Memory IO Processor (RAPIDS) Specification

Generated by md\_to\_docx.py

today

Abstract

Table of Contents

# Memory IO Processor (RAPIDS) Specification

## Chapter 1: Overview

## Chapter 2: Blocks

### RAPIDS System Overview

The Memory Input/Output Processor (RAPIDS) is a high-performance data streaming engine operating at 2.5 GHz, designed to provide reliable, low-latency data transfer between external DRAM and processing cores through the Network network. The architecture features dual independent data paths with sophisticated multi-channel support for up to 32 concurrent streams, implementing pure pipeline architectures that eliminate traditional FSM overhead to achieve sustained throughput exceeding 1.2 Tbps per path.

|  |
| --- |
| RAPIDS/Network/MSAP |

RAPIDS/Network/MSAP

#### Scheduler Group

The Scheduler Group provides complete integrated channel processing through an innovative dual-FSM architecture that combines descriptor execution with parallel address alignment optimization. The main scheduler FSM manages descriptor lifecycle, credit operations, and program sequencing, while the address alignment FSM runs concurrently to pre-calculate optimal transfer parameters including burst lengths and chunk enable patterns. This approach eliminates alignment calculation overhead from critical AXI timing paths, enabling sustained high-performance operation while maintaining comprehensive control over stream boundaries and error handling.

#### Descriptor Engine

The Descriptor Engine implements a sophisticated six-state finite state machine that orchestrates descriptor fetching and processing operations with dual-path support for both APB programming interface requests and RDA packet interface operations. The FSM manages the complete descriptor lifecycle from initial request validation through AXI read transactions, descriptor parsing, and final output generation. RDA packets receive priority processing over APB requests, ensuring optimal network responsiveness, while comprehensive stream boundary support includes complete EOS/EOL/EOD field extraction and propagation through a 4-deep descriptor FIFO.

#### Program Engine

The Program Engine implements a streamlined four-state finite state machine that manages post-processing write operations for each virtual channel after descriptor completion. The FSM handles program address writes with configurable data values to support notification, control, and cleanup operations with comprehensive timeout mechanisms and robust error handling. Conditional programming through graceful null address handling enables descriptor-driven execution, while shared AXI interface operations use ID-based response routing for proper multi-channel coordination and runtime reconfiguration support.

#### Source Data Path Group

The Source Data Path implements a complete data transmission pipeline from scheduler requests through Network packet delivery, featuring a revolutionary pure pipeline architecture with zero-FSM overhead for optimal performance. The path integrates a sophisticated AXI read engine with multi-channel arbitration, preallocation-based deadlock prevention, and chunk-aware burst optimization to achieve maximum memory bandwidth utilization. Integrated SRAM control provides multi-channel buffering with stream-aware flow control, while the Network master implements a four-stage pipeline with credit-based flow control and mathematically proven zero packet loss guarantees.

#### Sink Data Path Group

The Sink Data Path provides comprehensive data reception and processing from Network packet arrival through final AXI memory writes, implementing sophisticated buffer management and multi-channel arbitration for optimal throughput. The Network slave features bulletproof ACK generation with dual FIFO queues, comprehensive validation, and intelligent packet routing based on packet classification. Sink SRAM control manages multi-channel buffering with sophisticated flow control and EOS completion signaling, while the AXI write engine implements pure pipeline architecture with zero-cycle arbitration, transfer strategy optimization, and chunk-aware write strobes for precise memory utilization.

#### Monitor Bus AXI4-Lite Group

The Monitor Bus AXI4-Lite Group provides unified system monitoring and event aggregation across all RAPIDS subsystems, implementing a comprehensive filtering and routing architecture for optimal system visibility. The group aggregates monitor streams from source and sink data paths through round-robin arbitration, applying configurable protocol-specific filtering for AXI, Network, and CORE events. The architecture supports both interrupt generation through error/interrupt FIFOs for critical events and external logging through configurable master write operations, providing complete event coverage across 544 defined event codes spanning all protocols and packet types for comprehensive system debugging and performance analysis. ### Scheduler Group

#### Scheduler Group

##### Overview

The Scheduler Group provides a complete integrated channel processing unit that combines the Scheduler, Descriptor Engine, and Monitor Bus Aggregator into a cohesive module. This wrapper simplifies system integration by providing a unified interface for complete channel processing functionality.

**NOTE:** The program engine has been **removed** from this module and replaced by separate ctrlrd\_engine and ctrlwr\_engine interfaces. These control engines are instantiated externally and connected through the ctrlrd\_\* and ctrlwr\_\* ports. For backwards compatibility, the legacy prog\_\* AXI ports are retained but tied off.

The wrapper implements coordinated reset handling across all components, enhanced data interface with alignment bus support using RAPIDS package types, and unified monitor bus aggregation for comprehensive system visibility.

|  |
| --- |
| scheduler\_group |

scheduler\_group

##### Key Features

* **Integrated Channel Processing**: Complete descriptor processing and scheduling functionality
* **Control Engine Interfaces**: Separate ctrlrd (control read) and ctrlwr (control write) interfaces for external control operations
* **Descriptor AXI Interface**: Single descriptor read interface (512-bit data width) for fetching descriptors
* **Legacy Compatibility**: Retained prog\_\* AXI ports (tied off) for backwards compatibility
* **Address Alignment Bus**: Pre-calculated alignment information using RAPIDS package types for optimal AXI performance
* **Channel Reset Coordination**: Graceful reset handling with completion signaling
* **Monitor Bus Aggregation**: Unified monitor output from descriptor engine, scheduler, and (tied-off) program engine
* **RAPIDS Package Types**: Uses standardized alignment\_info\_t and transfer\_phase\_t types for enhanced data interface

##### Interface Specification

###### Clock and Reset

| Signal Name | Type | Width | Direction | Required | Description |
| --- | --- | --- | --- | --- | --- |
| **clk** | logic | 1 | Input | Yes | System clock |
| **rst\_n** | logic | 1 | Input | Yes | Active-low asynchronous reset |

###### APB Programming Interface

| Signal Name | Type | Width | Direction | Required | Description |
| --- | --- | --- | --- | --- | --- |
| **apb\_valid** | logic | 1 | Input | Yes | APB descriptor fetch request valid |
| **apb\_ready** | logic | 1 | Output | Yes | APB descriptor fetch request ready |
| **apb\_addr** | logic | ADDR\_WIDTH | Input | Yes | APB descriptor address |

###### RDA Packet Interface

| Signal Name | Type | Width | Direction | Required | Description |
| --- | --- | --- | --- | --- | --- |
| **rda\_valid** | logic | 1 | Input | Yes | RDA packet valid |
| **rda\_ready** | logic | 1 | Output | Yes | RDA packet ready |
| **rda\_packet** | logic | DATA\_WIDTH | Input | Yes | RDA packet data |
| **rda\_channel** | logic | CHAN\_WIDTH | Input | Yes | RDA target channel |

###### EOS Completion Interface

| Signal Name | Type | Width | Direction | Required | Description |
| --- | --- | --- | --- | --- | --- |
| **eos\_completion\_valid** | logic | 1 | Input | Yes | EOS completion notification valid |
| **eos\_completion\_ready** | logic | 1 | Output | Yes | EOS completion notification ready |
| **eos\_completion\_channel** | logic | CHAN\_WIDTH | Input | Yes | Channel with EOS completion |

###### Configuration Interface

| Signal Name | Type | Width | Direction | Required | Description |
| --- | --- | --- | --- | --- | --- |
| **cfg\_idle\_mode** | logic | 1 | Input | Yes | Global idle mode control |
| **cfg\_channel\_wait** | logic | 1 | Input | Yes | Channel wait control |
| **cfg\_channel\_enable** | logic | 1 | Input | Yes | Channel enable control |
| **cfg\_use\_credit** | logic | 1 | Input | Yes | Credit mode enable |
| **cfg\_initial\_credit** | logic | 4 | Input | Yes | Initial credit values (exponential encoding: 0->1, 1->2, 2->4, …, 14->16384, 15->infinity) |
| **credit\_increment** | logic | 1 | Input | Yes | Credit increment request |
| **cfg\_prefetch\_enable** | logic | 1 | Input | Yes | Descriptor prefetch enable |
| **cfg\_fifo\_threshold** | logic | 4 | Input | Yes | Descriptor FIFO threshold |
| **cfg\_addr0\_base** | logic | ADDR\_WIDTH | Input | Yes | Address range 0 base |
| **cfg\_addr0\_limit** | logic | ADDR\_WIDTH | Input | Yes | Address range 0 limit |
| **cfg\_addr1\_base** | logic | ADDR\_WIDTH | Input | Yes | Address range 1 base |
| **cfg\_addr1\_limit** | logic | ADDR\_WIDTH | Input | Yes | Address range 1 limit |
| **cfg\_channel\_reset** | logic | 1 | Input | Yes | Dynamic channel reset request |
| **cfg\_ctrlrd\_max\_try** | logic | 8 | Input | Yes | Control read max retry count |

###### Status Interface

| Signal Name | Type | Width | Direction | Required | Description |
| --- | --- | --- | --- | --- | --- |
| **descriptor\_engine\_idle** | logic | 1 | Output | Yes | Descriptor engine idle status |
| **program\_engine\_idle** | logic | 1 | Output | Yes | Program engine idle status (tied to 1’b1 - removed) |
| **scheduler\_idle** | logic | 1 | Output | Yes | Scheduler idle status |
| **descriptor\_credit\_counter** | logic | 32 | Output | Yes | Current credit counter value |
| **fsm\_state** | logic | 8 | Output | Yes | Current scheduler FSM state |
| **scheduler\_error** | logic | 1 | Output | Yes | Scheduler error flag |
| **backpressure\_warning** | logic | 1 | Output | Yes | Backpressure warning flag |

###### Data Engine Interface

| Signal Name | Type | Width | Direction | Required | Description |
| --- | --- | --- | --- | --- | --- |
| **data\_valid** | logic | 1 | Output | Yes | Data transfer request active |
| **data\_ready** | logic | 1 | Input | Yes | Data engine ready for transfer |
| **data\_address** | logic | ADDR\_WIDTH | Output | Yes | Current data address |
| **data\_length** | logic | 32 | Output | Yes | Remaining data length |
| **data\_type** | logic | 2 | Output | Yes | Packet type from descriptor |
| **data\_eos** | logic | 1 | Output | Yes | End of Stream indicator |
| **data\_transfer\_length** | logic | 32 | Input | Yes | Actual transfer length completed |
| **data\_error** | logic | 1 | Input | Yes | Data transfer error |
| **data\_done\_strobe** | logic | 1 | Input | Yes | Data transfer completed |

###### Address Alignment Bus Interface

| Signal Name | Type | Width | Direction | Required | Description |
| --- | --- | --- | --- | --- | --- |
| **data\_alignment\_info** | alignment\_info\_t | struct | Output | Yes | Pre-calculated alignment information (RAPIDS package type) |
| **data\_alignment\_valid** | logic | 1 | Output | Yes | Alignment information valid |
| **data\_alignment\_ready** | logic | 1 | Input | Yes | Alignment information ready |
| **data\_alignment\_next** | logic | 1 | Input | Yes | Request next alignment calculation |
| **data\_transfer\_phase** | transfer\_phase\_t | enum | Output | Yes | Current transfer phase (RAPIDS package type) |
| **data\_sequence\_complete** | logic | 1 | Output | Yes | Transfer sequence complete |

###### Control Read Engine Interface

| Signal Name | Type | Width | Direction | Required | Description |
| --- | --- | --- | --- | --- | --- |
| **ctrlrd\_valid** | logic | 1 | Output | Yes | Control read request valid |
| **ctrlrd\_ready** | logic | 1 | Input | Yes | Control read request ready |
| **ctrlrd\_addr** | logic | ADDR\_WIDTH | Output | Yes | Control read address |
| **ctrlrd\_data** | logic | 32 | Output | Yes | Control read expected data |
| **ctrlrd\_mask** | logic | 32 | Output | Yes | Control read data mask |
| **ctrlrd\_error** | logic | 1 | Input | Yes | Control read error (mismatch or AXI error) |
| **ctrlrd\_result** | logic | 32 | Input | Yes | Control read actual data result |

###### Control Write Engine Interface

| Signal Name | Type | Width | Direction | Required | Description |
| --- | --- | --- | --- | --- | --- |
| **ctrlwr\_valid** | logic | 1 | Output | Yes | Control write request valid |
| **ctrlwr\_ready** | logic | 1 | Input | Yes | Control write request ready |
| **ctrlwr\_addr** | logic | ADDR\_WIDTH | Output | Yes | Control write address |
| **ctrlwr\_data** | logic | 32 | Output | Yes | Control write data |
| **ctrlwr\_error** | logic | 1 | Input | Yes | Control write error (AXI error) |

###### RDA Credit Return Interface

| Signal Name | Type | Width | Direction | Required | Description |
| --- | --- | --- | --- | --- | --- |
| **rda\_complete\_valid** | logic | 1 | Output | Yes | RDA completion notification |
| **rda\_complete\_ready** | logic | 1 | Input | Yes | RDA completion ready |
| **rda\_complete\_channel** | logic | CHAN\_WIDTH | Output | Yes | Channel with RDA completion |

###### Descriptor Engine AXI4 Master Read Interface

**NOTE:** This interface connects internally to the descriptor engine. External connectivity is provided by scheduler\_group\_array which arbitrates multiple channels.

| Signal Name | Type | Width | Direction | Required | Description |
| --- | --- | --- | --- | --- | --- |
| **desc\_ar\_valid** | logic | 1 | Output | Yes | Read address valid |
| **desc\_ar\_ready** | logic | 1 | Input | Yes | Read address ready |
| **desc\_ar\_addr** | logic | ADDR\_WIDTH | Output | Yes | Read address |
| **desc\_ar\_len** | logic | 8 | Output | Yes | Burst length |
| **desc\_ar\_size** | logic | 3 | Output | Yes | Burst size |
| **desc\_ar\_burst** | logic | 2 | Output | Yes | Burst type |
| **desc\_ar\_id** | logic | AXI\_ID\_WIDTH | Output | Yes | Read ID (internal transaction tracking) |
| **desc\_ar\_lock** | logic | 1 | Output | Yes | Lock type |
| **desc\_ar\_cache** | logic | 4 | Output | Yes | Cache type |
| **desc\_ar\_prot** | logic | 3 | Output | Yes | Protection type |
| **desc\_ar\_qos** | logic | 4 | Output | Yes | QoS identifier |
| **desc\_ar\_region** | logic | 4 | Output | Yes | Region identifier |
| **desc\_r\_valid** | logic | 1 | Input | Yes | Read data valid |
| **desc\_r\_ready** | logic | 1 | Output | Yes | Read data ready |
| **desc\_r\_data** | logic | DATA\_WIDTH | Input | Yes | Read data |
| **desc\_r\_resp** | logic | 2 | Input | Yes | Read response |
| **desc\_r\_last** | logic | 1 | Input | Yes | Read last |
| **desc\_r\_id** | logic | AXI\_ID\_WIDTH | Input | Yes | Read ID |

###### Program Engine AXI4 Master Write Interface (Legacy - Tied Off)

**NOTE:** The program engine has been **removed** from scheduler\_group. These ports are retained for backwards compatibility but are **tied off** (all outputs driven to 0, ready signals driven to 1). Control operations are now handled through ctrlrd\_\* and ctrlwr\_\* interfaces.

| Signal Name | Type | Width | Direction | Status | Description |
| --- | --- | --- | --- | --- | --- |
| **prog\_aw\_valid** | logic | 1 | Output | Tied to 0 | Write address valid (inactive) |
| **prog\_aw\_ready** | logic | 1 | Input | Ignored | Write address ready |
| **prog\_aw\_addr** | logic | ADDR\_WIDTH | Output | Tied to 0 | Write address |
| **prog\_aw\_len** | logic | 8 | Output | Tied to 0 | Burst length |
| **prog\_aw\_size** | logic | 3 | Output | Tied to 0 | Burst size |
| **prog\_aw\_burst** | logic | 2 | Output | Tied to 0 | Burst type |
| **prog\_aw\_id** | logic | AXI\_ID\_WIDTH | Output | Tied to 0 | Write ID |
| **prog\_aw\_lock** | logic | 1 | Output | Tied to 0 | Lock type |
| **prog\_aw\_cache** | logic | 4 | Output | Tied to 0 | Cache type |
| **prog\_aw\_prot** | logic | 3 | Output | Tied to 0 | Protection type |
| **prog\_aw\_qos** | logic | 4 | Output | Tied to 0 | QoS identifier |
| **prog\_aw\_region** | logic | 4 | Output | Tied to 0 | Region identifier |
| **prog\_w\_valid** | logic | 1 | Output | Tied to 0 | Write data valid (inactive) |
| **prog\_w\_ready** | logic | 1 | Input | Ignored | Write data ready |
| **prog\_w\_data** | logic | 32 | Output | Tied to 0 | Write data |
| **prog\_w\_strb** | logic | 4 | Output | Tied to 0 | Write strobes |
| **prog\_w\_last** | logic | 1 | Output | Tied to 0 | Write last |
| **prog\_b\_valid** | logic | 1 | Input | Ignored | Write response valid |
| **prog\_b\_ready** | logic | 1 | Output | Tied to 1 | Write response ready (always ready) |
| **prog\_b\_resp** | logic | 2 | Input | Ignored | Write response |
| **prog\_b\_id** | logic | AXI\_ID\_WIDTH | Input | Ignored | Write ID |

###### Monitor Bus Interface

| Signal Name | Type | Width | Direction | Required | Description |
| --- | --- | --- | --- | --- | --- |
| **mon\_valid** | logic | 1 | Output | Yes | Monitor packet valid |
| **mon\_ready** | logic | 1 | Input | Yes | Monitor ready to accept packet |
| **mon\_packet** | logic | 64 | Output | Yes | Monitor packet data |

##### Architecture

###### Internal Components

* **Descriptor Engine**: Handles APB and RDA descriptor processing with AXI read operations
* **Scheduler**: Main FSM managing descriptor execution, credit management, and control engine sequencing
* **Monitor Bus Aggregator**: Round-robin aggregation of monitor events from descriptor engine and scheduler
* **Control Engine Interfaces**: Exposed ctrlrd and ctrlwr ports for external control read/write engines

**NOTE:** The program engine has been **removed** and replaced by external control engines: - Control operations are now handled by external ctrlrd\_engine and ctrlwr\_engine modules - These connect through simplified ctrlrd\_\* and ctrlwr\_\* interfaces (not full AXI) - Legacy prog\_\* AXI ports retained but tied off for backwards compatibility

###### Address Alignment Processing

The scheduler includes a dedicated Address Alignment FSM that runs in parallel with the main scheduler during the SCHED\_DESCRIPTOR\_ACTIVE state. This FSM pre-calculates all alignment information and provides it to data engines via the alignment bus interface, eliminating alignment calculation overhead from the critical AXI timing paths.

###### Channel Reset Coordination

All components support coordinated channel reset through the cfg\_channel\_reset input. Each component handles reset gracefully: - **Descriptor Engine**: Completes AXI read transactions in WAIT\_READ state before reset - **Scheduler**: Completes control operations (ctrlrd/ctrlwr) before reset

Reset completion is indicated through the idle status outputs (descriptor\_engine\_idle, scheduler\_idle). The program\_engine\_idle output is tied to 1’b1 (always idle) since the program engine has been removed.

###### Control Operation Sequencing

The scheduler coordinates control read and write operations through the external control engines: 1. Descriptor processing completes 2. If ctrlrd needed: Issue control read request and wait for result 3. If ctrlwr needed: Issue control write request and wait for completion 4. Activate data path engines 5. Return to idle state

Control engines are instantiated externally (typically in scheduler\_group\_array) and arbitrated across multiple channels.

##### Network 2.0 Support

The scheduler group supports the Network 2.0 protocol specification for RDA packet processing, which uses chunk enables instead of the older start/len approach for indicating valid data chunks within each 512-bit packet.

