status.cpp”, and user interaction is facilitated through shell commands in “xlnx_shell_ethernet.cpp”.

The command available in the shell for managing the Ethernet are listed in Table 5.

Table 5 Ethernet Shell Command

Command	Argument	Description
getstatus	None	Retrieve the status of Ethernet channels
clearlatches	<channel>	Clear/reset status latches for the specified channel
gethwdebuginfo	<channel>	Retrieve detailed debugging information for a specific channel

3.3.3 Feed Handler

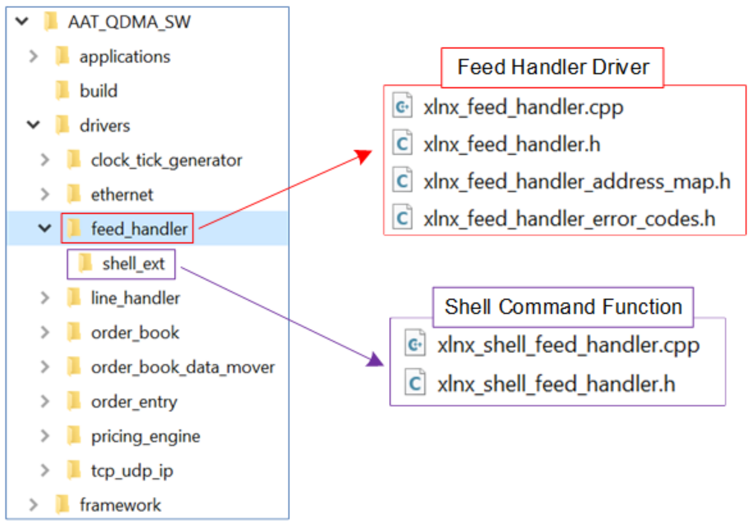


Figure 32 Feed Handler Driver

The Feed Handler software manages securities and trading updates for CME trading using register write and read operations. It enables adding, deleting, and refreshing securities, controls the start, stop, or reset operations of the hardware block, and provides monitoring for hardware status and statistics. The core functionality is implemented in “xlnx_feed_handler.cpp”, while shell commands for user interaction are defined in “xlnx_shell_feed_handler.cpp”.

The Feed Handler shell command facilitate interaction with the hardware. The command list is provided in Table 6.

Table 6 Feed handler Shell Command

Command	Argument	Description
add	<securityID>	Add a security at the first available index
addat	<securityID> <index>	Add a security at a specific index
delete	<securityID>	Delete a security
deleteall	None	Delete all securities
start	None	Start block processing
stop	None	Stop block processing
getstatus	None	Get block status
list	None	Print a list of securities
refresh	None	Refresh the internal security ID cache
resetstats	None	Reset statistics counters
readdata	None	Read the last captured data

3.3.4 Line Handler

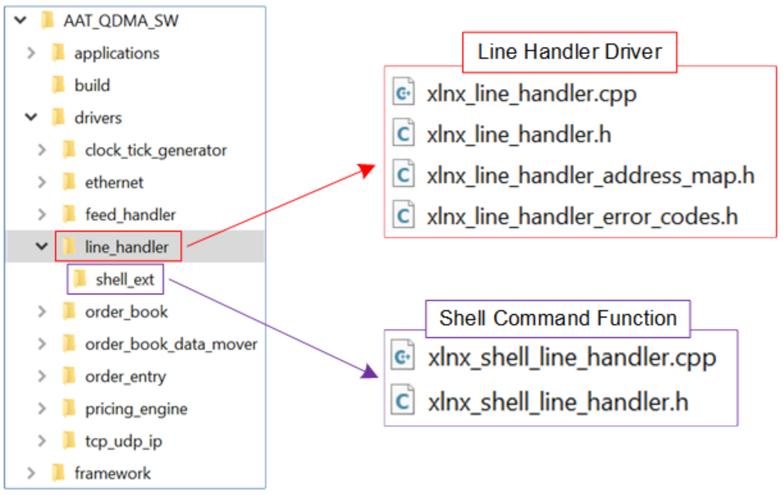


Figure 33 Line Handler Driver

The Line Handler software manages UDP multicast feeds by interacting with hardware control registers. It supports advanced configuration and monitoring capabilities, including multicast filter management, reliability modes, sequence timing, and debug features like traffic echoing. The main driver is implemented in “xlinx_line_handler.cpp”, while the shell commands for user interaction are defined in “xlnx_shell_line_handler.cpp”, while the shell commands for user interaction are defined in “xlnx_shell_line_handler.cpp”.