##### Usage Guidelines

###### Channel Reset Sequence

// Example channel reset coordination  
logic all\_engines\_idle = descriptor\_engine\_idle & program\_engine\_idle & scheduler\_idle;  
  
// Assert reset when needed  
if (reset\_request && !cfg\_channel\_reset) begin  
 cfg\_channel\_reset <= 1'b1;  
end   
// Deassert reset when all engines are idle  
else if (cfg\_channel\_reset && all\_engines\_idle) begin  
 cfg\_channel\_reset <= 1'b0;  
end

###### Performance Optimization

* Use separate AXI interconnect paths for descriptor read and program write
* Configure alignment bus ready signals appropriately for optimal throughput
* Monitor FSM states and idle signals for debugging and performance analysis
* Utilize pre-calculated alignment information for efficient AXI transfers

###### Monitor Bus Configuration

The wrapper provides unified monitor output with agent ID-based filtering: - Descriptor Engine events: Use agent ID DESC\_MON\_AGENT\_ID - Program Engine events: Use agent ID PROG\_MON\_AGENT\_ID - Scheduler events: Use agent ID SCHED\_MON\_AGENT\_ID #### Scheduler

##### Overview

The Scheduler orchestrates data movement operations by managing descriptor execution, coordinating with data engines, and controlling program sequences. The module implements two sophisticated state machines: a main scheduler FSM for descriptor execution and an address alignment FSM that runs in parallel to pre-calculate optimal transfer parameters for maximum AXI performance.

|  |
| --- |
| scheduler |

scheduler

##### Key Features

* **Dual State Machine Architecture**: Main scheduler FSM plus parallel address alignment FSM
* **Descriptor-Driven Operation**: Processes descriptors from descriptor engine with full stream control
* **Address Alignment Pre-calculation**: Dedicated FSM provides alignment information before AXI operations
* **Credit Management**: Atomic credit operations with early warning thresholds
* **Stream Boundary Support**: Complete EOS processing with sequence completion tracking
* **Program Sequencing**: Sequential program engine coordination for post-processing
* **Channel Reset Support**: Graceful channel shutdown with proper completion signaling
* **Monitor Integration**: Comprehensive event reporting for system visibility

##### Interface Specification

###### Configuration Parameters

| Parameter | Default Value | Description |
| --- | --- | --- |
| CHANNEL\_ID | 0 | Static channel identifier for this scheduler instance |
| NUM\_CHANNELS | 32 | Total number of channels in system |
| CHAN\_WIDTH | $clog2(NUM\_CHANNELS) | Width of channel address fields |
| ADDR\_WIDTH | 64 | Address width for data operations |
| DATA\_WIDTH | 512 | Descriptor packet width |
| CREDIT\_WIDTH | 8 | Credit counter width |
| TIMEOUT\_CYCLES | 1000 | Data transfer timeout threshold |
| EARLY\_WARNING\_THRESHOLD | 4 | Credit warning threshold |

###### Clock and Reset Signals

| Signal Name | Type | Width | Direction | Required | Description |
| --- | --- | --- | --- | --- | --- |
| **clk** | logic | 1 | Input | Yes | System clock |
| **rst\_n** | logic | 1 | Input | Yes | Active-low asynchronous reset |

###### Configuration Interface

| Signal Name | Type | Width | Direction | Required | Description |
| --- | --- | --- | --- | --- | --- |
| **cfg\_idle\_mode** | logic | 1 | Input | Yes | Force scheduler to idle state |
| **cfg\_channel\_wait** | logic | 1 | Input | Yes | Wait for channel enable |
| **cfg\_channel\_enable** | logic | 1 | Input | Yes | Channel enable control |
| **cfg\_use\_credit** | logic | 1 | Input | Yes | Enable credit-based flow control |
| **cfg\_initial\_credit** | logic | CREDIT\_WIDTH | Input | Yes | Initial credit value (exponential encoding) |
| **credit\_increment** | logic | 1 | Input | Yes | Credit increment request |
| **cfg\_channel\_reset** | logic | 1 | Input | Yes | Dynamic channel reset request |

###### Status Interface

| Signal Name | Type | Width | Direction | Required | Description |
| --- | --- | --- | --- | --- | --- |
| **scheduler\_idle** | logic | 1 | Output | Yes | Scheduler idle status indicator |
| **fsm\_state** | scheduler\_state\_t | 3 | Output | Yes | Current main FSM state |
| **descriptor\_credit\_counter** | logic | CREDIT\_WIDTH | Output | Yes | Current credit count |
| **scheduler\_error** | logic | 1 | Output | Yes | Scheduler error status |
| **backpressure\_warning** | logic | 1 | Output | Yes | Credit or timeout warning |

###### Descriptor Engine Interface

| Signal Name | Type | Width | Direction | Required | Description |
| --- | --- | --- | --- | --- | --- |
| **descriptor\_valid** | logic | 1 | Input | Yes | Descriptor available from engine |
| **descriptor\_ready** | logic | 1 | Output | Yes | Ready to accept descriptor |
| **descriptor\_packet** | logic | DATA\_WIDTH | Input | Yes | Descriptor packet data |
| **descriptor\_same** | logic | 1 | Input | Yes | Same descriptor flag |
| **descriptor\_error** | logic | 1 | Input | Yes | Descriptor error status |
| **descriptor\_is\_rda** | logic | 1 | Input | Yes | RDA packet indicator |
| **descriptor\_rda\_channel** | logic | CHAN\_WIDTH | Input | Yes | RDA channel identifier |
| **descriptor\_eos** | logic | 1 | Input | Yes | End of Stream from descriptor |
| **descriptor\_type** | logic | 2 | Input | Yes | Packet type |

###### Data Engine Interface

| Signal Name | Type | Width | Direction | Required | Description |
| --- | --- | --- | --- | --- | --- |
| **data\_valid** | logic | 1 | Output | Yes | Data transfer request active |
| **data\_ready** | logic | 1 | Input | Yes | Data engine ready for transfer |
| **data\_address** | logic | ADDR\_WIDTH | Output | Yes | Current data address |
| **data\_length** | logic | 32 | Output | Yes | Remaining data length |
| **data\_type** | logic | 2 | Output | Yes | Packet type from descriptor |
| **data\_eos** | logic | 1 | Output | Yes | End of Stream indicator |
| **data\_transfer\_length** | logic | 32 | Input | Yes | Actual transfer length completed |
| **data\_error** | logic | 1 | Input | Yes | Data transfer error |
| **data\_done\_strobe** | logic | 1 | Input | Yes | Data transfer completed |

###### Address Alignment Bus Interface

| Signal Name | Type | Width | Direction | Required | Description |
| --- | --- | --- | --- | --- | --- |
| **data\_alignment\_info** | alignment\_info\_t | Variable | Output | Yes | Pre-calculated alignment information |
| **data\_alignment\_valid** | logic | 1 | Output | Yes | Alignment information valid |
| **data\_alignment\_ready** | logic | 1 | Input | Yes | Alignment information ready |
| **data\_alignment\_next** | logic | 1 | Input | Yes | Request next alignment calculation |
| **data\_transfer\_phase** | transfer\_phase\_t | 3 | Output | Yes | Current transfer phase |
| **data\_sequence\_complete** | logic | 1 | Input | Yes | Transfer sequence complete |

###### Ctrlrd Engine Interface (Control Read Operations)

| Signal Name | Type | Width | Direction | Required | Description |
| --- | --- | --- | --- | --- | --- |
| **ctrlrd\_valid** | logic | 1 | Output | Yes | Control read operation request |
| **ctrlrd\_ready** | logic | 1 | Input | Yes | Control read match/completion (asserts when read data matches expected value) |
| **ctrlrd\_addr** | logic | ADDR\_WIDTH | Output | Yes | Control read address |
| **ctrlrd\_data** | logic | 32 | Output | Yes | Control read expected/comparison data |
| **ctrlrd\_mask** | logic | 32 | Output | Yes | Control read mask/flags |
| **ctrlrd\_error** | logic | 1 | Input | Yes | Control read operation error (valid when ctrlrd\_valid && ctrlrd\_ready) |
| **ctrlrd\_result** | logic | 32 | Input | Yes | Control read result data (actual value read) |

**Ctrlrd Ready Behavior:** - ctrlrd\_ready asserts **only when** the read response matches the expected value (ctrlrd\_data) - The ctrlrd engine performs retry operations until match or max retries exceeded - ctrlrd\_error is valid when both ctrlrd\_valid and ctrlrd\_ready are asserted - Error conditions: AXI response error OR max retry count exceeded without match

**Operation Flow:** 1. Scheduler asserts ctrlrd\_valid with address, expected data, and mask 2. Ctrlrd engine performs AXI read and compares result with expected data 3. If match: ctrlrd\_ready asserts (proceed to data operations) 4. If no match: Retry read up to cfg\_ctrlrd\_max\_try times 5. If max retries exceeded: ctrlrd\_ready and ctrlrd\_error assert together (enter error state)

**Note:** Ctrlrd operations execute BEFORE data operations, enabling pre-descriptor flag reads or synchronization checks with configurable retry mechanisms.

###### Ctrlwr Engine Interface (Control Write Operations)

| Signal Name | Type | Width | Direction | Required | Description |
| --- | --- | --- | --- | --- | --- |
| **ctrlwr\_valid** | logic | 1 | Output | Yes | Control write operation request |
| **ctrlwr\_ready** | logic | 1 | Input | Yes | Control write operation complete |
| **ctrlwr\_addr** | logic | ADDR\_WIDTH | Output | Yes | Control write address (ctrlwr0 or ctrlwr1) |
| **ctrlwr\_data** | logic | 32 | Output | Yes | Control write data (ctrlwr0 or ctrlwr1) |
| **ctrlwr\_error** | logic | 1 | Input | Yes | Control write operation error |

**Note:** Ctrlwr operations execute AFTER data operations complete, enabling post-descriptor notifications or cleanup operations.

###### RDA Completion Interface

| Signal Name | Type | Width | Direction | Required | Description |
| --- | --- | --- | --- | --- | --- |
| **rda\_complete\_valid** | logic | 1 | Output | Yes | RDA completion notification |
| **rda\_complete\_ready** | logic | 1 | Input | Yes | RDA completion ready |
| **rda\_complete\_channel** | logic | CHAN\_WIDTH | Output | Yes | RDA completion channel |

###### Monitor Bus Interface

| Signal Name | Type | Width | Direction | Required | Description |
| --- | --- | --- | --- | --- | --- |
| **mon\_valid** | logic | 1 | Output | Yes | Monitor packet valid |
| **mon\_ready** | logic | 1 | Input | Yes | Monitor ready to accept packet |
| **mon\_packet** | logic | 64 | Output | Yes | Monitor packet data |

##### Dual State Machine Architecture

###### Main Scheduler FSM

|  |
| --- |
| Address Alignment FSM |

Address Alignment FSM

**States:** - **SCHED\_IDLE**: Ready for new descriptor, all operations complete - **SCHED\_WAIT\_FOR\_CONTROL**: Wait for channel enable and control signals - **SCHED\_ISSUE\_CTRLRD**: Issue control read operation (pre-descriptor, conditional) - **SCHED\_DESCRIPTOR\_ACTIVE**: Execute data transfer operations (alignment FSM runs in parallel) - **SCHED\_ISSUE\_CTRLWR0**: Issue first control write operation (post-descriptor, conditional) - **SCHED\_ISSUE\_CTRLWR1**: Issue second control write operation (post-descriptor, conditional) - **SCHED\_ERROR**: Handle error conditions with sticky error flags

**Operation Sequence:**

IDLE -> WAIT\_FOR\_CONTROL -> ISSUE\_CTRLRD (if needed) ->  
 DESCRIPTOR\_ACTIVE -> ISSUE\_CTRLWR0 (if needed) -> ISSUE\_CTRLWR1 (if needed) -> IDLE

###### Address Alignment FSM (Parallel Operation)

|  |
| --- |
| Address Alignment FSM |

Address Alignment FSM

**States:** - **ALIGN\_IDLE**: Ready for new alignment calculation - **ANALYZE\_ADDRESS**: Single-cycle address analysis and strategy selection - **CALC\_FIRST\_TRANSFER**: Generate alignment transfer parameters - **CALC\_STREAMING**: Calculate optimal streaming burst parameters - **CALC\_FINAL\_TRANSFER**: Handle final partial transfer calculations - **ALIGNMENT\_COMPLETE**: Provide complete alignment information to engines - **ALIGNMENT\_ERROR**: Handle invalid alignment scenarios

##### Address Alignment System

###### Parallel FSM Operation

The address alignment FSM runs in parallel with the main scheduler during the SCHED\_DESCRIPTOR\_ACTIVE state:

// Alignment FSM trigger condition  
if ((r\_current\_state == SCHED\_DESCRIPTOR\_ACTIVE) && (r\_data\_length > 32'h0)) begin  
 w\_alignment\_next\_state = ANALYZE\_ADDRESS;  
end

###### Alignment Information Structure

typedef struct packed {  
 logic is\_aligned; // Already 64-byte aligned  
 logic [5:0] addr\_offset; // Address offset within boundary  
 logic [31:0] first\_transfer\_bytes; // Bytes in alignment transfer  
 logic [NUM\_CHUNKS-1:0] first\_chunk\_enables; // Chunk pattern for alignment  
 logic [7:0] optimal\_burst\_len; // Optimal burst after alignment  
 logic [31:0] final\_transfer\_bytes; // Bytes in final transfer  
 logic [NUM\_CHUNKS-1:0] final\_chunk\_enables; // Final chunk pattern  
 logic [3:0] total\_transfers; // Pre-calculated total count  
 alignment\_strategy\_t alignment\_strategy; // Strategy selection  
} alignment\_info\_t;

###### Transfer Phase Management

typedef enum logic [2:0] {  
 PHASE\_IDLE = 3'h0, // No active transfer  
 PHASE\_ALIGNMENT = 3'h1, // Initial alignment transfer  
 PHASE\_STREAMING = 3'h2, // Optimal aligned streaming  
 PHASE\_FINAL = 3'h3, // Final partial transfer  
 PHASE\_COMPLETE = 3'h4 // Transfer sequence complete  
} transfer\_phase\_t;

##### Descriptor Processing

###### Descriptor Packet Format (512 bits)

| Bit Range | Field Name | Width | Description |
| --- | --- | --- | --- |
| **511:480** | *Reserved* | 32 bits | Reserved for future use |
| **479:448** | ctrlwr1\_data | 32 bits | Control Write 1 data value |
| **447:384** | ctrlwr1\_addr | 64 bits | Control Write 1 address |
| **383:352** | ctrlwr0\_data | 32 bits | Control Write 0 data value |
| **351:288** | ctrlwr0\_addr | 64 bits | Control Write 0 address |
| **287:256** | ctrlrd\_mask | 32 bits | Control Read mask/flags |
| **255:224** | ctrlrd\_data | 32 bits | Control Read data value |
| **223:160** | ctrlrd\_addr | 64 bits | Control Read address |
| **159:128** | next\_descriptor\_addr[63:32] | 32 bits | Next descriptor pointer (upper) |
| **127:96** | next\_descriptor\_addr[31:0] | 32 bits | Next descriptor pointer (lower) |
| **95:64** | data\_addr[63:32] | 32 bits | Data operation address (upper) |
| **63:32** | data\_addr[31:0] | 32 bits | Data operation address (lower) |
| **31:0** | data\_length | 32 bits | Data transfer length (bytes) |

###### Descriptor Field Details

**1. Data Operation Fields (bits 0-95):** - data\_length [31:0] - Transfer length in **bytes** - data\_addr [95:32] - 64-bit memory address for data operation

**2. Next Descriptor Pointer (bits 96-159):** - next\_descriptor\_addr [159:96] - 64-bit address of next descriptor in linked list

**3. Control Read Fields (bits 160-287):** - ctrlrd\_addr [223:160] - 64-bit address for pre-descriptor control read operation - ctrlrd\_data [255:224] - 32-bit data value for control read (comparison/expected value) - ctrlrd\_mask [287:256] - 32-bit mask/flags for control read operation

**4. Control Write Fields (bits 288-479):** - ctrlwr0\_addr [351:288] - 64-bit address for first post-descriptor control write - ctrlwr0\_data [383:352] - 32-bit data for first control write - ctrlwr1\_addr [447:384] - 64-bit address for second post-descriptor control write - ctrlwr1\_data [479:448] - 32-bit data for second control write

**Operation Sequence:** 1. **Control Read (ctrlrd)**: Executed FIRST, before data operations (if ctrlrd\_addr != 0) 2. **Data Operation**: Memory transfer operation (source/sink data path) 3. **Control Write 0 (ctrlwr0)**: Executed after data completion (if ctrlwr0\_addr != 0) 4. **Control Write 1 (ctrlwr1)**: Executed after ctrlwr0 completion (if ctrlwr1\_addr != 0)

**Null Address Handling:** Setting any control address field to 64’h0 skips that operation.

###### Descriptor Field Extraction

// Descriptor unpacking in SCHED\_IDLE state  
if ((r\_current\_state == SCHED\_IDLE) && descriptor\_valid && descriptor\_ready) begin  
 // Extract descriptor fields from 512-bit packet  
  
 // Data operation fields (bits 0-95)  
 r\_data\_length <= descriptor\_packet[31:0];  
 r\_data\_addr <= {descriptor\_packet[95:64], descriptor\_packet[63:32]};  
  
 // Next descriptor pointer (bits 96-159)  
 r\_next\_descriptor\_addr <= {descriptor\_packet[159:128], descriptor\_packet[127:96]};  
  
 // Control Read fields (bits 160-287)  
 r\_ctrlrd\_addr <= {descriptor\_packet[223:192], descriptor\_packet[191:160]};  
 r\_ctrlrd\_data <= descriptor\_packet[255:224];  
 r\_ctrlrd\_mask <= descriptor\_packet[287:256];  
  
 // Control Write 0 fields (bits 288-383)  
 r\_ctrlwr0\_addr <= {descriptor\_packet[351:320], descriptor\_packet[319:288]};  
 r\_ctrlwr0\_data <= descriptor\_packet[383:352];  
  
 // Control Write 1 fields (bits 384-479)  
 r\_ctrlwr1\_addr <= {descriptor\_packet[447:416], descriptor\_packet[415:384]};  
 r\_ctrlwr1\_data <= descriptor\_packet[479:448];  
  
 // Companion signals  
 r\_data\_type <= descriptor\_type;  
 r\_packet\_eos\_received <= descriptor\_eos;  
  
 // Decrement credit on descriptor acceptance  
 if (cfg\_use\_credit && (r\_descriptor\_credit\_counter > 0)) begin  
 r\_descriptor\_credit\_counter <= r\_descriptor\_credit\_counter - 1;  
 end  
end

###### Companion Control Signals

In addition to the 512-bit descriptor\_packet, these companion signals provide metadata:

| Signal Name | Width | Description |
| --- | --- | --- |
| descriptor\_type | 2 bits | Descriptor type (from descriptor engine) |
| descriptor\_same | 1 bit | Same descriptor indicator |
| descriptor\_error | 1 bit | Error flag from descriptor engine |
| descriptor\_is\_rda | 1 bit | RDA packet indicator |
| descriptor\_rda\_channel | CHAN\_WIDTH | RDA channel number |
| descriptor\_eos | 1 bit | End of stream marker |
| descriptor\_eol | 1 bit | End of line marker |
| descriptor\_eod | 1 bit | End of data marker |

**Note:** Stream boundary signals (EOS/EOL/EOD) are provided as companion signals from the descriptor engine, not embedded in the descriptor packet itself.

##### Credit Management

###### Exponential Credit Encoding

The scheduler implements **exponential credit encoding** for the cfg\_initial\_credit configuration input, providing a wide range of credit values with a compact 4-bit configuration:

| cfg\_initial\_credit | Binary | Actual Credits | Description |
| --- | --- | --- | --- |
| 0 | 4’b0000 | 1 | Minimum credits (2^0) |
| 1 | 4’b0001 | 2 | Low credits (2^1) |
| 2 | 4’b0010 | 4 | (2^2) |
| 3 | 4’b0011 | 8 | (2^3) |
| 4 | 4’b0100 | 16 | (2^4) |
| 5 | 4’b0101 | 32 | (2^5) |
| 6 | 4’b0110 | 64 | (2^6) |
| 7 | 4’b0111 | 128 | (2^7) |
| 8 | 4’b1000 | 256 | (2^8) |
| 9 | 4’b1001 | 512 | (2^9) |
| 10 | 4’b1010 | 1024 | (2^10) |
| 11 | 4’b1011 | 2048 | (2^11) |
| 12 | 4’b1100 | 4096 | (2^12) |
| 13 | 4’b1101 | 8192 | (2^13) |
| 14 | 4’b1110 | 16384 | Maximum credits (2^14) |
| 15 | 4’b1111 | 0 | **DISABLED** (no credits - blocks all operations) |

**Encoding Rationale:** Exponential encoding allows compact 4-bit configuration to represent a wide range from 1 to 16384 credits, plus a special “disabled” mode (15 = 0 credits). This enables fine-grained control for low-traffic scenarios (1-8 credits) while supporting high-throughput operations (256-16384 credits) without requiring a wide configuration bus. Setting cfg=15 disables the credit system by initializing the counter to 0, effectively blocking all descriptor processing when credit mode is enabled.

###### Credit Initialization

// Reset initialization with exponential encoding  
always\_ff @(posedge clk) begin  
 if (!rst\_n) begin  
 // Exponential credit encoding:  
 // 0->1, 1->2, 2->4, 3->8, ..., 14->16384 (exponential: 2^n)  
 // 15->0 (special case: DISABLED - no credits, blocks all operations)  
 r\_descriptor\_credit\_counter <= (cfg\_initial\_credit == 4'hF) ? 32'h00000000 :  
 (cfg\_initial\_credit == 4'h0) ? 32'h00000001 :  
 (32'h1 << cfg\_initial\_credit);  
 end  
end

###### Credit Operations

// Credit state tracking  
logic [31:0] r\_descriptor\_credit\_counter; // 32-bit to hold decoded value  
logic w\_credit\_available;  
logic w\_credit\_warning;  
  
// Credit availability check  
assign w\_credit\_available = cfg\_use\_credit ?  
 (r\_descriptor\_credit\_counter > 0) : 1'b1;  
  
// Early warning threshold (linear comparison against threshold)  
assign w\_credit\_warning = cfg\_use\_credit ?  
 (r\_descriptor\_credit\_counter <= EARLY\_WARNING\_THRESHOLD) : 1'b0;

###### Credit Update Logic

// Credit decrement on descriptor acceptance (linear operation on decoded value)  
if (descriptor\_valid && descriptor\_ready && cfg\_use\_credit) begin  
 r\_descriptor\_credit\_counter <= r\_descriptor\_credit\_counter - 1;  
end  
  
// Credit increment on external request (linear operation on decoded value)  
if (credit\_increment && cfg\_use\_credit) begin  
 r\_descriptor\_credit\_counter <= r\_descriptor\_credit\_counter + 1;  
end

**Important:** The exponential encoding applies **only to initialization** from cfg\_initial\_credit. Once initialized, the credit counter operates as a standard linear counter (increment by 1, decrement by 1).

##### Control Operation Sequencing

###### Control Read Operations (Pre-Descriptor)

The scheduler issues control read operations BEFORE data operations when ctrlrd\_addr is non-zero:

// Control read needed check  
logic w\_ctrlrd\_needed = (r\_ctrlrd\_addr != 64'h0);  
  
// State transitions from WAIT\_FOR\_CONTROL  
SCHED\_WAIT\_FOR\_CONTROL: begin  
 if (!cfg\_channel\_wait && cfg\_channel\_enable) begin  
 if (w\_ctrlrd\_needed) begin  
 w\_next\_state = SCHED\_ISSUE\_CTRLRD; // Issue control read first  
 end else begin  
 w\_next\_state = SCHED\_DESCRIPTOR\_ACTIVE; // Skip to data operations  
 end  
 end  
end  
  
// Control read state with retry mechanism  
SCHED\_ISSUE\_CTRLRD: begin  
 if (ctrlrd\_ready && !ctrlrd\_error) begin  
 // Read matched expected value - proceed to data operations  
 w\_next\_state = SCHED\_DESCRIPTOR\_ACTIVE;  
 end else if (ctrlrd\_ready && ctrlrd\_error) begin  
 // Max retries exceeded or AXI error - enter error state  
 w\_next\_state = SCHED\_ERROR;  
 end  
 // Else: Stay in SCHED\_ISSUE\_CTRLRD while ctrlrd engine retries  
end  
  
// Control read output signals  
assign ctrlrd\_valid = (r\_current\_state == SCHED\_ISSUE\_CTRLRD);  
assign ctrlrd\_addr = r\_ctrlrd\_addr;  
assign ctrlrd\_data = r\_ctrlrd\_data; // Expected/comparison data  
assign ctrlrd\_mask = r\_ctrlrd\_mask; // Mask/flags for operation

**Ctrlrd Operation Details:** - **Match Success**: ctrlrd\_ready asserts when read data matches expected value -> Proceed to DESCRIPTOR\_ACTIVE - **Retry Loop**: Ctrlrd engine automatically retries reads up to cfg\_ctrlrd\_max\_try times - **Timeout/Error**: ctrlrd\_ready && ctrlrd\_error indicates failure -> Enter SCHED\_ERROR state - **Use Cases**: Flag polling, synchronization checks, pre-condition validation

###### Control Write Operations (Post-Descriptor)

The scheduler issues control write operations AFTER data operations complete:

// Control write needed checks  
logic w\_ctrlwr0\_needed = (r\_ctrlwr0\_addr != 64'h0);  
logic w\_ctrlwr1\_needed = (r\_ctrlwr1\_addr != 64'h0);  
  
// State transitions for control write operations  
case (r\_current\_state)  
 SCHED\_DESCRIPTOR\_ACTIVE: begin  
 if (w\_descriptor\_complete) begin  
 // After data operations, issue control writes if needed  
 if (w\_ctrlwr0\_needed) begin  
 w\_next\_state = SCHED\_ISSUE\_CTRLWR0;  
 end else if (w\_ctrlwr1\_needed) begin  
 w\_next\_state = SCHED\_ISSUE\_CTRLWR1;  
 end else begin  
 w\_next\_state = SCHED\_IDLE;  
 end  
 end  
 end  
  
 SCHED\_ISSUE\_CTRLWR0: begin  
 if (ctrlwr\_ready) begin  
 // After ctrlwr0 completes, issue ctrlwr1 if needed  
 if (w\_ctrlwr1\_needed) begin  
 w\_next\_state = SCHED\_ISSUE\_CTRLWR1;  
 end else begin  
 w\_next\_state = SCHED\_IDLE;  
 end  
 end  
 end  
  
 SCHED\_ISSUE\_CTRLWR1: begin  
 if (ctrlwr\_ready) begin  
 w\_next\_state = SCHED\_IDLE;  
 end  
 end  
endcase  
  
// Control write output multiplexing  
assign ctrlwr\_valid = (r\_current\_state == SCHED\_ISSUE\_CTRLWR0) ||  
 (r\_current\_state == SCHED\_ISSUE\_CTRLWR1);  
assign ctrlwr\_addr = (r\_current\_state == SCHED\_ISSUE\_CTRLWR0) ? r\_ctrlwr0\_addr : r\_ctrlwr1\_addr;  
assign ctrlwr\_data = (r\_current\_state == SCHED\_ISSUE\_CTRLWR0) ? r\_ctrlwr0\_data : r\_ctrlwr1\_data;

###### Null Address Handling

All control operations support conditional execution via null address checking:

// Null address = skip operation  
// ctrlrd\_addr = 64'h0 -> Skip control read operation  
// ctrlwr0\_addr = 64'h0 -> Skip control write 0 operation  
// ctrlwr1\_addr = 64'h0 -> Skip control write 1 operation

This enables descriptor-driven conditional control operations without additional configuration.