The Line Handler shell commands facilitate user interaction and are summarized in Table 7.

Table 7 Line Handler Shell Command

Command	Argument	Description
getstatus	None	Retrieve the Line Handler block status
add	<inputport> <ipaddr> <port> <splitID>	Add a multicast filter for a port
delete	<inputport> <ipaddr> <port> <splitID>	Delete a multicast filter from a port
deleteall	<inputport>	Clear all multicast filters for a port
setechoenabled	<inputport> <bool>	Enable or disable debug traffic echo
setechodest	<inputport> <ipaddr> <port>	Set UDP destination for debug echo
setsequencetimer	<microseconds>	Set the sequence reset timer
resetsequence	None	Reset the sequence number
sethighreliability	<bool>	Enable or disable high reliability mode
setspooltimerlimit	<microseconds>	Set spool timeout limit before aborting
setspoolpacketlimit	<numpackets>	Set spool packet limit before aborting
setdequeuethrottle	<throttlerate>	Sets the dequeue throttle rate (in clock cycles)

3.3.5 OrderBook

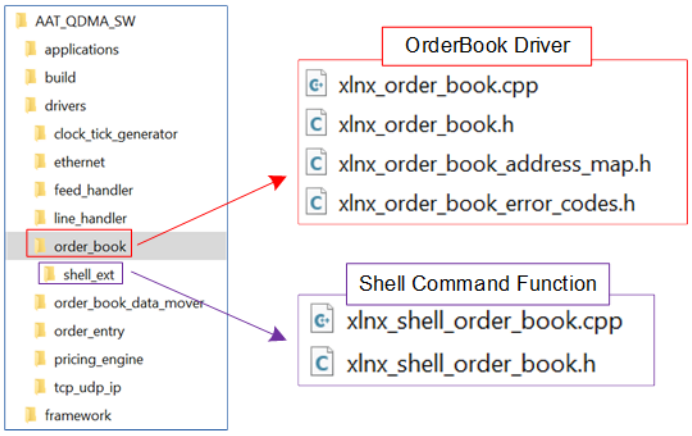


Figure 34 OrderBook Driver

The OrderBook driver manages the control and monitoring of the hardware orderbook functionality. It enables starting and stopping the processing block, configuring the capture index for ticker symbols, and enabling or disabling output to the DataMover submodule. The software also facilitates status monitoring, reading aggregated order data, and resetting statistical counters. The driver implementation is in “xlnx_order_book.cpp”, while the shell commands for user interaction are defined in “xlnx_shell_order_book.cpp”.

The OrderBook shell commands allow interaction with the hardware and are listed in Table 8.

Table 8 OrderBook Shell Command

Command	Argument	Description
getstatus	None	Get the status of the OrderBook block
readdata	None	Read aggregated order book data
resetstats	None	Reset statistical counters
setcaptureindex	<symbolindex>	Set the symbol index for data filtering
setdmoutput	<bool>	Enable or disable output to the DataMover submodule
start	None	Start the OrderBook processing block
stop	None	Stop the OrderBook processing block