##### Channel Reset Coordination

###### Graceful Reset Handling

// Channel reset coordination  
assign w\_safe\_to\_reset = (r\_current\_state == SCHED\_IDLE) &&   
 (r\_alignment\_state == ALIGN\_IDLE) &&  
 w\_no\_pending\_operations &&  
 !r\_channel\_reset\_active;  
  
assign scheduler\_idle = w\_safe\_to\_reset && !r\_channel\_reset\_active;

###### Reset Behavior

1. **Block New Operations**: Stop accepting descriptors during reset
2. **Complete Active Operations**: Finish data transfers and program operations
3. **Reset State Machines**: Both main and alignment FSMs return to idle
4. **Signal Completion**: Assert scheduler\_idle when reset complete

##### Timeout Detection

###### Timeout Monitoring

// Timeout counter for stuck operations  
logic [31:0] r\_timeout\_counter;  
logic w\_timeout\_expired;  
  
assign w\_timeout\_expired = (r\_timeout\_counter >= TIMEOUT\_CYCLES);  
  
// Timeout counter management  
always\_ff @(posedge clk) begin  
 if (!rst\_n || (r\_current\_state == SCHED\_IDLE)) begin  
 r\_timeout\_counter <= 32'h0;  
 end else if (r\_current\_state == SCHED\_DESCRIPTOR\_ACTIVE) begin  
 r\_timeout\_counter <= r\_timeout\_counter + 1;  
 end  
end

##### Error Handling

###### Sticky Error Flags

The scheduler implements **sticky error registers** to capture transient error signals from engines and hold them until the FSM can process the error condition. This ensures errors are never missed due to timing mismatches between engine error assertion and FSM state checking.

**Sticky Error Registers:** - r\_data\_error\_sticky - Captures errors from data engine - r\_ctrlrd\_error\_sticky - Captures errors from control read engine - r\_ctrlwr\_error\_sticky - Captures errors from control write engine

**Purpose:** Engine error signals (e.g., ctrlwr\_error) may assert synchronously with ready signals during handshakes. If the FSM transitions to a new state before checking the error, the transient error signal could be lost. Sticky registers hold errors until the FSM explicitly processes them.

// Persistent error tracking with sticky registers  
logic r\_data\_error\_sticky;  
logic r\_ctrlrd\_error\_sticky;  
logic r\_ctrlwr\_error\_sticky;  
  
// Sticky error flag management  
always\_ff @(posedge clk) begin  
 if (!rst\_n) begin  
 r\_data\_error\_sticky <= 1'b0;  
 r\_ctrlrd\_error\_sticky <= 1'b0;  
 r\_ctrlwr\_error\_sticky <= 1'b0;  
 end else begin  
 // Capture errors when they occur  
 if (data\_error) r\_data\_error\_sticky <= 1'b1;  
 if (ctrlrd\_error) r\_ctrlrd\_error\_sticky <= 1'b1;  
 if (ctrlwr\_error) r\_ctrlwr\_error\_sticky <= 1'b1;  
  
 // Clear sticky errors only when transitioning FROM error state TO idle  
 // This ensures errors are held long enough to trigger ERROR state  
 if (r\_current\_state == SCHED\_ERROR && w\_next\_state == SCHED\_IDLE) begin  
 r\_data\_error\_sticky <= 1'b0;  
 r\_ctrlrd\_error\_sticky <= 1'b0;  
 r\_ctrlwr\_error\_sticky <= 1'b0;  
 end  
 end  
end

**Critical Timing:** Sticky errors are cleared **only during ERROR->IDLE transition**, NOT on IDLE state entry. This prevents the following timing bug:

**[FAIL] Wrong Timing (Bug):**

Cycle N: FSM in SCHED\_ISSUE\_CTRLWR0, ctrlwr\_error=1 & ctrlwr\_ready=1  
 FSM transitions to SCHED\_IDLE (missed error check!)  
 r\_ctrlwr\_error\_sticky <= 1 (takes effect next cycle)  
Cycle N+1: FSM in SCHED\_IDLE, r\_ctrlwr\_error\_sticky=1  
 Clear on IDLE entry -> r\_ctrlwr\_error\_sticky <= 0  
 Error lost before detection logic sees it!

**[PASS] Correct Timing (Fixed):**

Cycle N: FSM in SCHED\_ISSUE\_CTRLWR0, ctrlwr\_error=1 & ctrlwr\_ready=1  
 General error detection sees ctrlwr\_error=1 OR r\_ctrlwr\_error\_sticky=1  
 FSM transitions to SCHED\_ERROR (error caught!)  
Cycle N+1: FSM in SCHED\_ERROR  
 Process error state...  
Cycle N+M: FSM transitions ERROR->IDLE  
 Clear sticky errors now that error has been processed

###### Error Detection Logic

**General Error Detection (applies in all states):**

// Check transient errors OR sticky errors  
if (data\_error || ctrlrd\_error || ctrlwr\_error || descriptor\_error ||  
 w\_timeout\_expired || r\_data\_error\_sticky || r\_ctrlrd\_error\_sticky || r\_ctrlwr\_error\_sticky) begin  
 w\_next\_state = SCHED\_ERROR;  
end

**State-Specific Error Checking (for ctrlrd):**

// SCHED\_ISSUE\_CTRLRD state has explicit error checking  
SCHED\_ISSUE\_CTRLRD: begin  
 if (ctrlrd\_ready && !ctrlrd\_error) begin  
 w\_next\_state = SCHED\_DESCRIPTOR\_ACTIVE; // Success  
 end else if (ctrlrd\_ready && ctrlrd\_error) begin  
 w\_next\_state = SCHED\_ERROR; // Error detected  
 end  
end

**Note:** Control write states (SCHED\_ISSUE\_CTRLWR0/CTRLWR1) rely on general error detection logic, not state-specific checks. The sticky error registers ensure ctrlwr errors are captured even if the FSM transitions before checking.

###### Error Recovery

// Error state exit conditions - check both transient and sticky errors  
logic w\_all\_errors\_clear = !data\_error && !ctrlrd\_error && !ctrlwr\_error &&  
 !descriptor\_error && !w\_timeout\_expired &&  
 !r\_data\_error\_sticky && !r\_ctrlrd\_error\_sticky && !r\_ctrlwr\_error\_sticky;  
  
// Error state recovery to IDLE  
if ((r\_current\_state == SCHED\_ERROR) && w\_all\_errors\_clear && w\_credit\_available) begin  
 w\_next\_state = SCHED\_IDLE; // Sticky errors cleared during this transition  
end

**Error Recovery Sequence:** 1. Error detected (transient or sticky) -> Enter SCHED\_ERROR state 2. Wait in SCHED\_ERROR until all error sources clear 3. Transition SCHED\_ERROR -> SCHED\_IDLE 4. Sticky error registers cleared during this transition 5. Ready for new descriptor processing

##### Performance Benefits of Address Alignment FSM

###### Hidden Latency Calculation

The parallel address alignment FSM provides significant performance benefits:

1. **Pre-calculation During Descriptor Processing**: Alignment analysis occurs during the non-critical descriptor processing phase
2. **Immediate Availability**: AXI engines receive pre-calculated alignment information immediately when needed
3. **No Critical Path Impact**: No alignment calculation overhead in AXI transaction timing
4. **Optimal Transfer Planning**: Complete transfer sequence pre-calculated with optimized burst patterns

###### Transfer Strategy Selection

// Transfer strategy enumeration  
typedef enum logic [2:0] {  
 STRATEGY\_SINGLE = 3'h0, // Single aligned transfer  
 STRATEGY\_ALIGNMENT = 3'h1, // Alignment + streaming  
 STRATEGY\_STREAMING = 3'h2, // Pure streaming transfers  
 STRATEGY\_FINAL = 3'h3, // Alignment + streaming + final  
 STRATEGY\_PRECISION = 3'h4 // Precision chunk-level transfers  
} alignment\_strategy\_t;

###### Chunk Enable Generation

// First transfer chunk enable calculation  
function logic [NUM\_CHUNKS-1:0] calc\_first\_chunk\_enables(  
 input logic [5:0] addr\_offset,  
 input logic [31:0] transfer\_bytes  
);  
 logic [3:0] start\_chunk = addr\_offset[5:2]; // Starting chunk index  
 logic [3:0] num\_chunks = (transfer\_bytes + 3) >> 2; // Number of chunks needed  
 logic [NUM\_CHUNKS-1:0] mask = (1 << num\_chunks) - 1; // Create mask  
 return mask << start\_chunk; // Shift to correct position  
endfunction  
  
// Final transfer chunk enable calculation   
function logic [NUM\_CHUNKS-1:0] calc\_final\_chunk\_enables(  
 input logic [31:0] final\_bytes  
);  
 logic [3:0] num\_chunks = (final\_bytes + 3) >> 2; // Number of chunks needed  
 return (1 << num\_chunks) - 1; // Mask from bit 0  
endfunction

##### Monitor Bus Events

The scheduler generates comprehensive monitor events:

###### Error Events

* **Timeout Error**: Data transfer timeout detection
* **Credit Exhausted**: Credit underflow condition
* **Descriptor Error**: Invalid descriptor content
* **Program Error**: Program operation failure
* **Alignment Error**: Invalid address alignment parameters

###### Performance Events

* **Descriptor Processing**: Descriptor execution start/completion
* **Credit Warning**: Early warning threshold reached
* **Transfer Phase**: Transfer phase transitions (alignment/streaming/final)
* **Program Sequence**: Program operation sequence tracking
* **Alignment Calculation**: Address alignment FSM operation timing

###### Completion Events

* **Data Transfer Complete**: Data operation completion
* **Program Complete**: Program operation completion
* **Stream Boundary**: EOS processing completion
* **Channel Reset**: Channel reset sequence completion
* **RDA Processing**: RDA packet processing completion

##### Usage Guidelines

###### Address Alignment Optimization

The address alignment system provides optimal AXI performance:

// Example alignment information usage in AXI engine  
if (data\_alignment\_valid) begin  
 case (data\_transfer\_phase)  
 PHASE\_ALIGNMENT: begin  
 axi\_addr <= data\_alignment\_info.aligned\_addr;  
 axi\_len <= data\_alignment\_info.first\_burst\_len;  
 chunk\_enables <= data\_alignment\_info.first\_chunk\_enables;  
 end  
 PHASE\_STREAMING: begin  
 axi\_addr <= aligned\_streaming\_addr;  
 axi\_len <= data\_alignment\_info.optimal\_burst\_len;  
 chunk\_enables <= 16'hFFFF; // All chunks valid  
 end  
 PHASE\_FINAL: begin  
 axi\_addr <= final\_transfer\_addr;  
 axi\_len <= data\_alignment\_info.final\_burst\_len;  
 chunk\_enables <= data\_alignment\_info.final\_chunk\_enables;  
 end  
 endcase  
end

###### Performance Monitoring

Monitor key performance indicators: - Credit utilization and warning frequency - Timeout occurrence and duration - Address alignment efficiency - Program sequence timing - Transfer phase distribution

###### Error Recovery

The scheduler provides comprehensive error handling: - Monitor timeout conditions for stuck transfers - Track credit exhaustion for flow control issues - Detect alignment calculation errors - Coordinate error recovery across all interfaces - Use sticky error flags for persistent fault tracking

###### Channel Reset Usage

// Example channel reset coordination  
logic reset\_request;  
logic reset\_complete;  
  
assign reset\_complete = scheduler\_idle;  
  
always\_ff @(posedge clk) begin  
 if (reset\_request && !cfg\_channel\_reset) begin  
 cfg\_channel\_reset <= 1'b1;  
 end else if (cfg\_channel\_reset && reset\_complete) begin  
 cfg\_channel\_reset <= 1'b0;  
 // Channel is now safely reset and ready for operation  
 end  
end

This dual-FSM scheduler architecture provides optimal performance through pre-calculated address alignment while maintaining comprehensive control over descriptor execution, credit management, and program sequencing operations. #### Descriptor Engine

##### Overview

The Descriptor Engine manages descriptor fetching and buffering operations with sophisticated dual-path processing for APB and RDA requests. The module implements a six-state state machine that handles descriptor address management, AXI read operations, descriptor parsing, and prefetch optimization with comprehensive stream boundary support and channel reset coordination.

|  |
| --- |
| descriptor engine |

descriptor engine

##### Key Features

* **Dual-Path Processing**: Handles both APB programming interface and RDA packet interface
* **Six-State State Machine**: Comprehensive descriptor lifecycle management
* **Stream Boundary Support**: Complete EOS/EOL/EOD field extraction and propagation
* **AXI Read Coordination**: Shared AXI interface with channel ID-based response routing
* **4-Deep Descriptor FIFO**: Maintains descriptor flow for continuous operation
* **Priority Handling**: RDA packets prioritized over APB requests
* **Channel Reset Support**: Graceful shutdown with proper AXI transaction completion
* **Monitor Integration**: Rich monitor events for descriptor processing visibility

##### Module Interface

###### Configuration Parameters

| Parameter | Default Value | Description |
| --- | --- | --- |
| CHANNEL\_ID | 0 | Static channel identifier for this engine instance |
| NUM\_CHANNELS | 32 | Total number of channels in system |
| CHAN\_WIDTH | $clog2(NUM\_CHANNELS) | Width of channel address fields |
| ADDR\_WIDTH | 64 | Address width for AXI transactions |
| DATA\_WIDTH | 512 | Descriptor packet width |
| AXI\_ID\_WIDTH | 8 | AXI transaction ID width |

###### Clock and Reset Signals

| Signal Name | Type | Width | Direction | Required | Description |
| --- | --- | --- | --- | --- | --- |
| **clk** | logic | 1 | Input | Yes | System clock |
| **rst\_n** | logic | 1 | Input | Yes | Active-low asynchronous reset |

###### APB Programming Interface

| Signal Name | Type | Width | Direction | Required | Description |
| --- | --- | --- | --- | --- | --- |
| **apb\_valid** | logic | 1 | Input | Yes | APB request valid |
| **apb\_ready** | logic | 1 | Output | Yes | APB request ready |
| **apb\_addr** | logic | ADDR\_WIDTH | Input | Yes | Descriptor address |

###### RDA Packet Interface (From Network Slave)

| Signal Name | Type | Width | Direction | Required | Description |
| --- | --- | --- | --- | --- | --- |
| **rda\_valid** | logic | 1 | Input | Yes | RDA packet valid |
| **rda\_ready** | logic | 1 | Output | Yes | RDA packet ready |
| **rda\_packet** | logic | DATA\_WIDTH | Input | Yes | RDA packet data |
| **rda\_channel** | logic | CHAN\_WIDTH | Input | Yes | RDA channel identifier |

###### Scheduler Interface

| Signal Name | Type | Width | Direction | Required | Description |
| --- | --- | --- | --- | --- | --- |
| **descriptor\_valid** | logic | 1 | Output | Yes | Descriptor available |
| **descriptor\_ready** | logic | 1 | Input | Yes | Ready to accept descriptor |
| **descriptor\_packet** | logic | DATA\_WIDTH | Output | Yes | Descriptor data |
| **descriptor\_same** | logic | 1 | Output | Yes | Same descriptor flag |
| **descriptor\_error** | logic | 1 | Output | Yes | Descriptor error |
| **descriptor\_is\_rda** | logic | 1 | Output | Yes | RDA packet indicator |
| **descriptor\_rda\_channel** | logic | CHAN\_WIDTH | Output | Yes | RDA channel identifier |
| **descriptor\_eos** | logic | 1 | Output | Yes | End of Stream |
| **descriptor\_eol** | logic | 1 | Output | Yes | End of Line |
| **descriptor\_eod** | logic | 1 | Output | Yes | End of Data |
| **descriptor\_type** | logic | 2 | Output | Yes | Packet type |

###### Configuration Interface

| Signal Name | Type | Width | Direction | Required | Description |
| --- | --- | --- | --- | --- | --- |
| **cfg\_addr0** | logic | ADDR\_WIDTH | Input | Yes | Address range 0 base |
| **cfg\_addr1** | logic | ADDR\_WIDTH | Input | Yes | Address range 1 base |
| **cfg\_channel\_reset** | logic | 1 | Input | Yes | Dynamic channel reset request |

###### Status Interface

| Signal Name | Type | Width | Direction | Required | Description |
| --- | --- | --- | --- | --- | --- |
| **descriptor\_engine\_idle** | logic | 1 | Output | Yes | Engine idle status indicator |

###### Shared AXI4 Master Read Interface (512-bit)

| Signal Name | Type | Width | Direction | Required | Description |
| --- | --- | --- | --- | --- | --- |
| **ar\_valid** | logic | 1 | Output | Yes | Read address valid |
| **ar\_ready** | logic | 1 | Input | Yes | Read address ready (arbitrated) |
| **ar\_addr** | logic | ADDR\_WIDTH | Output | Yes | Read address |
| **ar\_len** | logic | 8 | Output | Yes | Burst length - 1 (always 0) |
| **ar\_size** | logic | 3 | Output | Yes | Transfer size (3’b110 for 64 bytes) |
| **ar\_burst** | logic | 2 | Output | Yes | Burst type (2’b01 INCR) |
| **ar\_id** | logic | AXI\_ID\_WIDTH | Output | Yes | Transaction ID (channel-based) |
| **ar\_lock** | logic | 1 | Output | Yes | Lock type (always 0) |
| **ar\_cache** | logic | 4 | Output | Yes | Cache attributes |
| **ar\_prot** | logic | 3 | Output | Yes | Protection attributes |
| **ar\_qos** | logic | 4 | Output | Yes | Quality of service |
| **ar\_region** | logic | 4 | Output | Yes | Region identifier |
| **r\_valid** | logic | 1 | Input | Yes | Read data valid |
| **r\_ready** | logic | 1 | Output | Yes | Read data ready |
| **r\_data** | logic | DATA\_WIDTH | Input | Yes | Read data |
| **r\_resp** | logic | 2 | Input | Yes | Read response |
| **r\_last** | logic | 1 | Input | Yes | Read last |
| **r\_id** | logic | AXI\_ID\_WIDTH | Input | Yes | Read ID |

###### Monitor Bus Interface

| Signal Name | Type | Width | Direction | Required | Description |
| --- | --- | --- | --- | --- | --- |
| **mon\_valid** | logic | 1 | Output | Yes | Monitor packet valid |
| **mon\_ready** | logic | 1 | Input | Yes | Monitor ready to accept packet |
| **mon\_packet** | logic | 64 | Output | Yes | Monitor packet data |

##### Descriptor Packet Format

###### Control Fields

| Field | Bits | Width | Description | Values |
| --- | --- | --- | --- | --- |
| **c0de** | [191:180] | 12 | Code field | Implementation specific |
| **tbd** | [179:165] | 15 | To be defined | Reserved |
| **eos** | [164] | 1 | End of Stream marker | 1=EOS, 0=Normal |
| **eol** | [163] | 1 | End of Line marker | 1=EOL, 0=Normal |
| **eod** | [162] | 1 | End of Data marker | 1=EOD, 0=Normal |
| **type** | [161:160] | 2 | Packet type | See table below |

###### Packet Type Encoding

| Type | Value | Description |
| --- | --- | --- |
| **FC Tile** | 2’b00 | Flow Control Tile packets |
| **TS Data** | 2’b01 | Time Series Data packets |
| **Risc-V** | 2’b10 | RISC-V processor packets |
| **RAW Data** | 2’b11 | Raw data packets |

###### Address and Data Fields

| Field | Bits | Width | Description |
| --- | --- | --- | --- |
| **Next Descriptor** | [159:128], [127:96] | 64 | Next descriptor address |
| **Data Address** | [95:64], [63:32] | 64 | Data transfer address |
| **Data Length** | [31:0] | 32 | Data transfer length |
| **Program 0 Address** | [287:256], [255:224] | 64 | Program 0 write address |
| **Program 0 Data** | [223:192] | 32 | Program 0 write data |
| **Program 1 Address** | [351:320], [319:288] | 64 | Program 1 write address |
| **Program 1 Data** | [383:352] | 32 | Program 1 write data |

##### State Machine Operation

###### State Definitions

| State | Description |
| --- | --- |
| **IDLE** | Ready for APB or RDA requests, monitor operation exclusivity |
| **ISSUE\_ADDR** | Issue AXI read transaction for APB requests |
| **WAIT\_DATA** | Wait for AXI read completion with channel ID matching |
| **COMPLETE** | Process descriptor data and generate output |
| **ERROR** | Handle errors with recovery capability |

###### Priority Processing

The Descriptor Engine implements sophisticated priority processing:

1. **RDA Priority**: RDA packets always processed before APB requests
2. **Operation Exclusivity**: Only one operation type active at a time
3. **Channel Reset Coordination**: Graceful shutdown with AXI completion

##### Architecture

##### Descriptor Engine FSM

The Descriptor Engine implements a sophisticated six-state finite state machine that orchestrates descriptor fetching and processing operations with dual-path support for both APB programming interface requests and RDA packet interface operations. The FSM manages the complete descriptor lifecycle from initial request validation through AXI read transactions, descriptor parsing, and final output generation.

|  |
| --- |
| Descriptor Engine FSM |

Descriptor Engine FSM

**Key States:** - **DESC\_IDLE**: Ready for APB or RDA requests with operation exclusivity monitoring - **DESC\_AXI\_READ**: Issues AXI read transactions for APB descriptor fetch requests  
- **DESC\_WAIT\_READ**: Waits for AXI read completion with channel ID-based response routing - **DESC\_CHECK**: Validates descriptor content and extracts stream boundary information - **DESC\_LOAD**: Processes descriptor data and generates enhanced output with metadata - **DESC\_ERROR**: Handles error conditions with comprehensive recovery capabilities

The FSM implements intelligent priority processing where RDA packets always take precedence over APB requests, ensuring optimal network responsiveness. Stream boundary support includes complete EOS/EOL/EOD field extraction and propagation, while the 4-deep descriptor FIFO maintains continuous operation flow. Channel reset coordination provides graceful shutdown capabilities, completing any in-flight AXI transactions before asserting idle status for system-level reset coordination.