3.3.6 DataMover

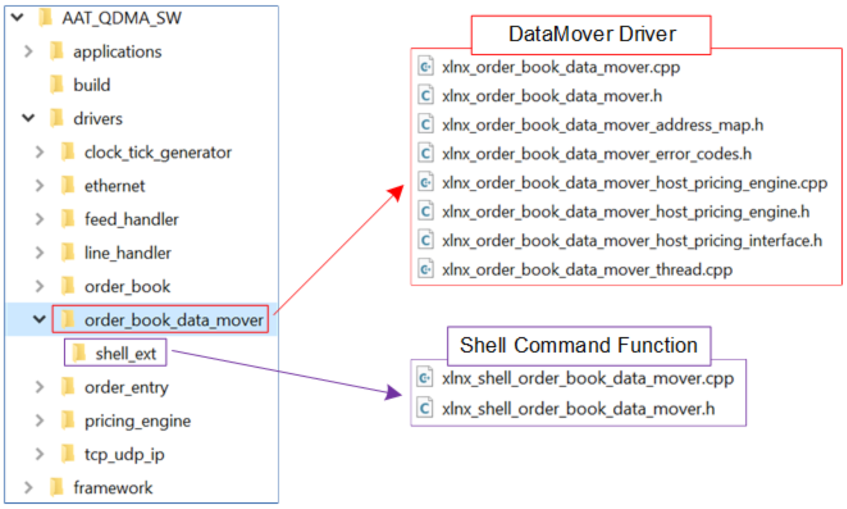

AAT_QDMA_SW
applications
build
drivers
clock_tick_generator
ethernet
feed_handler
line_handler
order_book
order_book_data_mover
shell_ext
order_entry
pricing_engine
tcp_udp_ip
framework
DataMover Driver
xlnx_order_book_data_mover.cpp
xlnx_order_book_data_mover.h
xlnx_order_book_data_mover_address_map.h
xlnx_order_book_data_mover_error_codes.h
xlnx_order_book_data_mover_host_pricing_engine.cpp
xlnx_order_book_data_mover_host_pricing_engine.h
xlnx_order_book_data_mover_host_pricing_interface.h
xlnx_order_book_data_mover_thread.cpp
Shell Command Function
xlnx_shell_order_book_data_mover.cpp
xlnx_shell_order_book_data_mover.h

Figure 35 DataMover Driver

The DataMover software facilitates data transfer between the OrderBook hardware and the Host Pricing Engine, using DPDK APIs to communicate with the QDMA IP. The software enables comprehensive control over DMA operations, including configuring DMA chunk sizes, managing processing threads, and resetting DMA statistics. It also supports debugging with verbose output and performs round-trip timing tests to evaluate communication efficiency. The main driver is implemented in “xlnx_order_book_data_mover.cpp”, while the shell commands are defined in “xlnx_shell_order_book_data_mover.cpp”.

The DataMover operates exclusively in “Pricing Engine Software” mode, where the host pricing engine processes received OrderBook data and sends operations back to the DataMover submodule for execution. The DMA operations are handled by the “Threadfunc” function, implemented in “OrderBookDataMover::ThreadFunc”, declared in “xlnx_order_book_data_mover_thread.cpp”.

The sequence of “ThreadFunc” operations is outlined below.

<u>Initialization</u>

1) Define DPDK parameters such as the memory pool (mbuf_pool), port ID (port_id), number of descriptors (nb_descs), number of queues (num_queues), maximum packets per command (max_burst_size), and maximum data size per descriptor (descs_size).
2) Allocate memory for storing packet buffers using “rte_pktmbuf_pool_create”.
3) Reserve a buffer from the memory pool (rte_pktmbuf_alloc) for transmitting data.
4) Configure the data length and packet length of the transfer memory buffer using “rte_pktmbuf_data_len” and “rte_pktmbuf_pkt_len”.
5) Set up the Ethernet device with one Rx queue and one Tx queue using “rte_eth_dev_configure” and start it with “rte_eth_dev_start”.
6) Initialize the Rx and Tx queues using “rte_eth_rx_queue_setup” and “rte_eth_tx_queue_setup”, linking them to the memory pool.
7) Activate the Ethernet device for data transmission and reception using rte_eth_dev_start.
8) Reset the thread statistics (m_threadStats) to track received, transmitted, and unexecuted packets.

Active Processing

1) Enter the main loop to continuously poll for received packets using "rte_eth_rx_burst".
2) Wait for received packets, unpack each packet using "UnpackResponse" to extract response data into a structured format, and pass the response to ProcessPricingData, which evaluates the data to determine whether a valid operation is required.
3) If the host pricing engine determines the operation is valid (bOperationValid = 1), pack the operation using "PackOperation" and transmit it to the hardware via rte_eth_tx_burst. Update statistics to reflect the number of transmitted packets ("m_threadStats.numTxPackets"). If the operation is not valid, increment the count of unexecuted operations ("m_threadStats.numNoExecutions").
4) If m_bYield is set, yield the thread to allow other processes to execute using "std::this_thread::yield". Otherwise, the thread keeps running until it is stopped by the system or finishes its task.