###### FIFO Management

APB Request Skid Buffer

// APB request skid buffer (2-deep)  
gaxi\_skid\_buffer #(  
 .DATA\_WIDTH(ADDR\_WIDTH),  
 .DEPTH(2),  
 .INSTANCE\_NAME("APB\_REQ\_SKID")  
) i\_apb\_request\_skid\_buffer (  
 .axi\_aclk(clk),  
 .axi\_aresetn(rst\_n),  
 .wr\_valid(apb\_valid),  
 .wr\_ready(apb\_ready),  
 .wr\_data(apb\_addr),  
 .rd\_valid(w\_apb\_skid\_valid\_out),  
 .rd\_ready(w\_apb\_skid\_ready\_out),  
 .rd\_data(w\_apb\_skid\_dout),  
 .count(),  
 .rd\_count()  
);

RDA Packet Skid Buffer

// RDA packet skid buffer (4-deep)  
gaxi\_skid\_buffer #(  
 .DATA\_WIDTH(DATA\_WIDTH + CHAN\_WIDTH),  
 .DEPTH(4),  
 .INSTANCE\_NAME("RDA\_PKT\_SKID")  
) i\_rda\_packet\_skid\_buffer (  
 .axi\_aclk(clk),  
 .axi\_aresetn(rst\_n),  
 .wr\_valid(rda\_valid),  
 .wr\_ready(rda\_ready),  
 .wr\_data({rda\_packet, rda\_channel}),  
 .rd\_valid(w\_rda\_skid\_valid\_out),  
 .rd\_ready(w\_rda\_skid\_ready\_out),  
 .rd\_data(w\_rda\_skid\_dout),  
 .count(),  
 .rd\_count()  
);

Descriptor Output FIFO

// Descriptor output FIFO (4-deep)  
gaxi\_skid\_buffer #(  
 .DATA\_WIDTH($bits(enhanced\_descriptor\_t)),  
 .DEPTH(4),  
 .INSTANCE\_NAME("DESC\_FIFO")  
) i\_descriptor\_fifo (  
 .axi\_aclk(clk),  
 .axi\_aresetn(rst\_n),  
 .wr\_valid(w\_desc\_fifo\_wr\_valid),  
 .wr\_ready(w\_desc\_fifo\_wr\_ready),  
 .wr\_data(w\_desc\_fifo\_wr\_data),  
 .rd\_valid(w\_desc\_fifo\_rd\_valid),  
 .rd\_ready(w\_desc\_fifo\_rd\_ready),  
 .rd\_data(w\_desc\_fifo\_rd\_data),  
 .count(),  
 .rd\_count()  
);

##### Monitor Events

###### Event Generation

// Monitor event types:  
// - CORE\_COMPL\_DESCRIPTOR\_LOADED: Descriptor processing completed  
// - Network\_STREAM\_END: EOS boundary detected  
// - Custom EOL/EOD events: Stream boundary processing  
// - CORE\_ERR\_DESCRIPTOR\_BAD\_ADDR: Address validation failed  
// - AXI\_ERR\_RESP\_SLVERR: AXI slave error  
// - AXI\_ERR\_RESP\_DECERR: AXI decode error

##### Performance Characteristics

###### Processing Rates

* **Descriptor Throughput**: 1 descriptor per cycle when FIFO space available
* **AXI Read Latency**: 10-20 cycles depending on arbitration and memory response
* **RDA Processing**: 1-2 cycles for direct processing without AXI
* **Validation Overhead**: <1 cycle for comprehensive validation

###### Resource Utilization

* **Skid Buffers**: 2-deep APB + 4-deep RDA + 4-deep output = 10 total entries
* **State Machine**: 6 states with efficient encoding
* **AXI Interface**: Standard AXI4 signals with channel ID embedding
* **Validation Logic**: Minimal overhead for address and stream boundary checking

### Sink Data Path

#### Sink Data Path

##### Overview

The Sink Data Path provides a complete integrated data reception and processing pipeline that combines AXI-Stream packet reception, multi-channel SRAM buffering, multi-channel AXI write arbitration, and comprehensive monitor bus aggregation. This wrapper manages the complete data flow from AXIS interface reception through final AXI memory writes.

**RTL Module:** rtl/amba/axis/axis\_slave.sv (AXIS interface) + RAPIDS sink components

The wrapper implements sophisticated TLAST (End of Stream) flow control where packet-level TLAST is managed by SRAM control and descriptor-level EOS completion is coordinated with the scheduler, ensuring proper stream boundary handling throughout the pipeline.

|  |
| --- |
| sink data path |

sink data path

##### Key Features

* **Complete Data Reception Pipeline**: From AXIS packets to AXI memory writes
* **Standard AXIS Interface**: Industry-standard AXI-Stream protocol (no custom credits/ACKs)
* **Multi-Channel SRAM Buffering**: Independent buffering for up to 32 channels
* **Advanced AXI Write Engine**: Multi-channel arbitration with transfer strategy optimization
* **TLAST Flow Management**: Packet-level and descriptor-level boundary coordination
* **RDA Packet Routing**: Direct RDA packet interfaces bypassing SRAM buffering
* **Monitor Bus Aggregation**: Unified monitoring from AXIS slave, SRAM control, and AXI engine
* **Enhanced Scheduler Interface**: Address alignment bus support for optimal AXI performance

##### Interface Specification

###### Clock and Reset

| Signal Name | Type | Width | Direction | Required | Description |
| --- | --- | --- | --- | --- | --- |
| **clk** | logic | 1 | Input | Yes | System clock |
| **rst\_n** | logic | 1 | Input | Yes | Active-low asynchronous reset |

###### AXI-Stream Slave Interface (RX)

| Signal Name | Type | Width | Direction | Required | Description |
| --- | --- | --- | --- | --- | --- |
| **axis\_snk\_rx\_tdata** | logic | DATA\_WIDTH | Input | Yes | Stream data payload |
| **axis\_snk\_rx\_tstrb** | logic | DATA\_WIDTH/8 | Input | Yes | Byte strobes (write enables) |
| **axis\_snk\_rx\_tlast** | logic | 1 | Input | Yes | Last transfer in packet |
| **axis\_snk\_rx\_tvalid** | logic | 1 | Input | Yes | Stream data valid |
| **axis\_snk\_rx\_tready** | logic | 1 | Output | Yes | Stream ready (backpressure) |
| **axis\_snk\_rx\_tuser** | logic | 16 | Input | Yes | User sideband (packet metadata) |

**TUSER Encoding (Sink RX):**

[15:8] - Channel ID  
[7:0] - Packet type/flags

**Note:** AXIS uses standard tvalid/tready backpressure. No ACK channels or custom credit mechanisms.

###### RDA Interfaces (Direct Bypass)

| Signal Name | Type | Width | Direction | Required | Description |
| --- | --- | --- | --- | --- | --- |
| **rda\_src\_valid** | logic | 1 | Output | Yes | RDA source packet valid |
| **rda\_src\_ready** | logic | 1 | Input | Yes | RDA source packet ready |
| **rda\_src\_packet** | logic | DATA\_WIDTH | Output | Yes | RDA source packet data |
| **rda\_src\_channel** | logic | CHAN\_WIDTH | Output | Yes | RDA source channel |
| **rda\_src\_eos** | logic | 1 | Output | Yes | RDA source End of Stream |
| **rda\_snk\_valid** | logic | 1 | Output | Yes | RDA sink packet valid |
| **rda\_snk\_ready** | logic | 1 | Input | Yes | RDA sink packet ready |
| **rda\_snk\_packet** | logic | DATA\_WIDTH | Output | Yes | RDA sink packet data |
| **rda\_snk\_channel** | logic | CHAN\_WIDTH | Output | Yes | RDA sink channel |
| **rda\_snk\_eos** | logic | 1 | Output | Yes | RDA sink End of Stream |

###### Multi-Channel Scheduler Interface

| Signal Name | Type | Width | Direction | Required | Description |
| --- | --- | --- | --- | --- | --- |
| **data\_valid** | logic | NUM\_CHANNELS | Input | Yes | Data transfer request per channel |
| **data\_ready** | logic | NUM\_CHANNELS | Output | Yes | Data transfer ready per channel |
| **data\_address** | logic | ADDR\_WIDTH x NUM\_CHANNELS | Input | Yes | Data address per channel |
| **data\_length** | logic | 32 x NUM\_CHANNELS | Input | Yes | Data length per channel |
| **data\_type** | logic | 2 x NUM\_CHANNELS | Input | Yes | Data type per channel |
| **data\_eos** | logic | NUM\_CHANNELS | Input | Yes | End of Stream per channel |
| **data\_transfer\_length** | logic | 32 x NUM\_CHANNELS | Output | Yes | Actual transfer length per channel |
| **data\_done\_strobe** | logic | NUM\_CHANNELS | Output | Yes | Transfer completion per channel |
| **data\_error** | logic | NUM\_CHANNELS | Output | Yes | Transfer error per channel |

###### Address Alignment Bus Interface

| Signal Name | Type | Width | Direction | Required | Description |
| --- | --- | --- | --- | --- | --- |
| **data\_alignment\_info** | alignment\_info\_t | NUM\_CHANNELS | Input | Yes | Alignment information per channel |
| **data\_alignment\_valid** | logic | NUM\_CHANNELS | Input | Yes | Alignment information valid per channel |
| **data\_alignment\_ready** | logic | NUM\_CHANNELS | Output | Yes | Alignment information ready per channel |
| **data\_alignment\_next** | logic | NUM\_CHANNELS | Output | Yes | Request next alignment per channel |
| **data\_transfer\_phase** | transfer\_phase\_t | NUM\_CHANNELS | Input | Yes | Transfer phase per channel |
| **data\_sequence\_complete** | logic | NUM\_CHANNELS | Output | Yes | Transfer sequence complete per channel |

###### EOS Completion Interface

| Signal Name | Type | Width | Direction | Required | Description |
| --- | --- | --- | --- | --- | --- |
| **eos\_completion\_valid** | logic | 1 | Output | Yes | EOS completion notification valid |
| **eos\_completion\_ready** | logic | 1 | Input | Yes | EOS completion notification ready |
| **eos\_completion\_channel** | logic | CHAN\_WIDTH | Output | Yes | Channel with EOS completion |

###### AXI4 Master Write Interface

| Signal Name | Type | Width | Direction | Required | Description |
| --- | --- | --- | --- | --- | --- |
| **m\_axi\_awvalid** | logic | 1 | Output | Yes | Write address valid |
| **m\_axi\_awready** | logic | 1 | Input | Yes | Write address ready |
| **m\_axi\_awaddr** | logic | AXI\_ADDR\_WIDTH | Output | Yes | Write address |
| **m\_axi\_awlen** | logic | 8 | Output | Yes | Burst length |
| **m\_axi\_awsize** | logic | 3 | Output | Yes | Burst size |
| **m\_axi\_awburst** | logic | 2 | Output | Yes | Burst type |
| **m\_axi\_awid** | logic | AXI\_ID\_WIDTH | Output | Yes | Write ID |
| **m\_axi\_awlock** | logic | 1 | Output | Yes | Lock type |
| **m\_axi\_awcache** | logic | 4 | Output | Yes | Cache type |
| **m\_axi\_awprot** | logic | 3 | Output | Yes | Protection type |
| **m\_axi\_awqos** | logic | 4 | Output | Yes | QoS identifier |
| **m\_axi\_awregion** | logic | 4 | Output | Yes | Region identifier |
| **m\_axi\_wvalid** | logic | 1 | Output | Yes | Write data valid |
| **m\_axi\_wready** | logic | 1 | Input | Yes | Write data ready |
| **m\_axi\_wdata** | logic | DATA\_WIDTH | Output | Yes | Write data |
| **m\_axi\_wstrb** | logic | DATA\_WIDTH/8 | Output | Yes | Write strobes |
| **m\_axi\_wlast** | logic | 1 | Output | Yes | Write last |
| **m\_axi\_bvalid** | logic | 1 | Input | Yes | Write response valid |
| **m\_axi\_bready** | logic | 1 | Output | Yes | Write response ready |
| **m\_axi\_bresp** | logic | 2 | Input | Yes | Write response |
| **m\_axi\_bid** | logic | AXI\_ID\_WIDTH | Input | Yes | Write ID |

###### Monitor Bus Interface

| Signal Name | Type | Width | Direction | Required | Description |
| --- | --- | --- | --- | --- | --- |
| **mon\_valid** | logic | 1 | Output | Yes | Monitor packet valid |
| **mon\_ready** | logic | 1 | Input | Yes | Monitor ready to accept packet |
| **mon\_packet** | logic | 64 | Output | Yes | Monitor packet data |

###### Status and Error Reporting

| Signal Name | Type | Width | Direction | Required | Description |
| --- | --- | --- | --- | --- | --- |
| **channel\_eos\_pending** | logic | NUM\_CHANNELS | Output | Yes | EOS pending per channel |
| **error\_header\_parity** | logic | 1 | Output | Yes | Header parity error |
| **error\_body\_parity** | logic | 1 | Output | Yes | Body parity error |
| **error\_buffer\_overflow** | logic | 1 | Output | Yes | Buffer overflow error |
| **error\_ack\_lost** | logic | 1 | Output | Yes | ACK lost error |
| **error\_channel\_id** | logic | CHAN\_WIDTH | Output | Yes | Channel with error |
| **packet\_count** | logic | 32 | Output | Yes | Total packet count |
| **error\_count** | logic | 16 | Output | Yes | Total error count |

##### Architecture

###### Internal Components

* **AXIS Slave**: Packet reception, validation, and routing with standard AXIS flow control
* **Sink SRAM Control**: Multi-channel buffering with TLAST/EOS completion signaling
* **Sink AXI Write Engine**: Multi-channel arbitration and AXI write operations
* **Monitor Bus Aggregator**: Round-robin aggregation from all three components

###### Data Flow Pipeline

1. **AXIS Packet Reception**: AXIS slave receives and validates incoming packets
2. **Packet Classification**: Packets routed to SRAM (FC/TS/RAW) or RDA bypass (RDA packets)
3. **Multi-Channel Buffering**: SRAM control provides independent per-channel buffering
4. **AXI Write Arbitration**: AXI engine arbitrates between channels using scheduler inputs
5. **TLAST Completion**: Packet-level TLAST triggers descriptor completion notification

###### TLAST Flow Management

The sink data path implements sophisticated stream boundary handling: - **Packet-Level TLAST**: Detected by AXIS slave, managed by SRAM control - **EOS Completion Interface**: SRAM control notifies scheduler of descriptor completion - **Stream Boundaries**: TLAST triggers proper completion signaling (maps to internal EOS)

###### Address Alignment Integration

The wrapper supports the scheduler’s address alignment bus, enabling: - **Pre-calculated Alignment**: Scheduler provides alignment information before transfers - **Optimal AXI Performance**: No alignment calculation overhead in AXI critical path - **Transfer Strategy Selection**: Alignment information drives AXI burst optimization

###### Multi-Channel AXI Arbitration

The AXI write engine implements sophisticated arbitration: - **Round-Robin Base**: Fair arbitration across active channels - **Transfer Strategy**: Precision/aligned/forced/single transfer modes - **Buffer-Aware**: Considers SRAM buffer status for optimal performance - **TSTRB-Based Strobes**: Precise write strobes based on AXIS byte strobes

##### AXIS Integration

The sink data path uses standard AXI-Stream (AXIS4) protocol for packet reception:

**Key Benefits:** 1. **Industry Standard**: AXIS is widely supported, well-documented protocol 2. **Simplified Flow Control**: Standard tvalid/tready backpressure (no custom ACK channels) 3. **Cleaner Byte Qualification**: Standard tstrb replaces custom chunk enables 4. **Packet Framing**: Standard tlast replaces custom EOS markers 5. **Better Tool Support**: Standard protocol enables better IP integration and verification

**TSTRB vs Legacy Chunk Enables:** - AXIS: 64-bit byte strobes for 512-bit data (byte-level granularity) - Legacy: 16-bit chunk enables (32-bit chunk granularity) - TSTRB provides finer control and maps directly to AXI4 wstrb

##### Usage Guidelines

###### Performance Optimization

* Configure SRAM depths based on expected buffering requirements
* Use address alignment bus for optimal AXI transfer planning
* Monitor buffer status and arbitration efficiency
* Adjust transfer strategies based on workload characteristics

###### Error Handling

The wrapper provides comprehensive error reporting: - Monitor TSTRB errors for data integrity - Check buffer overflow conditions - Verify AXIS protocol compliance - Track per-channel error statistics

###### TLAST Processing

Proper stream boundary handling requires: 1. Monitor packet-level TLAST from AXIS slave 2. Track EOS completion notifications to scheduler (TLAST -> EOS mapping) 3. Coordinate descriptor completion with stream boundaries 4. Use standard AXIS backpressure (tready) for flow control #### Network Slave

##### Overview

The Network Slave receives and processes packets from the Network network with comprehensive validation, intelligent routing, and bulletproof ACK generation. The module implements deep buffering architecture with perfect data transfer guarantees, enhanced error detection, and sophisticated packet classification for optimal system performance.

##### Key Features

* **Perfect Data Transfer**: Zero packet loss and zero ACK loss guarantees
* **Comprehensive Validation**: Multi-layer validation for data integrity
* **Intelligent Packet Routing**: Automatic classification and routing to appropriate destinations
* **Deep Skid Buffering**: 8-entry buffers for robust flow control
* **Bulletproof ACK Generation**: FIFO-based ACK system prevents ACK loss
* **Stream Boundary Support**: Complete EOS processing and tracking
* **Error Detection**: Comprehensive error isolation and reporting
* **Monitor Integration**: Rich monitor events for system visibility

##### Interface Specification

###### Configuration Parameters

| Parameter | Default Value | Description |
| --- | --- | --- |
| NUM\_CHANNELS | 32 | Number of virtual channels |
| CHAN\_WIDTH | $clog2(NUM\_CHANNELS) | Width of channel address fields |
| DATA\_WIDTH | 512 | Data width for packet interfaces |
| ADDR\_WIDTH | 64 | Address width for Network packets |
| NUM\_CHUNKS | 16 | Number of 32-bit chunks (512/32) |
| DEPTH | 8 | Skid buffer depth for robust flow control |

###### Clock and Reset Signals

| Signal Name | Type | Width | Direction | Required | Description |
| --- | --- | --- | --- | --- | --- |
| **clk** | logic | 1 | Input | Yes | System clock |
| **rst\_n** | logic | 1 | Input | Yes | Active-low asynchronous reset |

###### Network Network Interface (Slave)

| Signal Name | Type | Width | Direction | Required | Description |
| --- | --- | --- | --- | --- | --- |
| **s\_network\_pkt\_addr** | logic | ADDR\_WIDTH | Input | Yes | Network packet address |
| **s\_network\_pkt\_addr\_par** | logic | 1 | Input | Yes | Network packet address parity |
| **s\_network\_pkt\_data** | logic | DATA\_WIDTH | Input | Yes | Network packet data |
| **s\_network\_pkt\_type** | logic | 2 | Input | Yes | Network packet type |
| **s\_network\_pkt\_chunk\_enables** | logic | NUM\_CHUNKS | Input | Yes | Network packet chunk enables |
| **s\_network\_pkt\_eos** | logic | 1 | Input | Yes | Network packet End of Stream |
| **s\_network\_pkt\_par** | logic | 1 | Input | Yes | Network packet data parity |
| **s\_network\_pkt\_valid** | logic | 1 | Input | Yes | Network packet valid |
| **s\_network\_pkt\_ready** | logic | 1 | Output | Yes | Network packet ready |

###### Network ACK Interface (Master)

| Signal Name | Type | Width | Direction | Required | Description |
| --- | --- | --- | --- | --- | --- |
| **m\_network\_ack\_addr** | logic | ADDR\_WIDTH | Output | Yes | Network ACK address |
| **m\_network\_ack\_addr\_par** | logic | 1 | Output | Yes | Network ACK address parity |
| **m\_network\_ack\_ack** | logic | 2 | Output | Yes | Network ACK type |
| **m\_network\_ack\_par** | logic | 1 | Output | Yes | Network ACK parity |
| **m\_network\_ack\_valid** | logic | 1 | Output | Yes | Network ACK valid |
| **m\_network\_ack\_ready** | logic | 1 | Input | Yes | Network ACK ready |

###### FUB Output Interface (To Sink SRAM Control)

| Signal Name | Type | Width | Direction | Required | Description |
| --- | --- | --- | --- | --- | --- |
| **wr\_src\_valid** | logic | 1 | Output | Yes | Write output data valid |
| **wr\_src\_ready** | logic | 1 | Input | Yes | Write ready to accept output data |
| **wr\_src\_packet** | logic | DATA\_WIDTH | Output | Yes | Write output data |
| **wr\_src\_channel** | logic | CHAN\_WIDTH | Output | Yes | Write source channel |
| **wr\_src\_eos** | logic | 1 | Output | Yes | Write End of Stream |
| **wr\_src\_chunk\_enables** | logic | NUM\_CHUNKS | Output | Yes | Write chunk enable mask |

###### RDA Interfaces (Direct Bypass)

| Signal Name | Type | Width | Direction | Required | Description |
| --- | --- | --- | --- | --- | --- |
| **rda\_src\_valid** | logic | 1 | Output | Yes | RDA source packet valid |
| **rda\_src\_ready** | logic | 1 | Input | Yes | RDA source packet ready |
| **rda\_src\_packet** | logic | DATA\_WIDTH | Output | Yes | RDA source packet data |
| **rda\_src\_channel** | logic | CHAN\_WIDTH | Output | Yes | RDA source channel |
| **rda\_src\_eos** | logic | 1 | Output | Yes | RDA source End of Stream |
| **rda\_snk\_valid** | logic | 1 | Output | Yes | RDA sink packet valid |
| **rda\_snk\_ready** | logic | 1 | Input | Yes | RDA sink packet ready |
| **rda\_snk\_packet** | logic | DATA\_WIDTH | Output | Yes | RDA sink packet data |
| **rda\_snk\_channel** | logic | CHAN\_WIDTH | Output | Yes | RDA sink channel |
| **rda\_snk\_eos** | logic | 1 | Output | Yes | RDA sink End of Stream |

###### Data Consumption Interface

| Signal Name | Type | Width | Direction | Required | Description |
| --- | --- | --- | --- | --- | --- |
| **data\_consumed\_valid** | logic | 1 | Input | Yes | Data consumption notification valid |
| **data\_consumed\_ready** | logic | 1 | Output | Yes | Data consumption notification ready |
| **data\_consumed\_channel** | logic | CHAN\_WIDTH | Input | Yes | Channel that consumed data |

###### Status and Error Reporting

| Signal Name | Type | Width | Direction | Required | Description |
| --- | --- | --- | --- | --- | --- |
| **channel\_eos\_pending** | logic | NUM\_CHANNELS | Output | Yes | EOS pending per channel |
| **error\_header\_parity** | logic | 1 | Output | Yes | Header parity error |
| **error\_body\_parity** | logic | 1 | Output | Yes | Body parity error |
| **error\_buffer\_overflow** | logic | 1 | Output | Yes | Buffer overflow error |
| **error\_ack\_lost** | logic | 1 | Output | Yes | ACK lost error |
| **error\_channel\_id** | logic | CHAN\_WIDTH | Output | Yes | Channel with error |
| **packet\_count** | logic | 32 | Output | Yes | Total packet count |
| **error\_count** | logic | 16 | Output | Yes | Total error count |

###### Monitor Bus Interface

| Signal Name | Type | Width | Direction | Required | Description |
| --- | --- | --- | --- | --- | --- |
| **mon\_valid** | logic | 1 | Output | Yes | Monitor packet valid |
| **mon\_ready** | logic | 1 | Input | Yes | Monitor ready to accept packet |
| **mon\_packet** | logic | 64 | Output | Yes | Monitor packet data |

##### Network Slave ACK FSM

The Network Slave implements a sophisticated six-state ACK generation finite state machine that ensures bulletproof acknowledgment handling with zero ACK loss guarantees through dual FIFO-based queuing architecture. The FSM manages priority-based ACK arbitration between packet acknowledgments (higher priority for immediate response) and credit acknowledgments (lower priority for flow control), preventing priority inversion and ACK loss under network congestion conditions.

|  |
| --- |
| Network Slave FSM |

Network Slave FSM

**Key States:** - **ACK\_IDLE**: Ready for new ACK generation requests with priority evaluation - **ACK\_PACKET\_PENDING**: Packet ACK queued in 4-entry high-priority FIFO - **ACK\_PACKET\_ACTIVE**: Transmitting packet ACK with immediate network response - **ACK\_CREDIT\_PENDING**: Credit ACK queued in 8-entry lower-priority FIFO  
- **ACK\_CREDIT\_ACTIVE**: Transmitting credit ACK for flow control coordination - **ACK\_ERROR**: Error handling with comprehensive recovery and isolation capabilities

The FSM coordinates with comprehensive packet validation including multi-layer parity checking, protocol compliance verification, and intelligent packet classification for automatic routing to FUB (FC/TS/RAW packets) or RDA bypass interfaces. Stream boundary processing with complete EOS lifecycle tracking ensures proper completion signaling, while the dual FIFO architecture mathematically guarantees zero ACK loss even under sustained network congestion scenarios.

##### Pipeline Architecture

###### Four-Stage Processing Pipeline

Network Network -> Input Validation -> Packet Classification -> Output Routing -> ACK Generation  
 ↓ ↓ ↓ ↓ ↓  
 Parity Check Protocol Check FC/TS/RAW/RDA FUB/RDA FIFO-based  
 Format Valid Address Valid Type Detection Interfaces ACK System

###### Packet Classification Logic

// Packet type detection from Network data  
assign w\_is\_fc\_packet = (pkt\_type == 2'b00); // Flow Control packets  
assign w\_is\_ts\_packet = (pkt\_type == 2'b01); // Time Stamp packets   
assign w\_is\_rv\_packet = (pkt\_type == 2'b10); // ReserVed packets  
assign w\_is\_raw\_packet = (pkt\_type == 2'b11); // Raw data packets  
  
// RDA packet detection from address field  
assign w\_is\_rda\_packet = (pkt\_addr[63:60] == 4'hF); // RDA packets use high address bits  
assign w\_rda\_is\_read = pkt\_addr[59]; // Read/Write direction bit

###### Intelligent Packet Routing

The module automatically routes packets based on type and destination:

1. **FC/TS/RAW Packets** -> FUB interface -> Sink SRAM Control
2. **RDA Read Packets** -> RDA Source interface -> Descriptor Engine
3. **RDA Write Packets** -> RDA Sink interface -> Descriptor Engine

##### ACK Generation System

###### FIFO-Based ACK Architecture

The module implements a sophisticated ACK system with dual FIFO queues:

// ACK FIFO structure  
typedef struct packed {  
 logic [ADDR\_WIDTH-1:0] addr;  
 logic addr\_par;  
 logic [1:0] ack\_type;  
 logic par;  
 logic [CHAN\_WIDTH-1:0] channel;  
 logic [31:0] timestamp;  
} ack\_request\_t;  
  
// Dual FIFO system  
// - Packet ACK FIFO: 4 entries (higher priority)  
// - Credit ACK FIFO: 8 entries (lower priority)

###### ACK Priority System

1. **Packet ACKs**: Higher priority (immediate response to received packets)
2. **Credit ACKs**: Lower priority (flow control and stream boundary notifications)
3. **FIFO Queuing**: Prevents ACK loss under network congestion
4. **State Machine**: Proper arbitration without priority inversion

###### ACK Types

* **SIMPLE\_ACK (2’b00)**: Basic packet acknowledgment
* **START\_ACK (2’b01)**: Start of stream acknowledgment
* **CREDIT\_ACK (2’b10)**: Credit return acknowledgment
* **STOP\_AT\_EOS\_ACK (2’b11)**: Stop at EOS acknowledgment

##### Validation Pipeline

###### Multi-Layer Validation

1. **Protocol Validation**: Verify Network protocol compliance
2. **Parity Validation**: Check address and data parity
3. **Format Validation**: Verify packet structure and fields
4. **Channel Validation**: Verify target channel is valid
5. **Buffer Validation**: Check buffer availability before acceptance

###### Error Detection and Isolation

// Comprehensive error detection  
assign w\_protocol\_error = w\_invalid\_packet\_type ||   
 w\_invalid\_channel ||  
 w\_address\_parity\_error ||  
 w\_data\_parity\_error;  
  
// Error isolation per channel  
always\_ff @(posedge clk) begin  
 if (w\_protocol\_error) begin  
 error\_channel\_id <= w\_in\_channel;  
 error\_count <= error\_count + 1;  
 end  
end

##### Flow Control and Backpressure

###### Credit-Based Flow Control

The module implements comprehensive flow control:

1. **Buffer Status Monitoring**: Track buffer utilization per channel
2. **Backpressure Generation**: Assert ready signals based on buffer availability
3. **Credit Return**: Notify network of consumed data for flow control
4. **Overflow Prevention**: Prevent buffer overflow through early backpressure

###### Stream Boundary Management

EOS (End of Stream) boundaries receive special handling:

// EOS tracking per channel  
always\_ff @(posedge clk) begin  
 if (!rst\_n) begin  
 channel\_eos\_pending <= '0;  
 end else begin  
 // Set EOS pending when EOS packet accepted  
 if (packet\_accepted && w\_input\_packet.eos) begin  
 channel\_eos\_pending[w\_input\_packet.chan] <= 1'b1;  
 end  
 // Clear EOS pending when consumption notified  
 if (data\_consumed\_valid && data\_consumed\_ready) begin  
 channel\_eos\_pending[data\_consumed\_channel] <= 1'b0;  
 end  
 end  
end

##### Network 2.0 Support

The Network slave fully supports the Network 2.0 protocol specification, using chunk enables instead of the older start/len approach for indicating valid data chunks within each 512-bit packet. This provides more flexible and precise control over partial data transfers.

###### Chunk Enable Processing

// Extract chunk enables from Network 2.0 packet data  
assign w\_chunk\_enables = s\_network\_pkt\_chunk\_enables;  
  
// Forward chunk enables to appropriate interface  
assign wr\_src\_chunk\_enables = w\_chunk\_enables; // To SRAM  
assign rda\_src\_chunk\_enables = w\_chunk\_enables; // To RDA (if needed)

##### Monitor Bus Events

The module generates comprehensive monitor events for system visibility:

###### Error Events

* **Parity Error**: Address or data parity mismatch
* **Protocol Error**: Invalid packet format or type
* **Buffer Overflow**: Channel buffer capacity exceeded
* **ACK Lost**: ACK generation or transmission failure

###### Performance Events

* **Packet Reception**: Successful packet acceptance
* **Packet Routing**: Successful packet classification and routing
* **ACK Generation**: Successful ACK generation and transmission
* **Credit Update**: Flow control credit return

###### Completion Events

* **Stream Boundary**: EOS packet processing complete
* **Buffer Operation**: Buffer read/write operations
* **Error Recovery**: Error condition resolution

##### Performance Characteristics

###### Throughput Analysis

* **Peak Bandwidth**: 512 bits x 1 GHz = 512 Gbps per channel
* **Sustained Rate**: 100% pipeline utilization with deep buffering
* **Multi-Channel**: Up to 32 channels with independent processing
* **Efficiency**: Deep skid buffers enable sustained operation under congestion

###### Latency Characteristics

* **Validation Latency**: <2 cycles for comprehensive validation
* **Buffer Traversal**: 8-cycle maximum skid buffer latency
* **ACK Generation**: <5 cycles from packet acceptance to ACK transmission
* **Error Detection**: <1 cycle for all validation errors

###### Reliability Metrics

* **Packet Loss**: 0% guaranteed (mathematically proven)
* **ACK Loss**: 0% guaranteed (FIFO-based queuing)
* **Error Detection**: 100% (comprehensive validation)
* **Recovery Time**: <10 cycles for error isolation and recovery

##### Usage Guidelines

###### Performance Optimization

* Configure buffer depths based on expected packet burst sizes
* Monitor channel utilization and error rates
* Adjust timeout values based on network latency
* Use monitor events for performance analysis and debugging

###### Error Handling

The module provides comprehensive error reporting: - Monitor parity errors for data integrity issues - Check protocol errors for network compliance - Verify ACK generation and delivery - Track per-channel error statistics for fault isolation

###### Flow Control Coordination

Proper flow control requires: 1. Monitor buffer status and backpressure signals 2. Coordinate with upstream traffic sources 3. Handle data consumption notifications correctly 4. Maintain proper ACK response timing #### Sink SRAM Control

##### Overview

The Sink SRAM Control provides sophisticated buffering and flow control for incoming data streams from the AXIS slave interface. The module implements single-writer architecture with multi-channel read capabilities, comprehensive stream boundary management, and precise byte strobe forwarding for optimal AXI write performance.

##### Key Features

* **Single Write Interface**: Simplified architecture with AXIS Slave as sole writer
* **Multi-Channel Read**: Parallel read interfaces for maximum AXI engine throughput
* **Stream Boundary Management**: Complete TLAST lifecycle tracking and completion signaling
* **Byte Strobe Forwarding**: Precise TSTRB storage and forwarding for AXI write strobes
* **SRAM Storage**: 530-bit entries with complete packet metadata
* **Buffer Flow Control**: Stream-aware backpressure and overflow prevention
* **Standard AXIS Backpressure**: FUB interface uses tvalid/tready for flow control
* **Monitor Integration**: Rich monitor events for system visibility

##### Interface Specification

###### Configuration Parameters

| Parameter | Default Value | Description |
| --- | --- | --- |
| CHANNELS | 32 | Number of virtual channels |
| LINES\_PER\_CHANNEL | 256 | SRAM depth per channel |
| DATA\_WIDTH | 512 | Data width in bits |
| PTR\_BITS | $clog2(LINES\_PER\_CHANNEL) + 1 | Pointer width (+1 for wrap bit) |
| CHAN\_BITS | $clog2(CHANNELS) | Channel address width |
| COUNT\_BITS | $clog2(LINES\_PER\_CHANNEL) | Counter width |
| NUM\_CHUNKS | 16 | Number of 32-bit chunks (512/32) |
| OVERFLOW\_MARGIN | 8 | Safety margin for overflow prevention |
| USED\_THRESHOLD | 4 | Minimum entries for read operation |

###### Clock and Reset Signals

| Signal Name | Type | Width | Direction | Required | Description |
| --- | --- | --- | --- | --- | --- |
| **clk** | logic | 1 | Input | Yes | System clock |
| **rst\_n** | logic | 1 | Input | Yes | Active-low asynchronous reset |

###### Write Interface (From AXIS Slave FUB)

| Signal Name | Type | Width | Direction | Required | Description |
| --- | --- | --- | --- | --- | --- |
| **fub\_axis\_tvalid** | logic | 1 | Input | Yes | FUB write request valid |
| **fub\_axis\_tready** | logic | 1 | Output | Yes | FUB write request ready |
| **fub\_axis\_tdata** | logic | DATA\_WIDTH | Input | Yes | FUB write data |
| **fub\_axis\_tstrb** | logic | DATA\_WIDTH/8 | Input | Yes | FUB byte strobes (write enables) |
| **fub\_axis\_tlast** | logic | 1 | Input | Yes | FUB last beat (end of packet) |
| **fub\_axis\_tuser** | logic | 16 | Input | Yes | FUB metadata (channel, type) |

###### Multi-Channel Read Interface (To AXI Engines)

| Signal Name | Type | Width | Direction | Required | Description |
| --- | --- | --- | --- | --- | --- |
| **rd\_valid** | logic | CHANNELS | Output | Yes | Read data valid per channel |
| **rd\_ready** | logic | CHANNELS | Input | Yes | Read data ready per channel |
| **rd\_data** | logic | DATA\_WIDTH x CHANNELS | Output | Yes | Read data per channel |
| **rd\_type** | logic | 2 x CHANNELS | Output | Yes | Packet type per channel |
| **rd\_eos** | logic | CHANNELS | Output | Yes | End of Stream per channel (from TLAST) |
| **rd\_tstrb** | logic | (DATA\_WIDTH/8) x CHANNELS | Output | Yes | Byte strobes per channel |
| **rd\_used\_count** | logic | 8 x CHANNELS | Output | Yes | Used entries per channel |
| **rd\_lines\_for\_transfer** | logic | 8 x CHANNELS | Output | Yes | Lines available for transfer per channel |

###### Data Consumption Notification

| Signal Name | Type | Width | Direction | Required | Description |
| --- | --- | --- | --- | --- | --- |
| **data\_consumed\_valid** | logic | 1 | Output | Yes | Consumption notification valid |
| **data\_consumed\_ready** | logic | 1 | Input | Yes | Consumption notification ready |
| **data\_consumed\_channel** | logic | CHAN\_BITS | Output | Yes | Channel that consumed data |

###### EOS Completion Interface

| Signal Name | Type | Width | Direction | Required | Description |
| --- | --- | --- | --- | --- | --- |
| **eos\_completion\_valid** | logic | 1 | Output | Yes | EOS completion notification valid |
| **eos\_completion\_ready** | logic | 1 | Input | Yes | EOS completion notification ready |
| **eos\_completion\_channel** | logic | CHAN\_BITS | Output | Yes | Channel with EOS completion |

###### Control and Status

| Signal Name | Type | Width | Direction | Required | Description |
| --- | --- | --- | --- | --- | --- |
| **drain\_enable** | logic | 1 | Input | Yes | Enable buffer draining mode |
| **channel\_full** | logic | CHANNELS | Output | Yes | Per-channel full status |
| **channel\_overflow** | logic | CHANNELS | Output | Yes | Per-channel overflow status |
| **backpressure\_warning** | logic | CHANNELS | Output | Yes | Per-channel backpressure warning |
| **eos\_pending** | logic | CHANNELS | Output | Yes | EOS pending per channel |

###### Monitor Bus Interface

| Signal Name | Type | Width | Direction | Required | Description |
| --- | --- | --- | --- | --- | --- |
| **mon\_valid** | logic | 1 | Output | Yes | Monitor packet valid |
| **mon\_ready** | logic | 1 | Input | Yes | Monitor ready to accept packet |
| **mon\_packet** | logic | 64 | Output | Yes | Monitor packet data |

##### SRAM Architecture

##### Sink SRAM Control FSM

The Sink SRAM Control operates through a sophisticated flow control and arbitration state machine that manages multi-channel buffer operations with stream-aware priority scheduling and comprehensive boundary processing. The FSM coordinates single-writer operations from the AXIS slave FUB interface with multi-channel read arbitration for AXI write engines, implementing priority-based scheduling that favors channels with pending stream boundaries over threshold-based normal operations.

|  |
| --- |
| Sink SRAM FSM |

Sink SRAM FSM

**Key Operations:** - **Write State Management**: Single-writer flow control with overflow prevention and metadata embedding for 530-bit SRAM entries - **Read Arbitration**: Multi-level priority arbitration favoring EOS/EOL/EOD pending channels, then threshold-based scheduling, then round-robin fairness - **Stream Boundary Processing**: Complete TLAST lifecycle tracking with dedicated completion signaling and consumption notification coordination - **Buffer Management**: Per-channel pointer management with wrap detection, used count tracking, and configurable threshold monitoring - **Flow Control Coordination**: Standard AXIS backpressure (fub\_axis\_tready) to upstream, overflow warning, and completion notifications

The FSM implements stream-aware buffer management where TLAST boundaries receive highest priority processing to ensure timely descriptor completion signaling, while sophisticated pointer arithmetic and buffer status tracking prevent overflow conditions and coordinate with upstream AXIS flow control. The architecture eliminates traditional multi-writer arbitration complexity through the single-writer design while maintaining optimal multi-channel read performance through priority-based scheduling algorithms.

###### Storage Format (594 bits total)

The SRAM stores complete packet metadata alongside data for precise forwarding:

// SRAM entry format:  
// {TYPE[7:0], TSTRB[63:0], DATA[511:0]} = 594 bits total  
localparam int EXTENDED\_SRAM\_WIDTH = 8 + (DATA\_WIDTH/8) + DATA\_WIDTH;  
  
// Write data composition (from AXIS FUB interface)  
assign w\_sram\_wr\_data = {  
 fub\_axis\_tuser[7:0], // Bits 593:586: Packet type from TUSER  
 fub\_axis\_tstrb, // Bits 585:522: Byte strobes (64 bits for 512-bit data)  
 fub\_axis\_tdata // Bits 521:0: Data payload  
};

###### TLAST Flow Management

**Critical Design Decision**: TLAST is NOT stored in SRAM but used for completion signaling:

1. **TLAST Detection**: AXIS packets arrive with TLAST in beat structure (fub\_axis\_tlast)
2. **TLAST Processing**: TLAST triggers descriptor completion logic (control only)
3. **TLAST Storage**: TLAST is NOT stored in SRAM - only payload data is stored
4. **TLAST Control**: TLAST used for completion signaling to scheduler (maps to internal EOS)
5. **EOS Completion**: Dedicated FIFO interface for EOS completion notifications

###### Multi-Channel Buffer Management

Each channel maintains independent: - **Write Pointer**: Binary pointer with wrap detection - **Read Pointer**: Binary pointer with wrap detection  
- **Used Count**: Number of valid entries available for reading - **Open Count**: Number of available entries for writing - **EOS Pending**: Flag indicating EOS completion pending

##### Buffer Flow Control

###### Write Acceptance Logic

// Write acceptance based on buffer availability (standard AXIS backpressure)  
logic [CHAN\_WIDTH-1:0] w\_channel;  
assign w\_channel = fub\_axis\_tuser[15:8]; // Extract channel from TUSER  
  
assign fub\_axis\_tready = !w\_channel\_full[w\_channel] &&  
 (r\_used\_count[w\_channel] < (LINES\_PER\_CHANNEL - OVERFLOW\_MARGIN));

###### Read Arbitration

The module implements sophisticated read arbitration:

// Priority levels for read arbitration  
// 1. Channels with EOS pending (highest priority)  
// 2. Channels meeting used threshold  
// 3. Round-robin fairness among eligible channels  
// 4. Drain mode considerations  
  
assign w\_eos\_priority = r\_eos\_pending;  
assign w\_threshold\_priority = (r\_used\_count[i] >= USED\_THRESHOLD);  
assign w\_drain\_priority = drain\_enable && (r\_used\_count[i] > 0);

###### Stream Boundary Processing

TLAST boundaries trigger special processing:

// TLAST completion signaling (maps to internal EOS)  
always\_ff @(posedge clk) begin  
 if (!rst\_n) begin  
 r\_eos\_pending <= '0;  
 end else begin  
 // Set EOS pending when TLAST packet written  
 if (fub\_axis\_tvalid && fub\_axis\_tready && fub\_axis\_tlast) begin  
 r\_eos\_pending[w\_channel] <= 1'b1;  
 end  
 // Clear EOS pending when completion signaled  
 if (eos\_completion\_valid && eos\_completion\_ready) begin  
 r\_eos\_pending[eos\_completion\_channel] <= 1'b0;  
 end  
 end  
end

##### Data Flow Architecture

FC/TS Packets: AXIS Slave FUB -> Sink SRAM Control -> AXI Write Engine  
 ↓ ↓ ↓  
 Single Write 594-bit Storage Multi-Channel Read  
 FUB Interface + Packet Metadata + TSTRB Forwarding  
 (AXIS-like) (TYPE + TSTRB)  
  
TLAST Flow: TLAST Detection -> Completion Signaling -> Scheduler Notification  
 ↓ ↓ ↓  
 Packet-Level Control-Only Descriptor-Level  
 TLAST in AXIS Processing Completion (EOS)

##### AXIS Integration

The sink SRAM control integrates with standard AXI-Stream protocol via the FUB interface from axis\_slave.sv:

**TSTRB (Byte Strobes):** - Standard AXIS uses 64-bit byte strobes for 512-bit data - tstrb[i] = 1 indicates byte i is valid - Stored in SRAM and forwarded to AXI write engine for precise wstrb generation - More fine-grained than legacy chunk enables (byte-level vs 32-bit chunk-level)

**TLAST (Packet Boundary):** - Standard AXIS uses TLAST to mark final beat of packet - Maps to internal EOS for completion signaling - NOT stored in SRAM (control-only signal) - Triggers descriptor completion to scheduler

##### Monitor Bus Events

The module generates comprehensive monitor events:

###### Error Events

* **Buffer Overflow**: When channel buffer exceeds capacity
* **Invalid Channel**: When write targets invalid channel
* **Pointer Corruption**: When pointer consistency checks fail

###### Completion Events

* **Write Completion**: Successful data write to buffer
* **Read Completion**: Successful data read from buffer
* **EOS Completion**: End of stream processing complete

###### Threshold Events

* **Backpressure Warning**: When buffer usage exceeds warning threshold
* **Buffer Full**: When channel buffer reaches capacity
* **Credit Exhausted**: When no buffer space available

##### Performance Characteristics

###### Throughput Metrics

* **Write Throughput**: 1 write per cycle when space available
* **Read Throughput**: 1 read per cycle per channel with proper SRAM implementation
* **Metadata Overhead**: 13.9% (82 metadata bits / 594 total bits)
* **Buffer Efficiency**: 95%+ utilization with stream-aware flow control

###### Latency Characteristics

* **Write Latency**: 1 cycle for data acceptance
* **Read Arbitration**: 1-3 cycles depending on priority and contention
* **TLAST Processing**: 1 cycle for EOS completion signaling
* **Completion Notification**: 2-4 cycles from consumption to scheduler notification

##### Implementation Notes

###### Multi-Channel Read Support

The current implementation provides a foundation for multi-channel reads but requires additional infrastructure for optimal concurrent operation:

1. **Multiple SRAM instances** (one per channel) - Highest performance
2. **Multi-port SRAM** with read arbitration - Good performance
3. **Time-multiplexed single-port** with scheduling - Acceptable performance

###### Buffer Management

Each channel operates independently with: - **Independent Pointer Management**: Separate read/write pointers per channel - **Per-Channel Flow Control**: Individual full/empty status tracking - **TLAST State Management**: Per-channel EOS pending tracking (derived from TLAST) - **Completion Coordination**: Channel-specific consumption notifications

##### Usage Guidelines

###### Performance Optimization

* Configure USED\_THRESHOLD based on AXI engine requirements
* Set OVERFLOW\_MARGIN to prevent buffer overflow under burst conditions
* Use EOS completion interface for proper stream boundary coordination (TLAST -> EOS)
* Monitor backpressure warnings to optimize buffer utilization
* Leverage standard AXIS flow control (fub\_axis\_tready) for upstream coordination

###### Error Handling

The module provides comprehensive error detection: - Monitor channel overflow conditions - Check buffer pointer consistency - Verify TLAST/EOS completion flow - Track per-channel error statistics through monitor events - Validate TSTRB patterns for data integrity #### Sink AXI Write Engine

##### Overview

The Sink AXI Write Engine provides high-performance multi-channel AXI write operations with sophisticated arbitration, alignment optimization, and transfer strategy selection. The module implements a pure pipeline architecture that eliminates FSM overhead, achieving zero-cycle arbitration to AXI address issue with perfect AXI streaming and natural backpressure handling.