Clean up

1) Stop the Ethernet device by halting its operations using " rte_eth_dev_stop".
2) Close the Ethernet device and release its resources using "rte_eth_dev_close".

The Host Pricing Engine software can be customized in "HostPricingEngine::PricingProcess" function which is located in "xlnx_order_book_data_mover_host_pricing_engine.cpp" file, making it easy to adapt to specific needs. The file destination is illustrated in Figure 35. The shell command associate with the Data Mover is listed on Table 9.

Table 9 DataMover Shell Command

Command	Argument	Description
getstatus	None	Retrieve the DataMover block status
start	None	Start DataMover operations
timing	None	Perform round-trip latency measurement
verboseon	None	Enable detailed debug logging
verboseoff	None	Disable detailed debug logging
threadstart	None	Start the pricing engine thread
threadstop	None	Stop the pricing engine thread
threadyield	<bool>	Enable or disable thread yielding
resetdmastats	None	Reset DMA statistics counters
sethwemupolldelay	<seconds>	Set polling delay for hardware emulation mode

3.3.7 OrderEntry

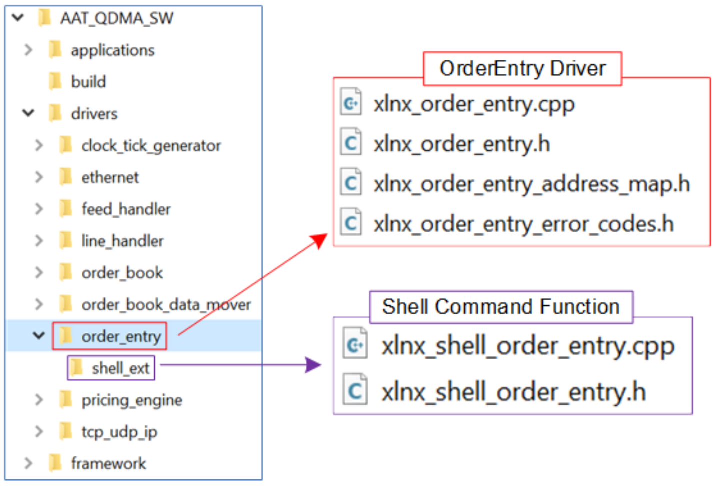


Figure 36 OrderEntry Driver

The Order Entry software manages the creation and transmission of trading orders by interacting with hardware control registers. It facilitates configuring connections to remote systems, generating partial checksums, monitoring hardware status, and resetting statistical counters. The software can also read the last emitted message and establish or reconnect TCP connections. The driver implementation is in “xlnx_order_entry.cpp”, while the user commands are defined in “xlnx_shell_order_entry.cpp”.

The OrderEntry shell commands provide a high-level interface for user interaction. Table 10 summarizes the available commands.

Table 10 OrderEntry Shell Command

Command	Argument	Description
getstatus	None	Retrieve block status
readmsg	None	Read the last emitted message
resetstats	None	Reset statistics counters
connect	<ipaddr> <port>	Establish connection to a remote system
disconnect	None	Close the connection to the remote system
reconnect	None	Close and re-open an existing connection
setcsumgen	<bool>	Enable or disable partial checksum generation