##### Key Features

* **Pure Pipeline Architecture**: Zero-cycle arbitration to AXI address issue with no FSM overhead
* **Multi-Channel Arbitration**: Round-robin arbitration across up to 32 independent channels
* **Address Alignment Integration**: Uses pre-calculated alignment information from scheduler
* **Transfer Strategy Selection**: Four distinct transfer modes for optimal performance
* **Chunk-Aware Strobes**: Precise write strobes based on Network 2.0 chunk enables
* **Buffer-Aware Arbitration**: Considers SRAM buffer status for optimal throughput
* **EOS Processing**: Handles End of Stream boundaries for proper completion signaling
* **Monitor Integration**: Comprehensive event reporting for system visibility

##### Interface Specification

###### Configuration Parameters

| Parameter | Default Value | Description |
| --- | --- | --- |
| NUM\_CHANNELS | 32 | Number of virtual channels |
| CHAN\_WIDTH | $clog2(NUM\_CHANNELS) | Width of channel address fields |
| ADDR\_WIDTH | 64 | Address width for AXI transactions |
| DATA\_WIDTH | 512 | Data width for AXI and internal interfaces |
| NUM\_CHUNKS | 16 | Number of 32-bit chunks (512/32) |
| AXI\_ID\_WIDTH | 8 | AXI transaction ID width |
| MAX\_BURST\_LEN | 64 | Maximum AXI burst length |
| MAX\_OUTSTANDING | 16 | Maximum outstanding transactions |
| TIMEOUT\_CYCLES | 1000 | Timeout threshold for stuck transfers |
| ALIGNMENT\_BOUNDARY | 64 | Address alignment boundary (64 bytes) |

###### Clock and Reset Signals

| Signal Name | Type | Width | Direction | Required | Description |
| --- | --- | --- | --- | --- | --- |
| **clk** | logic | 1 | Input | Yes | System clock |
| **rst\_n** | logic | 1 | Input | Yes | Active-low asynchronous reset |

###### Multi-Channel Scheduler Interface

| Signal Name | Type | Width | Direction | Required | Description |
| --- | --- | --- | --- | --- | --- |
| **data\_valid** | logic | NUM\_CHANNELS | Input | Yes | Data transfer request per channel |
| **data\_ready** | logic | NUM\_CHANNELS | Output | Yes | Data transfer ready per channel |
| **data\_address** | logic | ADDR\_WIDTH x NUM\_CHANNELS | Input | Yes | Data address per channel |
| **data\_length** | logic | 32 x NUM\_CHANNELS | Input | Yes | Data length per channel (in 4-byte chunks) |
| **data\_type** | logic | 2 x NUM\_CHANNELS | Input | Yes | Data type per channel |
| **data\_eos** | logic | NUM\_CHANNELS | Input | Yes | End of Stream per channel |
| **data\_transfer\_length** | logic | 32 x NUM\_CHANNELS | Output | Yes | Actual transfer length per channel |
| **data\_done\_strobe** | logic | NUM\_CHANNELS | Output | Yes | Transfer completion per channel |
| **data\_error** | logic | NUM\_CHANNELS | Output | Yes | Transfer error per channel |

###### Address Alignment Bus Interface

| Signal Name | Type | Width | Direction | Required | Description |
| --- | --- | --- | --- | --- | --- |
| **data\_alignment\_info** | alignment\_info\_t | NUM\_CHANNELS | Input | Yes | Pre-calculated alignment information per channel |
| **data\_alignment\_valid** | logic | NUM\_CHANNELS | Input | Yes | Alignment information valid per channel |
| **data\_alignment\_ready** | logic | NUM\_CHANNELS | Output | Yes | Alignment information ready per channel |
| **data\_alignment\_next** | logic | NUM\_CHANNELS | Output | Yes | Request next alignment per channel |
| **data\_transfer\_phase** | transfer\_phase\_t | NUM\_CHANNELS | Input | Yes | Transfer phase per channel |
| **data\_sequence\_complete** | logic | NUM\_CHANNELS | Output | Yes | Transfer sequence complete per channel |

###### SRAM Read Interface (Multi-Channel)

| Signal Name | Type | Width | Direction | Required | Description |
| --- | --- | --- | --- | --- | --- |
| **rd\_valid** | logic | NUM\_CHANNELS | Input | Yes | Read data valid per channel |
| **rd\_ready** | logic | NUM\_CHANNELS | Output | Yes | Read data ready per channel |
| **rd\_data** | logic | DATA\_WIDTH x NUM\_CHANNELS | Input | Yes | Read data per channel |
| **rd\_type** | logic | 2 x NUM\_CHANNELS | Input | Yes | Packet type per channel |
| **rd\_chunk\_valid** | logic | NUM\_CHUNKS x NUM\_CHANNELS | Input | Yes | Chunk enables per channel |
| **rd\_used\_count** | logic | 8 x NUM\_CHANNELS | Input | Yes | Used entries per channel |
| **rd\_lines\_for\_transfer** | logic | 8 x NUM\_CHANNELS | Input | Yes | Lines available for transfer per channel |

###### AXI4 Master Write Interface

| Signal Name | Type | Width | Direction | Required | Description |
| --- | --- | --- | --- | --- | --- |
| **aw\_valid** | logic | 1 | Output | Yes | Write address valid |
| **aw\_ready** | logic | 1 | Input | Yes | Write address ready |
| **aw\_addr** | logic | ADDR\_WIDTH | Output | Yes | Write address |
| **aw\_len** | logic | 8 | Output | Yes | Burst length |
| **aw\_size** | logic | 3 | Output | Yes | Burst size |
| **aw\_burst** | logic | 2 | Output | Yes | Burst type |
| **aw\_id** | logic | AXI\_ID\_WIDTH | Output | Yes | Write ID |
| **aw\_lock** | logic | 1 | Output | Yes | Lock type |
| **aw\_cache** | logic | 4 | Output | Yes | Cache attributes |
| **aw\_prot** | logic | 3 | Output | Yes | Protection attributes |
| **aw\_qos** | logic | 4 | Output | Yes | QoS identifier |
| **aw\_region** | logic | 4 | Output | Yes | Region identifier |
| **w\_valid** | logic | 1 | Output | Yes | Write data valid |
| **w\_ready** | logic | 1 | Input | Yes | Write data ready |
| **w\_data** | logic | DATA\_WIDTH | Output | Yes | Write data |
| **w\_strb** | logic | DATA\_WIDTH/8 | Output | Yes | Write strobes |
| **w\_last** | logic | 1 | Output | Yes | Write last |
| **b\_valid** | logic | 1 | Input | Yes | Write response valid |
| **b\_ready** | logic | 1 | Output | Yes | Write response ready |
| **b\_resp** | logic | 2 | Input | Yes | Write response |
| **b\_id** | logic | AXI\_ID\_WIDTH | Input | Yes | Write ID |

###### Configuration Interface

| Signal Name | Type | Width | Direction | Required | Description |
| --- | --- | --- | --- | --- | --- |
| **cfg\_transfer\_beats** | logic | 8 | Input | Yes | Default transfer beats |
| **cfg\_enable\_variable\_beats** | logic | 1 | Input | Yes | Enable variable beat transfers |
| **cfg\_force\_single\_beat** | logic | 1 | Input | Yes | Force single beat mode |
| **cfg\_timeout\_enable** | logic | 1 | Input | Yes | Enable timeout detection |

###### Status Interface

| Signal Name | Type | Width | Direction | Required | Description |
| --- | --- | --- | --- | --- | --- |
| **engine\_idle** | logic | 1 | Output | Yes | Engine idle status |
| **engine\_busy** | logic | 1 | Output | Yes | Engine busy status |
| **outstanding\_count** | logic | 16 | Output | Yes | Outstanding transaction count |

###### Monitor Bus Interface

| Signal Name | Type | Width | Direction | Required | Description |
| --- | --- | --- | --- | --- | --- |
| **mon\_valid** | logic | 1 | Output | Yes | Monitor packet valid |
| **mon\_ready** | logic | 1 | Input | Yes | Monitor ready to accept packet |
| **mon\_packet** | logic | 64 | Output | Yes | Monitor packet data |

##### Pure Pipeline Architecture

###### Zero-Cycle Arbitration

The engine implements a revolutionary pure pipeline architecture:

Channel Arbitration -> Immediate AXI Address -> Pipelined Data Flow -> Natural Backpressure  
 ↓ ↓ ↓ ↓  
 Round-Robin Zero-Cycle Issue Perfect AXI Streaming Backpressure Flow  
 Fair Selection No FSM Overhead Optimal Bandwidth Clean Interface

###### Pipeline Stages

// Pure combinational arbitration to AXI address issue  
logic [NUM\_CHANNELS-1:0] w\_channel\_request\_mask;  
logic w\_grant\_valid;  
logic [CHAN\_WIDTH-1:0] w\_grant\_id;  
  
// Request mask: channels with data, alignment, and SRAM data available  
assign w\_channel\_request\_mask = data\_valid & data\_alignment\_valid & rd\_valid;  
  
// Zero-cycle arbitration to AXI address  
assign aw\_valid = w\_grant\_valid;  
assign aw\_addr = data\_address[w\_grant\_id];  
assign aw\_id = {{(AXI\_ID\_WIDTH-CHAN\_WIDTH){1'b0}}, w\_grant\_id};

##### Multi-Channel Arbitration

###### Round-Robin Fairness

// Round-robin arbiter in grant-ack mode  
arbiter\_round\_robin #(  
 .CLIENTS(NUM\_CHANNELS),  
 .WAIT\_GNT\_ACK(1) // Hold grant until sequence complete  
) u\_channel\_arbiter (  
 .clk(clk),  
 .rst\_n(rst\_n),  
 .request(w\_channel\_request\_mask),  
 .grant\_ack(w\_grant\_ack),  
 .grant\_valid(w\_grant\_valid),  
 .grant(w\_grant),  
 .grant\_id(w\_grant\_id)  
);  
  
// Release grant when sequence completes  
assign w\_grant\_ack[i] = w\_grant[i] && data\_sequence\_complete[i] &&   
 w\_address\_accepted && w\_data\_complete;

###### Request Generation

Channels are eligible for arbitration when: 1. **Scheduler Request**: data\_valid[i] asserted with valid transfer request 2. **Alignment Ready**: data\_alignment\_valid[i] with pre-calculated parameters 3. **SRAM Data**: rd\_valid[i] with buffered data available 4. **Buffer Status**: rd\_lines\_for\_transfer[i] > 0 indicating sufficient buffering

##### Transfer Strategy Selection

###### Four Transfer Strategies

1. **Precision Strategy**: Exact chunk-level transfers for small data
2. **Aligned Strategy**: Optimal alignment to 64-byte boundaries
3. **Forced Strategy**: User-configured fixed burst patterns
4. **Single Strategy**: Single-beat transfers for minimal latency

###### Strategy Selection Logic

// Transfer strategy from alignment information  
case (data\_alignment\_info[channel].alignment\_strategy)  
 STRATEGY\_PRECISION: begin  
 aw\_len <= 8'h00; // Single beat  
 w\_strb <= precision\_strobe\_pattern;  
 end  
 STRATEGY\_ALIGNED: begin  
 aw\_len <= data\_alignment\_info[channel].optimal\_burst\_len - 1;  
 w\_strb <= alignment\_strobe\_pattern;  
 end  
 STRATEGY\_STREAMING: begin  
 aw\_len <= cfg\_transfer\_beats - 1;  
 w\_strb <= {(DATA\_WIDTH/8){1'b1}}; // All bytes  
 end  
 STRATEGY\_SINGLE: begin  
 aw\_len <= 8'h00; // Single beat  
 w\_strb <= single\_beat\_pattern;  
 end  
endcase

##### Address Alignment Integration

###### Pre-Calculated Optimization

The engine leverages the scheduler’s address alignment FSM:

// Use pre-calculated alignment information  
if (data\_alignment\_valid[channel]) begin  
 case (data\_transfer\_phase[channel])  
 PHASE\_ALIGNMENT: begin  
 aw\_addr <= data\_alignment\_info[channel].aligned\_addr;  
 aw\_len <= data\_alignment\_info[channel].first\_burst\_len;  
 chunk\_enables <= data\_alignment\_info[channel].first\_chunk\_enables;  
 end  
 PHASE\_STREAMING: begin  
 aw\_addr <= streaming\_addr;  
 aw\_len <= data\_alignment\_info[channel].optimal\_burst\_len;  
 chunk\_enables <= 16'hFFFF; // All chunks valid  
 end  
 PHASE\_FINAL: begin  
 aw\_addr <= final\_addr;  
 aw\_len <= data\_alignment\_info[channel].final\_burst\_len;  
 chunk\_enables <= data\_alignment\_info[channel].final\_chunk\_enables;  
 end  
 endcase  
end

###### Performance Benefits

1. **No Alignment Overhead**: Zero calculation time in critical AXI path
2. **Optimal Burst Planning**: Pre-calculated burst lengths and patterns
3. **Precise Chunk Strobes**: Pre-calculated strobe patterns from chunk enables
4. **Transfer Sequence Coordination**: Complete sequence planned in advance

##### Chunk-Aware Write Strobes

###### Network 2.0 Chunk Processing

// Generate AXI write strobes from Network 2.0 chunk enables  
function logic [DATA\_WIDTH/8-1:0] generate\_write\_strobes(  
 input logic [NUM\_CHUNKS-1:0] chunk\_enables,  
 input logic [5:0] addr\_offset  
);  
 logic [DATA\_WIDTH/8-1:0] strobe\_mask;  
   
 // Convert chunk enables to byte strobes  
 for (int i = 0; i < NUM\_CHUNKS; i++) begin  
 if (chunk\_enables[i]) begin  
 strobe\_mask[i\*4 +: 4] = 4'hF; // 4 bytes per chunk  
 end else begin  
 strobe\_mask[i\*4 +: 4] = 4'h0;  
 end  
 end  
   
 // Apply address offset alignment  
 return strobe\_mask >> addr\_offset;  
endfunction

###### Precision Write Control

The engine provides precise write control: - **Byte-Level Precision**: Accurate strobes based on chunk enables - **Alignment Handling**: Proper strobe shifting for unaligned addresses - **Partial Transfer Support**: Handles final partial transfers correctly - **Memory Efficiency**: Only writes valid data bytes

##### Buffer-Aware Arbitration

###### SRAM Buffer Coordination

// Enhanced arbitration with buffer awareness  
logic [NUM\_CHANNELS-1:0] w\_buffer\_ready;  
  
// Check buffer status for arbitration eligibility  
generate  
 for (genvar i = 0; i < NUM\_CHANNELS; i++) begin : gen\_buffer\_check  
 assign w\_buffer\_ready[i] = (rd\_used\_count[i] >= USED\_THRESHOLD) &&  
 (rd\_lines\_for\_transfer[i] > 0);  
 end  
endgenerate  
  
// Combined request mask includes buffer readiness  
assign w\_channel\_request\_mask = data\_valid & data\_alignment\_valid &   
 rd\_valid & w\_buffer\_ready;

###### Flow Control Benefits

1. **Prevents Starvation**: Ensures sufficient buffered data before arbitration
2. **Optimizes Bursts**: Waits for optimal buffer fill levels
3. **Reduces Latency**: Prevents pipeline stalls from insufficient data
4. **Improves Efficiency**: Maximizes AXI burst utilization

##### EOS Processing

###### Stream Boundary Handling

// EOS detection and completion signaling  
logic [NUM\_CHANNELS-1:0] w\_eos\_detected;  
logic [NUM\_CHANNELS-1:0] w\_sequence\_complete;  
  
assign w\_eos\_detected = data\_eos & data\_valid;  
  
// Sequence completion triggers for grant release  
assign data\_sequence\_complete = w\_sequence\_complete | w\_eos\_detected;  
  
// EOS completion notification  
always\_ff @(posedge clk) begin  
 if (!rst\_n) begin  
 data\_done\_strobe <= '0;  
 end else begin  
 data\_done\_strobe <= w\_eos\_detected & w\_grant\_valid;  
 end  
end

##### Performance Characteristics

###### Throughput Analysis

* **Peak Bandwidth**: 512 bits x 1 GHz = 512 Gbps theoretical maximum
* **Sustained Rate**: >95% efficiency with proper buffer management
* **Multi-Channel**: Full bandwidth utilization across active channels
* **Zero-Cycle Arbitration**: No arbitration overhead in critical path

###### Latency Characteristics

* **Arbitration Latency**: 0 cycles (pure combinational)
* **Address Issue**: Immediate upon grant
* **Data Pipeline**: 1-2 cycles through SRAM to AXI
* **Completion Notification**: <3 cycles from last data

###### Efficiency Metrics

* **AXI Utilization**: >98% with optimal burst patterns
* **Buffer Efficiency**: >95% with buffer-aware arbitration
* **Channel Fairness**: Perfect round-robin fairness
* **Error Rate**: <0.01% under normal operating conditions

##### Monitor Bus Events

The sink AXI write engine generates comprehensive monitor events:

###### Performance Events

* **Channel Arbitration**: Channel selection and grant timing
* **Transfer Start**: AXI transaction initiation
* **Transfer Complete**: AXI transaction completion
* **Burst Efficiency**: Burst length and utilization metrics
* **Buffer Utilization**: SRAM buffer usage per channel

###### Error Events

* **AXI Error**: AXI write response error conditions
* **Timeout**: Transaction timeout detection
* **Buffer Underrun**: Insufficient SRAM data during transfer
* **Alignment Error**: Address alignment validation failure

###### Completion Events

* **Sequence Complete**: Transfer sequence completion per channel
* **EOS Processing**: End of Stream boundary processing
* **Channel Done**: Channel transfer completion notification
* **Pipeline Flush**: Pipeline cleanup operations

##### Usage Guidelines

###### Performance Optimization

* Use alignment bus for optimal AXI transfer planning
* Configure buffer thresholds for optimal arbitration efficiency
* Monitor channel utilization and adjust arbitration priorities
* Use transfer strategies appropriate for data patterns

###### Buffer Management

Optimal buffer management requires: 1. Configure USED\_THRESHOLD based on burst requirements 2. Monitor rd\_lines\_for\_transfer for arbitration efficiency 3. Coordinate with SRAM control for optimal buffer utilization 4. Balance buffer depth vs. latency requirements

###### Error Handling

The engine provides comprehensive error detection: - Monitor AXI response codes for memory subsystem health - Track timeout conditions for performance analysis - Verify buffer availability before transfer initiation - Use monitor events for debugging and optimization

### Source Data Path

#### Source Data Path

##### Overview

The Source Data Path provides a complete integrated data transmission pipeline that combines multi-channel scheduler interfaces, AXI memory read operations, SRAM buffering, AXIS packet transmission, and comprehensive monitor bus aggregation. This wrapper manages the complete data flow from scheduler requests through final AXIS stream transmission.