3.3.8 Pricing Engine

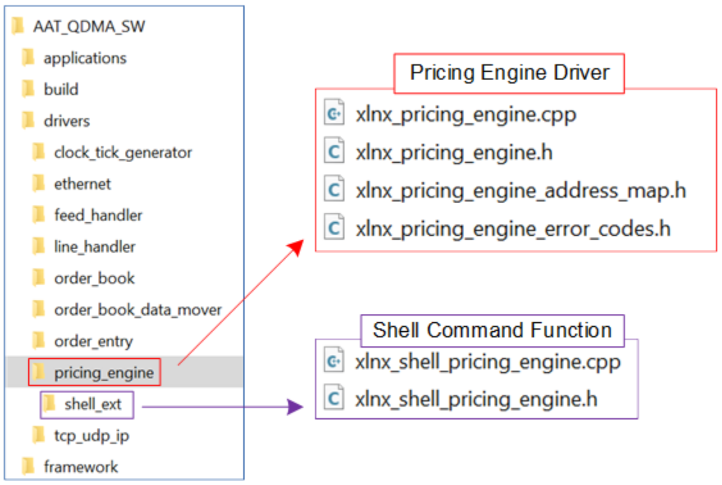


Figure 37 Pricing Engine Driver

The Pricing Engine software manages and monitors trading strategies and rules for specified symbols by interacting with hardware control registers. It supports configuring global trading modes and strategies, monitoring block status, reading trading data, and resetting statistical counters. These capabilities enable dynamic control of trading logic and parameters based on Order Book updates. The driver implementation is in “xlnx_pricing_engine.cpp”, while user commands are defined in “xlnx_shell_pricing_engine.cpp”.

The Pricing Engine shell provides a user interface for configuration and monitoring. Table 11 summarizes the available commands.

Table 11 Pricing Engine Shell Command

Command	Argument	Description
setglobalmode	<bool>	Enables or disable global pricing strategy
setglobalstrategy	<none|peg|limit>	Set the global trading strategy applied to all symbols
getstatus	None	Retrieve the block status
readdata	None	Read trading data for the last operation
resetstats	None	Reset statistical counters

3.3.9 TCP/UDP IP

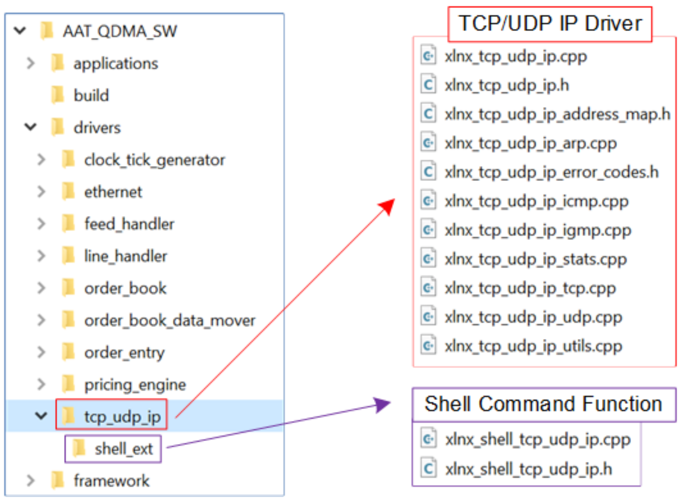


Figure 38 TCP/UDP IP Driver

The TCP/UDP IP software provides comprehensive control and monitoring of TCP and UDP networking functionalities by interacting with hardware registers. It supports advanced features like IGMP (Internet Group Management Protocol), ARP (Address Resolution Protocol), ICMP (Internet Control Message Protocol), and multicast operations, in addition handling network configuration, data transfer, and statistics monitoring.

The driver implementation is divided into multiple protocol-specific modules, including ARP, ICMP, IGMP, and UDP, with core functionality defined in "xlnx_tcp_udp_ip.cpp". User commands are grouped into four categories: Network Parameter Configuration, UDP Configuration, ARP Configuration and Block Management, all defined in "xlnx_shell_tcp_udp_ip.cpp".

<u>Network Parameter Configuration</u>

This category provides commands to configure basic network parameters like the MAC address, IPv4 address, subnet mask, and gateway address. Table 12 summarizes these commands.

Table 12 Network Config Shell Command

Command	Argument	Description
setmacaddr	<macaddr>	Set the MAC address
setipaddr	<ipaddr>	Set the IPv4 address
setsubnetmask	<ipmask>	Set the subnet mask
setgateway	<ipaddr>	Set the gateway IPv4 address

UDP Configuration

The UDP configuration commands allow management of IGMP settings, multicast address, ICMP, and UDP listening ports. Table 13 lists the commands.