**RTL Module:** rtl/amba/axis/axis\_master.sv (AXIS interface) + RAPIDS source components

The wrapper implements sophisticated address alignment processing and stream boundary management to ensure optimal AXI read performance and reliable AXIS packet delivery with zero packet loss guarantees.

|  |
| --- |
| source data path |

source data path

##### Key Features

* **Complete Data Transmission Pipeline**: From scheduler requests to AXIS packet transmission
* **Standard AXIS Interface**: Industry-standard AXI-Stream protocol (no custom credits)
* **Multi-Channel Scheduler Interface**: Enhanced interface with address alignment bus support
* **AXI Read Engine**: Optimized multi-channel AXI read operations with tracking
* **SRAM Buffering**: Multi-channel buffering with preallocation and flow control
* **AXIS Master**: Multi-channel arbitration with standard AXIS backpressure
* **Monitor Bus Aggregation**: Unified monitoring from AXI engine, SRAM control, and AXIS master
* **Address Alignment Integration**: Pre-calculated alignment information for optimal performance

##### Interface Specification

###### Clock and Reset

| Signal Name | Type | Width | Direction | Required | Description |
| --- | --- | --- | --- | --- | --- |
| **clk** | logic | 1 | Input | Yes | System clock |
| **rst\_n** | logic | 1 | Input | Yes | Active-low asynchronous reset |

###### Configuration Interface

| Signal Name | Type | Width | Direction | Required | Description |
| --- | --- | --- | --- | --- | --- |
| **cfg\_transfer\_size** | logic | 2 | Input | Yes | Transfer size configuration (00=1Beat, 01=1KB, 10=2KB, 11=4KB) |
| **cfg\_streaming\_enable** | logic | 1 | Input | Yes | Streaming mode enable |
| **cfg\_sram\_enable** | logic | 1 | Input | Yes | SRAM buffering enable |
| **cfg\_channel\_enable** | logic | NUM\_CHANNELS | Input | Yes | Per-channel enable control |

###### Multi-Channel Scheduler Interface

| Signal Name | Type | Width | Direction | Required | Description |
| --- | --- | --- | --- | --- | --- |
| **data\_valid** | logic | NUM\_CHANNELS | Input | Yes | Data transfer request per channel |
| **data\_ready** | logic | NUM\_CHANNELS | Output | Yes | Data transfer ready per channel |
| **data\_address** | logic | ADDR\_WIDTH x NUM\_CHANNELS | Input | Yes | Data address per channel |
| **data\_length** | logic | 32 x NUM\_CHANNELS | Input | Yes | Data length per channel |
| **data\_type** | logic | 2 x NUM\_CHANNELS | Input | Yes | Data type per channel |
| **data\_eos** | logic | NUM\_CHANNELS | Input | Yes | End of Stream per channel |
| **data\_transfer\_length** | logic | 32 x NUM\_CHANNELS | Output | Yes | Actual transfer length per channel |
| **data\_done\_strobe** | logic | NUM\_CHANNELS | Output | Yes | Transfer completion per channel |
| **data\_error** | logic | NUM\_CHANNELS | Output | Yes | Transfer error per channel |

###### Address Alignment Bus Interface

| Signal Name | Type | Width | Direction | Required | Description |
| --- | --- | --- | --- | --- | --- |
| **data\_alignment\_info** | alignment\_info\_t | NUM\_CHANNELS | Input | Yes | Pre-calculated alignment information per channel |
| **data\_alignment\_valid** | logic | NUM\_CHANNELS | Input | Yes | Alignment information valid per channel |
| **data\_alignment\_ready** | logic | NUM\_CHANNELS | Output | Yes | Alignment information ready per channel |
| **data\_alignment\_next** | logic | NUM\_CHANNELS | Output | Yes | Request next alignment per channel |
| **data\_transfer\_phase** | transfer\_phase\_t | NUM\_CHANNELS | Input | Yes | Current transfer phase per channel |
| **data\_sequence\_complete** | logic | NUM\_CHANNELS | Output | Yes | Transfer sequence complete per channel |

###### AXI4 Master Read Interface

| Signal Name | Type | Width | Direction | Required | Description |
| --- | --- | --- | --- | --- | --- |
| **ar\_valid** | logic | 1 | Output | Yes | Read address valid |
| **ar\_ready** | logic | 1 | Input | Yes | Read address ready |
| **ar\_addr** | logic | ADDR\_WIDTH | Output | Yes | Read address |
| **ar\_len** | logic | 8 | Output | Yes | Burst length |
| **ar\_size** | logic | 3 | Output | Yes | Burst size |
| **ar\_burst** | logic | 2 | Output | Yes | Burst type |
| **ar\_id** | logic | AXI\_ID\_WIDTH | Output | Yes | Read ID |
| **ar\_lock** | logic | 1 | Output | Yes | Lock type |
| **ar\_cache** | logic | 4 | Output | Yes | Cache type |
| **ar\_prot** | logic | 3 | Output | Yes | Protection type |
| **ar\_qos** | logic | 4 | Output | Yes | QoS identifier |
| **ar\_region** | logic | 4 | Output | Yes | Region identifier |
| **r\_valid** | logic | 1 | Input | Yes | Read data valid |
| **r\_ready** | logic | 1 | Output | Yes | Read data ready |
| **r\_data** | logic | DATA\_WIDTH | Input | Yes | Read data |
| **r\_resp** | logic | 2 | Input | Yes | Read response |
| **r\_last** | logic | 1 | Input | Yes | Read last |
| **r\_id** | logic | AXI\_ID\_WIDTH | Input | Yes | Read ID |

###### AXI-Stream Master Interface (TX)

| Signal Name | Type | Width | Direction | Required | Description |
| --- | --- | --- | --- | --- | --- |
| **axis\_src\_tx\_tdata** | logic | DATA\_WIDTH | Output | Yes | Stream data payload |
| **axis\_src\_tx\_tstrb** | logic | DATA\_WIDTH/8 | Output | Yes | Byte strobes (write enables) |
| **axis\_src\_tx\_tlast** | logic | 1 | Output | Yes | Last transfer in packet |
| **axis\_src\_tx\_tvalid** | logic | 1 | Output | Yes | Stream data valid |
| **axis\_src\_tx\_tready** | logic | 1 | Input | Yes | Stream ready (backpressure) |
| **axis\_src\_tx\_tuser** | logic | 16 | Output | Yes | User sideband (packet metadata) |

**TUSER Encoding (Source TX):**

[15:8] - Channel ID  
[7:0] - Packet type/flags

**Note:** AXIS uses standard tvalid/tready backpressure. No credit channels or custom flow control mechanisms.

###### Monitor Bus Interface

| Signal Name | Type | Width | Direction | Required | Description |
| --- | --- | --- | --- | --- | --- |
| **mon\_valid** | logic | 1 | Output | Yes | Monitor packet valid |
| **mon\_ready** | logic | 1 | Input | Yes | Monitor ready to accept packet |
| **mon\_packet** | logic | 64 | Output | Yes | Monitor packet data |

###### Status and Control

| Signal Name | Type | Width | Direction | Required | Description |
| --- | --- | --- | --- | --- | --- |
| **engine\_idle** | logic | NUM\_CHANNELS | Output | Yes | Engine idle status per channel |
| **engine\_busy** | logic | NUM\_CHANNELS | Output | Yes | Engine busy status per channel |
| **sram\_space\_available** | logic | NUM\_CHANNELS | Output | Yes | SRAM space available per channel |
| **axis\_backpressure\_active** | logic | 1 | Output | Yes | AXIS backpressure status |
| **error\_axi\_timeout** | logic | 1 | Output | Yes | AXI timeout error |
| **error\_axis\_protocol** | logic | 1 | Output | Yes | AXIS protocol error |
| **error\_buffer\_underflow** | logic | 1 | Output | Yes | Buffer underflow error |
| **error\_channel\_id** | logic | CHAN\_WIDTH | Output | Yes | Channel with error |

##### Architecture

###### Internal Components

* **Source AXI Read Engine**: Multi-channel AXI read operations with address tracking
* **Source SRAM Control**: Multi-channel buffering with preallocation and flow control
* **AXIS Master**: Multi-channel arbitration with standard AXIS backpressure handling
* **Monitor Bus Aggregator**: Round-robin aggregation from all three components

###### Data Flow Pipeline

1. **Scheduler Request**: Multi-channel scheduler provides data requests with alignment information
2. **AXI Read Operations**: AXI engine performs optimized read operations using pre-calculated alignment
3. **SRAM Buffering**: SRAM control provides multi-channel buffering with flow control
4. **AXIS Transmission**: AXIS master transmits packets with standard backpressure handling
5. **Flow Control**: Standard AXIS backpressure propagates through pipeline

###### Address Alignment Integration

The wrapper integrates with the scheduler’s address alignment bus: - **Pre-Calculated Alignment**: Scheduler provides complete alignment information - **Optimal AXI Planning**: Address offset, burst planning, and chunk enables pre-calculated - **Transfer Phase Management**: Alignment/streaming/final transfer phases coordinated - **Performance Benefits**: Eliminates alignment calculation overhead from AXI critical path

###### Multi-Channel Operation

Each channel operates independently with: - **Independent Credit Tracking**: Separate SRAM and Network credit management - **Channel-Aware Arbitration**: Fair arbitration across active channels - **Per-Channel Status**: Individual idle/busy and error reporting - **Configurable Operation**: Per-channel enable and configuration control

###### Standard AXIS Flow Control

The wrapper implements standard AXIS backpressure: - **SRAM Buffering**: Prevent SRAM buffer overflow - **AXIS Backpressure**: Standard tvalid/tready flow control - **Upstream Coordination**: Backpressure propagates from AXIS to SRAM to AXI - **Buffer Management**: Deep buffering absorbs backpressure transients

##### AXIS Integration

The source data path uses standard AXI-Stream (AXIS4) protocol for packet transmission:

**Key Benefits:** 1. **Industry Standard**: AXIS is widely supported, well-documented protocol 2. **Simplified Flow Control**: Standard tvalid/tready backpressure (no custom credits) 3. **Cleaner Byte Qualification**: Standard tstrb from chunk enables 4. **Packet Framing**: Standard tlast from internal EOS markers 5. **Better Tool Support**: Standard protocol enables better IP integration and verification

**TSTRB from Chunk Enables:** - Chunk enables (16-bit for 512-bit data) -> TSTRB (64-bit byte strobes) - Each chunk enable bit controls 4 bytes (32 bits) - TSTRB provides byte-level granularity for precise data handling

##### Usage Guidelines

###### Performance Optimization

* Use address alignment bus for optimal AXI read planning
* Configure SRAM preallocation thresholds based on workload
* Adjust buffer depths based on downstream AXIS sink latency
* Monitor channel utilization and arbitration efficiency

###### Address Alignment Usage

The wrapper leverages pre-calculated alignment information:

// Alignment information provided by scheduler  
alignment\_info\_t alignment\_info = data\_alignment\_info[channel];  
if (data\_alignment\_valid[channel]) begin  
 // Use pre-calculated alignment for optimal AXI reads  
 ar\_addr <= alignment\_info.aligned\_addr;  
 ar\_len <= alignment\_info.optimal\_burst\_len;  
 // Chunk enables already calculated for TSTRB conversion  
 chunk\_enables <= alignment\_info.chunk\_enables;  
end

###### Flow Control Management

Proper AXIS backpressure handling requires: 1. Monitor SRAM buffer availability and utilization 2. Respect AXIS tready signal for downstream backpressure 3. Use deep buffering to absorb transient backpressure 4. Handle buffer underflow and timeout conditions

###### Error Handling

The wrapper provides comprehensive error detection: - Monitor AXI and AXIS timeout conditions - Check buffer underflow situations - Verify AXIS protocol compliance - Track per-channel error statistics

###### Stream Boundary Processing

The wrapper handles stream boundaries correctly: - EOS propagation through the pipeline - Proper packet boundary management - Credit return on stream completion - Coordination with downstream processors

### MonBus AXIL Group

#### Monitor Bus AXI-Lite Group

##### Overview

The Monitor Bus AXI-Lite Group aggregates monitor bus streams from source and sink data paths, applies configurable filtering based on protocol and packet types, and routes filtered packets to dual output paths for comprehensive system monitoring and interrupt generation.

The wrapper implements round-robin arbitration between source and sink monitor streams, comprehensive packet filtering for AXI, Network, and CORE protocols, and provides both interrupt generation and external logging capabilities through separate AXI-Lite interfaces.

##### Key Features

* **Round-Robin Arbitration**: Between source and sink monitor streams with built-in skid buffering
* **Multi-Protocol Filtering**: Configurable packet filtering for AXI, Network, and CORE protocols
* **Dual Output Paths**: Error/interrupt FIFO for immediate attention and master write FIFO for logging
* **Interrupt Generation**: Automatic interrupt assertion when error FIFO contains events
* **Address Management**: Configurable address range for master write operations with automatic wraparound
* **Deep Buffering**: Separate FIFO depths for error/interrupt and master write paths

##### Interface Specification

###### Clock and Reset

| Signal Name | Type | Width | Direction | Required | Description |
| --- | --- | --- | --- | --- | --- |
| **axi\_aclk** | logic | 1 | Input | Yes | AXI clock |
| **axi\_aresetn** | logic | 1 | Input | Yes | AXI active-low reset |

###### Monitor Bus Inputs

| Signal Name | Type | Width | Direction | Required | Description |
| --- | --- | --- | --- | --- | --- |
| **source\_monbus\_valid** | logic | 1 | Input | Yes | Source monitor stream valid |
| **source\_monbus\_ready** | logic | 1 | Output | Yes | Source monitor stream ready |
| **source\_monbus\_packet** | logic | 64 | Input | Yes | Source monitor packet data |
| **sink\_monbus\_valid** | logic | 1 | Input | Yes | Sink monitor stream valid |
| **sink\_monbus\_ready** | logic | 1 | Output | Yes | Sink monitor stream ready |
| **sink\_monbus\_packet** | logic | 64 | Input | Yes | Sink monitor packet data |

###### AXI-Lite Slave Interface (Error/Interrupt FIFO Access)

| Signal Name | Type | Width | Direction | Required | Description |
| --- | --- | --- | --- | --- | --- |
| **s\_axil\_arvalid** | logic | 1 | Input | Yes | Read address valid |
| **s\_axil\_arready** | logic | 1 | Output | Yes | Read address ready |
| **s\_axil\_araddr** | logic | ADDR\_WIDTH | Input | Yes | Read address |
| **s\_axil\_arprot** | logic | 3 | Input | Yes | Read protection attributes |
| **s\_axil\_rvalid** | logic | 1 | Output | Yes | Read data valid |
| **s\_axil\_rready** | logic | 1 | Input | Yes | Read data ready |
| **s\_axil\_rdata** | logic | DATA\_WIDTH | Output | Yes | Read data |
| **s\_axil\_rresp** | logic | 2 | Output | Yes | Read response |

###### AXI-Lite Master Interface (Monitor Data Logging)

| Signal Name | Type | Width | Direction | Required | Description |
| --- | --- | --- | --- | --- | --- |
| **m\_axil\_awvalid** | logic | 1 | Output | Yes | Write address valid |
| **m\_axil\_awready** | logic | 1 | Input | Yes | Write address ready |
| **m\_axil\_awaddr** | logic | ADDR\_WIDTH | Output | Yes | Write address |
| **m\_axil\_awprot** | logic | 3 | Output | Yes | Write protection attributes |
| **m\_axil\_wvalid** | logic | 1 | Output | Yes | Write data valid |
| **m\_axil\_wready** | logic | 1 | Input | Yes | Write data ready |
| **m\_axil\_wdata** | logic | DATA\_WIDTH | Output | Yes | Write data |
| **m\_axil\_wstrb** | logic | DATA\_WIDTH/8 | Output | Yes | Write strobes |
| **m\_axil\_bvalid** | logic | 1 | Input | Yes | Write response valid |
| **m\_axil\_bready** | logic | 1 | Output | Yes | Write response ready |
| **m\_axil\_bresp** | logic | 2 | Input | Yes | Write response |

###### Configuration Interface

| Signal Name | Type | Width | Direction | Required | Description |
| --- | --- | --- | --- | --- | --- |
| **cfg\_base\_addr** | logic | ADDR\_WIDTH | Input | Yes | Base address for master writes |
| **cfg\_limit\_addr** | logic | ADDR\_WIDTH | Input | Yes | Limit address for master writes |

###### AXI Protocol Configuration

| Signal Name | Type | Width | Direction | Required | Description |
| --- | --- | --- | --- | --- | --- |
| **cfg\_axi\_pkt\_mask** | logic | 16 | Input | Yes | Drop mask for AXI packet types |
| **cfg\_axi\_err\_select** | logic | 16 | Input | Yes | Error FIFO select for AXI packet types |
| **cfg\_axi\_error\_mask** | logic | 16 | Input | Yes | AXI error event mask |
| **cfg\_axi\_timeout\_mask** | logic | 16 | Input | Yes | AXI timeout event mask |
| **cfg\_axi\_compl\_mask** | logic | 16 | Input | Yes | AXI completion event mask |
| **cfg\_axi\_thresh\_mask** | logic | 16 | Input | Yes | AXI threshold event mask |
| **cfg\_axi\_perf\_mask** | logic | 16 | Input | Yes | AXI performance event mask |
| **cfg\_axi\_addr\_mask** | logic | 16 | Input | Yes | AXI address match event mask |
| **cfg\_axi\_debug\_mask** | logic | 16 | Input | Yes | AXI debug event mask |

###### Network Protocol Configuration

| Signal Name | Type | Width | Direction | Required | Description |
| --- | --- | --- | --- | --- | --- |
| **cfg\_network\_pkt\_mask** | logic | 16 | Input | Yes | Drop mask for Network packet types |
| **cfg\_network\_err\_select** | logic | 16 | Input | Yes | Error FIFO select for Network packet types |
| **cfg\_network\_error\_mask** | logic | 16 | Input | Yes | Network error event mask |
| **cfg\_network\_timeout\_mask** | logic | 16 | Input | Yes | Network timeout event mask |
| **cfg\_network\_compl\_mask** | logic | 16 | Input | Yes | Network completion event mask |
| **cfg\_network\_credit\_mask** | logic | 16 | Input | Yes | Network credit event mask |
| **cfg\_network\_channel\_mask** | logic | 16 | Input | Yes | Network channel event mask |
| **cfg\_network\_stream\_mask** | logic | 16 | Input | Yes | Network stream event mask |

###### CORE Protocol Configuration

| Signal Name | Type | Width | Direction | Required | Description |
| --- | --- | --- | --- | --- | --- |
| **cfg\_core\_pkt\_mask** | logic | 16 | Input | Yes | Drop mask for CORE packet types |
| **cfg\_core\_err\_select** | logic | 16 | Input | Yes | Error FIFO select for CORE packet types |
| **cfg\_core\_error\_mask** | logic | 16 | Input | Yes | CORE error event mask |
| **cfg\_core\_timeout\_mask** | logic | 16 | Input | Yes | CORE timeout event mask |
| **cfg\_core\_compl\_mask** | logic | 16 | Input | Yes | CORE completion event mask |
| **cfg\_core\_thresh\_mask** | logic | 16 | Input | Yes | CORE threshold event mask |
| **cfg\_core\_perf\_mask** | logic | 16 | Input | Yes | CORE performance event mask |
| **cfg\_core\_debug\_mask** | logic | 16 | Input | Yes | CORE debug event mask |

###### Status and Interrupt

| Signal Name | Type | Width | Direction | Required | Description |
| --- | --- | --- | --- | --- | --- |
| **irq\_out** | logic | 1 | Output | Yes | Interrupt output (asserted when error FIFO not empty) |

##### Architecture

###### Internal Components

* **Monitor Bus Arbiter**: Round-robin arbitration between source and sink streams
* **Packet Filter**: Multi-level filtering based on protocol, packet type, and event codes
* **Error/Interrupt FIFO**: Stores filtered events requiring immediate attention
* **Master Write FIFO**: Stores filtered events for external logging
* **AXI-Lite Slave**: Provides access to error/interrupt FIFO contents
* **AXI-Lite Master**: Writes monitor data to configurable memory regions

###### Filtering Pipeline

1. **Protocol Extraction**: Extract protocol field from 64-bit monitor packet
2. **Packet Type Filtering**: Apply protocol-specific packet type masks
3. **Event Code Filtering**: Apply individual event code masks within each packet type
4. **Routing Decision**: Route to error FIFO, master write FIFO, or drop based on configuration

###### Address Management

The master write interface maintains an address counter that: - Starts at cfg\_base\_addr - Increments by 4 bytes (32-bit bus) or 8 bytes (64-bit bus) per transaction - Wraps to cfg\_base\_addr when exceeding cfg\_limit\_addr - Supports both 32-bit and 64-bit AXI-Lite data widths

##### Monitor Bus AXI-Lite Group FSM

The Monitor Bus AXI-Lite Group implements a dedicated master write state machine that manages efficient logging of filtered monitor packets to external memory through configurable AXI-Lite write operations. The FSM coordinates 64-bit monitor packet adaptation to both 32-bit and 64-bit AXI-Lite data buses with sophisticated address management and comprehensive error handling for reliable system observability.

|  |
| --- |
| Monitor Bus Write FSM |

Monitor Bus Write FSM

**Key Operations:** - **Write Transaction Management**: AXI4-Lite compliant write operations with proper address and data phase coordination for reliable monitor packet logging - **Bus Width Adaptation**: Dynamic adaptation between 64-bit monitor packets and configurable 32-bit/64-bit AXI-Lite data buses with automatic two-phase writes for narrow buses - **Address Management**: Configurable base address with automatic increment and wraparound logic for circular buffer logging with overflow prevention - **FIFO Coordination**: Deep FIFO buffering for monitor packets with ready/valid handshake management to prevent data loss during high-frequency monitoring events - **Error Recovery**: Comprehensive error detection and recovery for AXI-Lite transaction failures with monitor event logging for system diagnostics

The FSM operates as a dedicated AXI-Lite master that provides robust monitor packet logging capabilities, implementing a five-state transaction pipeline that ensures reliable write operations while adapting to different bus configurations. The architecture supports both immediate error/interrupt FIFO access through a slave interface and continuous logging through the master interface, enabling comprehensive system monitoring with configurable filtering and dual-path event routing for both real-time alerts and historical analysis.

##### Network 2.0 Support

The monitor bus aggregation supports the Network 2.0 protocol specification, which uses chunk enables instead of the older start/len approach for indicating valid data chunks within each 512-bit packet. This provides more flexible and precise control over partial data transfers.