Table 13 UDP Config Shell Command

Command	Argument	Description
setigmp	<bool>	Enable or disable IGMP
setigmpver	<ver>	Set the IGMP version (2 or 3)
igmpprint	<ipaddr>	Print the contents of the IGMP table
addmcast	<ipaddr>	Add a multicast IP address
deletemcast	<ipaddr>	Delete a multicast IP address
seticmp	<bool>	Enable or disable ICMP
addport	<portnum>	Add a UDP listening port
deleteport	<portnum>	Delete a UDP listening port
deleteallports	None	Deletes all UDP listening ports

APR Configuration

This category manages the ARP table, allowing users to print, add, delete, or clear ARP entries. Table 14 summarizes the ARP configuration commands.

Table 14 ARP Config Shell Command

Command	Argument	Description
arpprint	None	Print the contents of the ARP table
arpadd	<ipaddr> <macaddr>	Add an ARP entry
arpdelete	<index>	Delete an ARP entry at the specified index
arpdeleteall	None	Delete all ARP entries

Block Management

Block management commands provide control over statistics monitoring and configuration settings. Table 15 summarizes these commands.

Table 15 Block Management Shell Command

Command	Argument	Description
printstats	None	Prints protocol-specific statistics counters
getstatus	None	Retrieve the block's status information
setconfigallowed	<bool>	Enable or disable configuration writes

3.4 AAT-QDMA Application

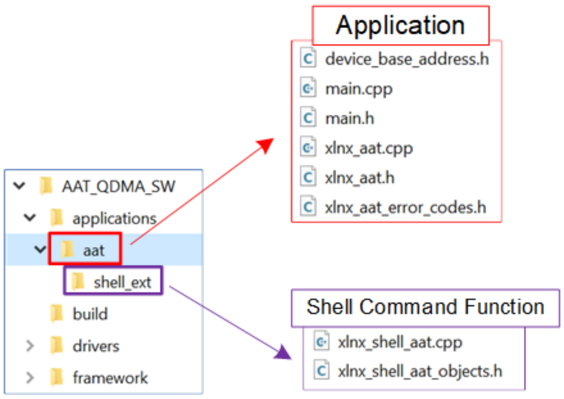


Figure 39 Application

The AAT-QDMA Application provides a unified platform for controlling and monitoring hardware components through a command-based interface. It integrates various hardware drivers to offer seamless interaction between software and FPGA-based hardware for trading applications. The application is divided into two key components. First is "Shell Command Interface", a command-line interface for executing user-defined operations on the AAT-QDMA system. Second is "Application" to manage the initialization, communication, and interaction with hardware components.

3.4.1 Shell Command

The Shell Command Interface enables user interaction with the AAT-QDMA system through predefined commands. These commands follow users to retrieve the status of the hardware objects, reinitialize system components, and start the DataMover for managing OrderBook output to the Host Pricing Engine.

The shell commands are implemented in "xlnx_shell_aat.cpp", and the available commands are listed in Table 16.

Table 16 AAT-QDMA Application Shell Command

Command	Argument	Description
getstatus	None	Retrieve the status of individual hardware objects
reinit	None	Reinitialize all AAT-QDMA objects
startdatamover	None	Start the DataMover submodule and enable OrderBook output

3.4.2 Application

The AAT-QDMA Application initializes and manages hardware components while binding command tables to the shell for user interaction. The main functionality is split into two phases: Initialization and Idle State.

Initialization

The initialization phase sets up the system, configures hardware and software components, and prepares the application for user commands.

1) The application initializes the DPDK Environment Abstraction Layer (EAL) using "rte_eal_init" to prepare the QDMA platform. It then verifies the availability of Ethernet devices using "rte_eth_dev_count_avail" and exits if no devices are found, ensuring readiness for QDMA operations.
2) Input and output streams are configured for console interaction, with settings adapted for compatibility with Linux or Windows environments.
3) The main application object ("g_aat") is initialized and associated with the hardware device interface to establish communication between hardware and application logic. During this step, the software assigns the base address of each hardware kernel to their corresponding software drivers.
4) Command table for various hardware components are bound to the shell console using "AddObjectCommandTable", ensuring that all hardware objects and their associated commands are accessible through the shell interface.

Idle

After initialization, the application enters the Idle state, where the shell is ready to accept user commands. The shell provides two operational modes for user interaction: One-Shot Mode and Interactive Mode.

- One-Shot Mode: In this mode, users execute a single command directly from the command line. This is ideal for automated scripts or scenarios where a specific operation needs to be performed without entering an interactive session. For example:

  ```
  >> ./aat_qdma_exe aat getstatus
  ```

- Interactive Mode: This mode opens a console session where users can input multiple commands continuously. It is particularly useful for debugging, testing, or managing the system in real-time. The session remains active until the user exits explicitly. For example:

  ```
  >> ./aat_qdma_exe
  ```

  In the interactive mode, users can type commands in the format: <object> <command> [arguments].

  For example: Retrieve the status of the OrderBook:

  ```
  >> orderbook getstatus
  ```

Users can view available commands for an object by typing:

```
>> <object> help
```

3.5 AAT-QDMA Script

The AAT-QDMA script consists of shell commands designed to configure all hardware and software components required for the system's operation. These scripts ensure that each component is properly initialized and configured. The configuration is divided into two files:

- **"**demo_setup.cfg**"**: Configures the system for "Pricing Engine on Card" mode.
- **"**demo_setup_with_datamover.cfg**"**: Configures the system for "Pricing Engine on Software" mode, including DataMover setup and excluding hardware Pricing Engine configurations.

3.5.1 demo_setup.cfg

The "demo_setup.cfg" file contains commands to initialize and configure the system components, including Ethernet, UDP/TCP IP, Line Handler, Feed Handler, Pricing Engine, OrderEntry, and ClockTick Generator. Below is a summary of the configurations applied:

<u>UDP IP</u>

The UDP IP configuration sets up the network parameters and communication settings required for processing incoming data. The example commands below demonstrate how to configure the UDP IP with detailed explanations for each step:

1) Assigns a unique IP address to the UDP interface for identifying the device on the network.

   ```
   >> udpip0 setipaddr 192.168.10.200
   ```

2) Configures the gateway address, which acts as the default route for network packets destined for external networks.

   ```
   >> udpip0 setgateway 192.168.10.100
   ```

3) Add ports for listening to incoming UDP traffic. These ports enable the device to receive data on specified port numbers, such as 14318 and 15318.

   ```
   >> udpip0 addport 14318
   >> udpip0 addport 15318
   ```

4) Activates the IGMP for managing multicast group memberships. This is required for multicast communication.

   ```
   >> udpip0 setigmp true
   ```

5) Configure the IGMP version to 3, which supports advanced multicast features.

   ```
   >> udpip0 setigmpver 3
   ```

6) Add multicast IP addresses to the IGMP table, enabling the device to join specific multicast groups for receiving group-specific data. In this example, the multicast addresses 224.0.31.9 and 224.0.32.9 are added.

   ```
   >> udpip0 addmcast 224.0.31.9
   >> udpip0 addmcast 224.0.32.9
   ```

TCP IP

The TCP IP configuration sets up the network parameters necessary for establishing and managing TCP connections. The example commands below detail how to configure the TCP IP and explain the purpose of each step:

1) Configures the IP address for the TCP interface. This address uniquely identifies the device on the network for TCP communication.

   ```
   >> tcpip setipaddr 192.168.20.200
   ```

2) Assigns the gateway address, which acts as the default route for packets destinated for external networks.

   ```
   >> tcpip setgateway 192.168.20.100
   ```

Line Handler

The Line Handler configuration manages filters and timing parameters for processing UDP multicast feeds. It is responsible for handling A and B lines, setting up port filters, and configuring sequence timers. The example commands below demonstrate how to configure the Line Handler with explanations for each step:

1) Adds filter for specific input ports, IP addresses, and ports to process multicast feeds.
   - The first command adds a filter for line A, processing data from IP address 205.209.221.75 on port 14318 with splitID set to 0.
   - The second command adds a filter for line B, processing data from IP address 205.209.212.75 on port 15318, also with splitID set to 0.

   ```
   >> linehandler add 0 205.209.221.75 14318 0
   >> linehandler add 0 205.209.212.75 15318 0
   ```

2) Configures the sequence timer to 1000 microseconds.

   ```
   >> linehandler setsequencetimer 1000
   ```

Feed Handler

The Feed Handler configuration manages the registration of Security IDs, enabling the processing of market data updates for specified securities. This setup ensures that the Feed Handler processes only the relevant data streams. Below are the commands to add Security IDs 1024, 2048, 3072, and others, including a large ID 305419896. Each registered Security ID corresponds to a specific ticker symbol or financial instrument in the market.

```
>> feedhandler add 1024
>> feedhandler add 2048
>> feedhandler add 3072
>> feedhandler add 4096
>> feedhandler add 5120
>> feedhandler add 6144
>> feedhandler add 7168
>> feedhandler add 8192
>> feedhandler add 9216
>> feedhandler add 10240
>> feedhandler add 305419896
```

Pricing Engine

The Pricing Engine configuration sets up global trading strategies and enables global pricing mode for all securities. These settings ensure that trading logic is applied across all securities in the system. Below are the commands and their explanations:

1) Configures the global trading strategy to “peg” using the “setglobalstrategy” command. The “peg strategy” dynamically adjusts bid and ask prices based on changes in the top-of-book (ToB) price.

```
>> pricingengine setglobalstrategy peg
```

2) Activate the global pricing mode using the “setglobalmode” command with the argument “true”. When enabled, global pricing mode applies the configured strategy to all securities.

```
>> pricingengine setglobalmode true
```

OrderEntry

The OrderEntry configuration establishes the connection to a remote system and optimizes data transmission by enabling hardware-based checksum generation. Below are the commands and their explanations:

1) Configures the OrderEntry to enable partial checksum generations using the “setcsumgen” command with the argument “true”. When enabled, the hardware offloads checksum calculations for TCP packets, reducing CPU overhead and improving transmission efficiency.

```
>> orderentry setcsumgen true
```

2) Sets up a connection to a remote system using the “connect” command with the specified IP address and port. In this example, the system connects to 192.168.20.100 on port 12345.

```
>> orderentry connect 192.168.20.100 12345
```

ClockTick Generator

The ClockTick Generator configuration sets precise timing intervals for each event stream and enables the streams to synchronize hardware modules in the system. Below are the commands and their explanations:

1) Configures the interval for each clock tick event stream using the “setinterval” command. In this example, stream 0 is configured with an interval of 1,000,000 microseconds (1 second).

```
>> clocktickgen setinterval 0 1000000
```

2) Activate a specific tick stream using the “setenable” command with the argument “true”. In this example, stream 0 is enabled to start generating tick events.

```
>> clocktickgen setenable 0 true
```

The generator supports up to 5 event streams, which can be assigned to specific hardware modules as follows: Stream 0-Feed Handler, Stream 1-OrderBook, Stream 2-Pricing Engine, Stream 3-OrderEntry, and Stream 4-Line Handler.

3.5.2 demo_setup_with_datamover.cfg

The “demo_setup_with_datamover.cfg” file is largely similar to “demo_setup.cfg”, with the following key differences:

- Enables the transmission of OrderBook data to the software Pricing Engine.
- The hardware Pricing Engine is not utilized in “Pricing Engine on Software” mode.

<u>DataMover</u>

The DataMover configuration enables the system to transfer OrderBook data to the software Pricing Engine. Below are the commands and their explanations:

1) The “startdatamover” command initializes the DataMover submodule, allowing the OrderBook to transmit data to the host for further processing.

   ```
   >> aat startdatamover
   ```

2) The “threadstart” command activates the thread responsible for running the software Pricing Engine, which processes data received from the OrderBook.

   ```
   >> datamover threadstart
   ```

4 Target Software

The software on the target system uses “tcpreplay” as a market data generator. “tcpreplay” is an open-source network traffic generator tool that allows users to replay previously captured network traffic from “pcap” (packet capture) files. It is commonly used for testing and debugging network devices, applications, and security systems by simulating real-world traffic conditions.

In this design, “tcpreplay” replays the “cme_input_arb.pcap” file to simulate CME market data, enabling the AAT-QDMA system to process and respond to market conditions.

For more information about “tcpreplay”, visit the official website:

<u>https://tcpreplay.appneta.com/</u>

5 Revision History

Revision	Date (D-M-Y)	Description
1.00	21-Jan-25	Initial version release