##### Usage Guidelines

###### Configuration Setup

1. Configure base and limit addresses for master write logging region
2. Set protocol-specific packet type masks to drop unwanted events
3. Configure error select masks to route critical events to interrupt FIFO
4. Set individual event masks for fine-grained filtering control

###### Interrupt Handling

The interrupt output is asserted whenever the error/interrupt FIFO contains one or more events. Software should: 1. Read from the slave AXI-Lite interface to retrieve error events 2. Process events appropriately based on protocol and event type 3. Continue reading until FIFO is empty (interrupt deasserts)

###### Performance Considerations

* Configure FIFO depths based on expected event rates and processing latency
* Use appropriate address ranges to avoid memory conflicts
* Monitor FIFO status to prevent overflow conditions
* Consider using deeper FIFOs for systems with high monitor event rates

### FSM Summary

#### RAPIDS RTL FSM Summary Table

##### Complete List of State Machines in RAPIDS RTL

| FSM Name | Module File | States | PlantUML Status | Purpose |
| --- | --- | --- | --- | --- |
| **Scheduler FSM** | scheduler.sv | 6 states | CURRENT | Descriptor execution, credit management, program sequencing |
| **Address Alignment FSM** | scheduler.sv | 7 states | CURRENT | Address alignment calculation and transfer planning |
| **Descriptor Engine FSM** | descriptor\_engine.sv | 6 states | CURRENT | APB/RDA descriptor processing, AXI read operations |
| **Program Write Engine FSM** | program\_engine.sv | 4 states | CURRENT | Post-processing program writes, AXI write operations |
| **Sink AXI Write Engine FSM** | sink\_axi\_write\_engine.sv | 7 states | CURRENT | Multi-channel AXI write arbitration and data transfer |
| **Source SRAM Control** | source\_sram\_control.sv | Resource Mgmt | CURRENT | Multi-channel SRAM resource management and EOS handling |
| **Sink SRAM Control FSM** | sink\_sram\_control.sv | 8 states | CURRENT | Single-write/multi-read SRAM control with stream boundaries |
| **Network Master FSM** | network\_master.sv | Pipeline stages | CURRENT | Credit-based Network packet transmission |
| **Network Slave ACK FSM** | network\_slave.sv | 6 states | CURRENT | ACK generation and priority arbitration |
| **Monitor Bus Write FSM** | monbus\_axil\_group.sv | 5 states | CURRENT | AXI4-Lite master write for monitor events |

##### State Machine Details

###### 1. Scheduler FSM (scheduler.sv)

**States:** SCHED\_IDLE, SCHED\_WAIT\_FOR\_CONTROL, SCHED\_DESCRIPTOR\_ACTIVE, SCHED\_ISSUE\_PROGRAM0, SCHED\_ISSUE\_PROGRAM1, SCHED\_ERROR

**Key Features:** - Dual EOS handling (packet-level + descriptor-level) - Credit management with early warning - Timeout detection and recovery - Sequential program engine coordination - Channel reset support with graceful shutdown - Stream boundary processing (EOS support) - Generic RDA completion interface - Sticky error flags for comprehensive tracking - Monitor bus integration with standardized packets

###### 2. Address Alignment FSM (scheduler.sv)

**States:** ALIGN\_IDLE, ANALYZE\_ADDRESS, CALC\_FIRST\_TRANSFER, CALC\_STREAMING, CALC\_FINAL\_TRANSFER, ALIGNMENT\_COMPLETE, ALIGNMENT\_ERROR

**Key Features:** - **Parallel Operation**: Runs in parallel with main scheduler FSM during SCHED\_DESCRIPTOR\_ACTIVE - **Pre-Calculated Alignment**: Complete address alignment analysis before AXI transactions - **Alignment Information Bus**: Provides comprehensive alignment data to AXI engines - **Hidden Latency**: Alignment calculation during non-critical descriptor processing - **Optimal Performance**: Eliminates alignment overhead from AXI critical timing paths

**Architecture Benefits:** - Single alignment calculation unit (resource efficient) - Pre-calculated chunk enables and burst parameters - Complete transfer sequence planning - Clean AXI engine interfaces without alignment logic

###### 3. Descriptor Engine FSM (descriptor\_engine.sv)

**States:** DESC\_IDLE, DESC\_APB\_READ, DESC\_AXI\_ADDR, DESC\_AXI\_DATA, DESC\_PROCESS, DESC\_ERROR

**Key Features:** - APB interface for descriptor reads - AXI4 master for RDA descriptor fetching - Descriptor validation and parsing - Stream boundary detection - Error handling and recovery - Monitor bus integration

###### 4. Program Write Engine FSM (program\_engine.sv)

**States:** PROG\_IDLE, PROG\_AXI\_ADDR, PROG\_AXI\_DATA, PROG\_RESPONSE

**Key Features:** - Post-processing program execution - AXI4 master write operations - Error handling and recovery - Completion signaling to scheduler

###### 5. Sink AXI Write Engine FSM (sink\_axi\_write\_engine.sv)

**States:** WRITE\_IDLE, WRITE\_ADDR, WRITE\_DATA, WRITE\_RESP, WRITE\_ERROR, WRITE\_FLUSH, WRITE\_BARRIER

**Key Features:** - Multi-channel arbitration (round-robin with priorities) - Stream boundary handling (EOS barriers) - Error recovery and reporting - Backpressure management - Channel reset coordination

###### 6. Source SRAM Control (source\_sram\_control.sv)

**Control Patterns:** MONITOR, WRITE\_VALIDATE, WRITE\_EXECUTE, READ\_SERVE, CONSUMPTION\_UPDATE, PREALLOC\_MANAGE, ERROR\_HANDLE

**Key Features:** - **Multi-Channel Resource Management**: Up to 32 concurrent channels - **EOS-Only Format**: 531-bit SRAM entries (reduced from 533-bit) - **Preallocation System**: Credit-based write authorization - **Channel Availability Interface**: Real-time space tracking - **Loaded Lines Generation**: Network Master channel selection assistance - **Deadlock Prevention**: Safety margins and threshold monitoring - **Performance Optimization**: Concurrent multi-channel operations

**Architecture Benefits:** - Dynamic resource allocation prevents waste - High throughput with scalable channel count - Comprehensive error recovery and monitoring - Clean separation of resource management from data flow

###### 7. Sink SRAM Control FSM (sink\_sram\_control.sv)

**States:** IDLE, WRITE\_READY, WRITE\_EXECUTE, READ\_ARBITRATE, READ\_EXECUTE, CONSUMPTION\_NOTIFY, ERROR, BARRIER\_MGMT

**Key Features:** - **Single Write Interface**: From Network Slave (533-bit format) - **Multi-Channel Read Interface**: To AXI Write Engine - **RDA Packet Bypass**: Direct routing to Descriptor Engine - **Stream Boundary Management**: EOS barrier handling - **Round-Robin Arbitration**: Fair channel selection with priorities - **Threshold-Based Flow Control**: Optimal buffer utilization

**Architecture Benefits:** - Clean write/read interface separation - Efficient metadata handling (4.1% overhead) - High buffer efficiency (95%+) - Robust boundary processing

###### 8. Network Master FSM (network\_master.sv)

**Pipeline Stages:** Credit-based transmission pipeline

**Key Features:** - Credit-based flow control - Packet generation and transmission - Channel arbitration and selection - Network interface management

###### 9. Network Slave ACK FSM (network\_slave.sv)

**States:** ACK\_IDLE, ACK\_PRIORITY, ACK\_GENERATE, ACK\_SEND, ACK\_COMPLETE, ACK\_ERROR

**Key Features:** - Priority-based ACK generation - Multiple ACK types support - Error handling and recovery - Network coordination

###### 10. Monitor Bus Write FSM (monbus\_axil\_group.sv)

**States:** WRITE\_IDLE, WRITE\_ADDR, WRITE\_DATA\_LOW, WRITE\_DATA\_HIGH, WRITE\_RESP

**Key Features:** - AXI4-Lite master write transactions - 64-bit monitor packet to 32-bit bus adaptation - Address increment management - Error handling and reporting - Two-phase write for 32-bit buses (low/high words) - Single-phase write for 64-bit buses - Configurable base address for monitor logging - FIFO-based packet queuing

##### FSM Interactions and Dependencies

###### Primary Data Flow FSMs

1. **Descriptor Engine** -> **Scheduler** -> **Program Write Engine**
2. **Network Slave** -> **Sink SRAM Control** -> **Sink AXI Write Engine**
3. **Source AXI Read Engine** -> **Source SRAM Control** -> **Network Master**
4. **Scheduler** -> **Address Alignment FSM** (parallel operation)

###### Support/Infrastructure FSMs

1. **Network Slave ACK FSM** - Supports packet reception
2. **Monitor Bus Write FSM** - Supports event logging
3. **Address Alignment FSM** - Supports optimal AXI performance

###### SRAM Control Comparison

| Feature | Source SRAM Control | Sink SRAM Control |
| --- | --- | --- |
| **Architecture** | Resource Management | Traditional FSM |
| **Write Interface** | Multi-Channel (up to 32) | Single Channel |
| **Read Interface** | Single Channel | Multi-Channel |
| **SRAM Format** | 531 bits (EOS only) | 533 bits (EOS + Type) |
| **Overhead** | 3.3% | 4.1% |
| **Primary Feature** | Preallocation Credits | Stream Barriers |
| **Channel Selection** | Dynamic Availability | Round-Robin + Priority |

###### Reset and Error Coordination

* All FSMs support channel reset with graceful shutdown
* Error states propagate through data flow chain
* Monitor FSMs provide observability for all operations
* Source/Sink SRAM controls coordinate resource availability

##### Address Alignment FSM Integration

The Address Alignment FSM is a critical component that:

###### **Parallel Operation Model**

* Runs concurrently with main scheduler FSM during SCHED\_DESCRIPTOR\_ACTIVE
* Provides alignment information before AXI engines begin transactions
* Hidden latency calculation during descriptor processing phase

###### **Alignment Information Bus**

typedef struct packed {  
 logic is\_aligned; // Pre-calculated  
 logic [5:0] addr\_offset; // Address alignment offset  
 logic [7:0] first\_burst\_len; // Optimized first burst  
 logic [7:0] optimal\_burst\_len; // Streaming burst length  
 logic [7:0] final\_burst\_len; // Final transfer burst  
 logic [NUM\_CHUNKS-1:0] first\_chunk\_enables; // First transfer chunks  
 logic [NUM\_CHUNKS-1:0] final\_chunk\_enables; // Final transfer chunks  
} alignment\_info\_t;

###### **Performance Benefits**

1. **Zero AXI Latency**: No alignment calculation in critical path
2. **Optimal Burst Planning**: Pre-calculated lengths maximize efficiency
3. **Precise Resource Usage**: Exact chunk enable prediction
4. **Enhanced Throughput**: Parallel operation eliminates alignment overhead

This comprehensive FSM architecture provides robust, high-performance data processing with optimal resource utilization and comprehensive error handling across the entire RAPIDS pipeline.

## Chapter 3: Interfaces and Busses

### Top-Level Ports

## RAPIDS Top-Level Interfaces v3.0 - AXIS4 Migration

### Clock and Reset

| Signal | IO | Description |
| --- | --- | --- |
| core\_clk | I | System clock |
| core\_rstn | I | Active-low reset |

### Source Data Path

#### AXI4 AR Master Interface (512-bit data)

| Signal | IO | Description |
| --- | --- | --- |
| axi\_src\_data\_ar\_valid | O | Read address valid |
| axi\_src\_data\_ar\_ready | I | Read address ready |
| axi\_src\_data\_ar\_addr[39:0] | O | Read address |
| axi\_src\_data\_ar\_len[7:0] | O | Burst length - 1 |
| axi\_src\_data\_ar\_size[2:0] | O | Transfer size |
| axi\_src\_data\_ar\_burst[1:0] | O | Burst type |
| axi\_src\_data\_ar\_id[7:0] | O | Transaction ID |
| axi\_src\_data\_ar\_lock | O | Lock type |
| axi\_src\_data\_ar\_cache[3:0] | O | Cache attributes |
| axi\_src\_data\_ar\_prot[2:0] | O | Protection attributes |
| axi\_src\_data\_ar\_qos[3:0] | O | Quality of Service |
| axi\_src\_data\_ar\_region[3:0] | O | Region identifier |
| axi\_src\_data\_ar\_user | O | User-defined |
| axi\_src\_data\_r\_valid | I | Read data valid |
| axi\_src\_data\_r\_ready | O | Read data ready |
| axi\_src\_data\_r\_data[511:0] | I | Read data |
| axi\_src\_data\_r\_id[7:0] | I | Transaction ID |
| axi\_src\_data\_r\_resp[1:0] | I | Read response |
| axi\_src\_data\_r\_last | I | Last transfer in burst |
| axi\_src\_data\_r\_user | I | User-defined |

#### AXI4 AW Master Interface (32-bit control data)

| Signal | IO | Description |
| --- | --- | --- |
| axi\_src\_ctrl\_aw\_valid | O | Write address valid |
| axi\_src\_ctrl\_aw\_ready | I | Write address ready |
| axi\_src\_ctrl\_aw\_addr[39:0] | O | Write address |
| axi\_src\_ctrl\_aw\_len[7:0] | O | Burst length - 1 |
| axi\_src\_ctrl\_aw\_size[2:0] | O | Transfer size |
| axi\_src\_ctrl\_aw\_burst[1:0] | O | Burst type |
| axi\_src\_ctrl\_aw\_id[7:0] | O | Transaction ID |
| axi\_src\_ctrl\_aw\_lock | O | Lock type |
| axi\_src\_ctrl\_aw\_cache[3:0] | O | Cache attributes |
| axi\_src\_ctrl\_aw\_prot[2:0] | O | Protection attributes |
| axi\_src\_ctrl\_aw\_qos[3:0] | O | Quality of Service |
| axi\_src\_ctrl\_aw\_region[3:0] | O | Region identifier |
| axi\_src\_ctrl\_aw\_user | O | User-defined |
| axi\_src\_ctrl\_w\_valid | O | Write data valid |
| axi\_src\_ctrl\_w\_ready | I | Write data ready |
| axi\_src\_ctrl\_w\_data[31:0] | O | Write data |
| axi\_src\_ctrl\_w\_strb[3:0] | O | Write strobes |
| axi\_src\_ctrl\_w\_last | O | Last transfer in burst |
| axi\_src\_ctrl\_w\_user | O | User-defined |
| axi\_src\_ctrl\_b\_valid | I | Write response valid |
| axi\_src\_ctrl\_b\_ready | O | Write response ready |
| axi\_src\_ctrl\_b\_id[7:0] | I | Transaction ID |
| axi\_src\_ctrl\_b\_resp[1:0] | I | Write response |
| axi\_src\_ctrl\_b\_user | I | User-defined |

#### AXI4 AR Master Interface (512-bit descriptor data)

| Signal | IO | Description |
| --- | --- | --- |
| axi\_src\_desc\_ar\_valid | O | Read address valid |
| axi\_src\_desc\_ar\_ready | I | Read address ready |
| axi\_src\_desc\_ar\_addr[39:0] | O | Read address |
| axi\_src\_desc\_ar\_len[7:0] | O | Burst length - 1 |
| axi\_src\_desc\_ar\_size[2:0] | O | Transfer size |
| axi\_src\_desc\_ar\_burst[1:0] | O | Burst type |
| axi\_src\_desc\_ar\_id[7:0] | O | Transaction ID |
| axi\_src\_desc\_ar\_lock | O | Lock type |
| axi\_src\_desc\_ar\_cache[3:0] | O | Cache attributes |
| axi\_src\_desc\_ar\_prot[2:0] | O | Protection attributes |
| axi\_src\_desc\_ar\_qos[3:0] | O | Quality of Service |
| axi\_src\_desc\_ar\_region[3:0] | O | Region identifier |
| axi\_src\_desc\_ar\_user | O | User-defined |
| axi\_src\_desc\_r\_valid | I | Read data valid |
| axi\_src\_desc\_r\_ready | O | Read data ready |
| axi\_src\_desc\_r\_data[511:0] | I | Read data |
| axi\_src\_desc\_r\_id[7:0] | I | Transaction ID |
| axi\_src\_desc\_r\_resp[1:0] | I | Read response |
| axi\_src\_desc\_r\_last | I | Last transfer in burst |
| axi\_src\_desc\_r\_user | I | User-defined |

#### AXI4-Lite Slave Interface (Source Configuration)

| Signal | IO | Description |
| --- | --- | --- |
| axil\_src\_cfg\_aw\_valid | I | Write address valid |
| axil\_src\_cfg\_aw\_ready | O | Write address ready |
| axil\_src\_cfg\_aw\_addr[39:0] | I | Write address |
| axil\_src\_cfg\_aw\_prot[2:0] | I | Protection attributes |
| axil\_src\_cfg\_w\_valid | I | Write data valid |
| axil\_src\_cfg\_w\_ready | O | Write data ready |
| axil\_src\_cfg\_w\_data[31:0] | I | Write data |
| axil\_src\_cfg\_w\_strb[3:0] | I | Write strobes |
| axil\_src\_cfg\_b\_valid | O | Write response valid |
| axil\_src\_cfg\_b\_ready | I | Write response ready |
| axil\_src\_cfg\_b\_resp[1:0] | O | Write response |
| axil\_src\_cfg\_ar\_valid | I | Read address valid |
| axil\_src\_cfg\_ar\_ready | O | Read address ready |
| axil\_src\_cfg\_ar\_addr[39:0] | I | Read address |
| axil\_src\_cfg\_ar\_prot[2:0] | I | Protection attributes |
| axil\_src\_cfg\_r\_valid | O | Read data valid |
| axil\_src\_cfg\_r\_ready | I | Read data ready |
| axil\_src\_cfg\_r\_data[31:0] | O | Read data |
| axil\_src\_cfg\_r\_resp[1:0] | O | Read response |

#### AXI4-Stream Master Interface (TX) - NEW v3.0

| Signal | IO | Description |
| --- | --- | --- |
| axis\_src\_tx\_tdata[511:0] | O | Stream data payload |
| axis\_src\_tx\_tstrb[63:0] | O | Byte strobes (write enables) |
| axis\_src\_tx\_tlast | O | Last transfer in packet |
| axis\_src\_tx\_tvalid | O | Stream data valid |
| axis\_src\_tx\_tready | I | Stream ready (backpressure) |
| axis\_src\_tx\_tuser[15:0] | O | User sideband (packet metadata) |

**TUSER Encoding (Source TX):**

[15:8] - Reserved for future use  
[7:0] - Packet type/flags

**Note:** AXIS uses standard tstrb for byte-level validity instead of custom chunk\_enables. All credits and ACK mechanisms removed from streaming interface.

### Sink Data Path

#### AXI4 AW Master Interface (512-bit data)

| Signal | IO | Description |
| --- | --- | --- |
| axi\_snk\_data\_aw\_valid | O | Write address valid |
| axi\_snk\_data\_aw\_ready | I | Write address ready |
| axi\_snk\_data\_aw\_addr[39:0] | O | Write address |
| axi\_snk\_data\_aw\_len[7:0] | O | Burst length - 1 |
| axi\_snk\_data\_aw\_size[2:0] | O | Transfer size |
| axi\_snk\_data\_aw\_burst[1:0] | O | Burst type |
| axi\_snk\_data\_aw\_id[7:0] | O | Transaction ID |
| axi\_snk\_data\_aw\_lock | O | Lock type |
| axi\_snk\_data\_aw\_cache[3:0] | O | Cache attributes |
| axi\_snk\_data\_aw\_prot[2:0] | O | Protection attributes |
| axi\_snk\_data\_aw\_qos[3:0] | O | Quality of Service |
| axi\_snk\_data\_aw\_region[3:0] | O | Region identifier |
| axi\_snk\_data\_aw\_user | O | User-defined |
| axi\_snk\_data\_w\_valid | O | Write data valid |
| axi\_snk\_data\_w\_ready | I | Write data ready |
| axi\_snk\_data\_w\_data[511:0] | O | Write data |
| axi\_snk\_data\_w\_strb[63:0] | O | Write strobes |
| axi\_snk\_data\_w\_last | O | Last transfer in burst |
| axi\_snk\_data\_w\_user | O | User-defined |
| axi\_snk\_data\_b\_valid | I | Write response valid |
| axi\_snk\_data\_b\_ready | O | Write response ready |
| axi\_snk\_data\_b\_id[7:0] | I | Transaction ID |
| axi\_snk\_data\_b\_resp[1:0] | I | Write response |
| axi\_snk\_data\_b\_user | I | User-defined |

#### AXI4 AW Master Interface (32-bit control data)

| Signal | IO | Description |
| --- | --- | --- |
| axi\_snk\_ctrl\_aw\_valid | O | Write address valid |
| axi\_snk\_ctrl\_aw\_ready | I | Write address ready |
| axi\_snk\_ctrl\_aw\_addr[39:0] | O | Write address |
| axi\_snk\_ctrl\_aw\_len[7:0] | O | Burst length - 1 |
| axi\_snk\_ctrl\_aw\_size[2:0] | O | Transfer size |
| axi\_snk\_ctrl\_aw\_burst[1:0] | O | Burst type |
| axi\_snk\_ctrl\_aw\_id[7:0] | O | Transaction ID |
| axi\_snk\_ctrl\_aw\_lock | O | Lock type |
| axi\_snk\_ctrl\_aw\_cache[3:0] | O | Cache attributes |
| axi\_snk\_ctrl\_aw\_prot[2:0] | O | Protection attributes |
| axi\_snk\_ctrl\_aw\_qos[3:0] | O | Quality of Service |
| axi\_snk\_ctrl\_aw\_region[3:0] | O | Region identifier |
| axi\_snk\_ctrl\_aw\_user | O | User-defined |
| axi\_snk\_ctrl\_w\_valid | O | Write data valid |
| axi\_snk\_ctrl\_w\_ready | I | Write data ready |
| axi\_snk\_ctrl\_w\_data[31:0] | O | Write data |
| axi\_snk\_ctrl\_w\_strb[3:0] | O | Write strobes |
| axi\_snk\_ctrl\_w\_last | O | Last transfer in burst |
| axi\_snk\_ctrl\_w\_user | O | User-defined |
| axi\_snk\_ctrl\_b\_valid | I | Write response valid |
| axi\_snk\_ctrl\_b\_ready | O | Write response ready |
| axi\_snk\_ctrl\_b\_id[7:0] | I | Transaction ID |
| axi\_snk\_ctrl\_b\_resp[1:0] | I | Write response |
| axi\_snk\_ctrl\_b\_user | I | User-defined |

#### AXI4 AR Master Interface (512-bit descriptor data)

| Signal | IO | Description |
| --- | --- | --- |
| axi\_snk\_desc\_ar\_valid | O | Read address valid |
| axi\_snk\_desc\_ar\_ready | I | Read address ready |
| axi\_snk\_desc\_ar\_addr[39:0] | O | Read address |
| axi\_snk\_desc\_ar\_len[7:0] | O | Burst length - 1 |
| axi\_snk\_desc\_ar\_size[2:0] | O | Transfer size |
| axi\_snk\_desc\_ar\_burst[1:0] | O | Burst type |
| axi\_snk\_desc\_ar\_id[7:0] | O | Transaction ID |
| axi\_snk\_desc\_ar\_lock | O | Lock type |
| axi\_snk\_desc\_ar\_cache[3:0] | O | Cache attributes |
| axi\_snk\_desc\_ar\_prot[2:0] | Protection attributes |  |
| axi\_snk\_desc\_ar\_qos[3:0] | O | Quality of Service |
| axi\_snk\_desc\_ar\_region[3:0] | O | Region identifier |
| axi\_snk\_desc\_ar\_user | O | User-defined |
| axi\_snk\_desc\_r\_valid | I | Read data valid |
| axi\_snk\_desc\_r\_ready | O | Read data ready |
| axi\_snk\_desc\_r\_data[511:0] | I | Read data |
| axi\_snk\_desc\_r\_id[7:0] | I | Transaction ID |
| axi\_snk\_desc\_r\_resp[1:0] | I | Read response |
| axi\_snk\_desc\_r\_last | I | Last transfer in burst |
| axi\_snk\_desc\_r\_user | I | User-defined |

#### AXI4-Lite Slave Interface (Sink Configuration)

| Signal | IO | Description |
| --- | --- | --- |
| axil\_snk\_cfg\_aw\_valid | I | Write address valid |
| axil\_snk\_cfg\_aw\_ready | O | Write address ready |
| axil\_snk\_cfg\_aw\_addr[39:0] | I | Write address |
| axil\_snk\_cfg\_aw\_prot[2:0] | I | Protection attributes |
| axil\_snk\_cfg\_w\_valid | I | Write data valid |
| axil\_snk\_cfg\_w\_ready | O | Write data ready |
| axil\_snk\_cfg\_w\_data[31:0] | I | Write data |
| axil\_snk\_cfg\_w\_strb[3:0] | I | Write strobes |
| axil\_snk\_cfg\_b\_valid | O | Write response valid |
| axil\_snk\_cfg\_b\_ready | I | Write response ready |
| axil\_snk\_cfg\_b\_resp[1:0] | O | Write response |
| axil\_snk\_cfg\_ar\_valid | I | Read address valid |
| axil\_snk\_cfg\_ar\_ready | O | Read address ready |
| axil\_snk\_cfg\_ar\_addr[39:0] | I | Read address |
| axil\_snk\_cfg\_ar\_prot[2:0] | I | Protection attributes |
| axil\_snk\_cfg\_r\_valid | O | Read data valid |
| axil\_snk\_cfg\_r\_ready | I | Read data ready |
| axil\_snk\_cfg\_r\_data[31:0] | O | Read data |
| axil\_snk\_cfg\_r\_resp[1:0] | O | Read response |

#### AXI4-Stream Slave Interface (RX) - NEW v3.0

| Signal | IO | Description |
| --- | --- | --- |
| axis\_snk\_rx\_tdata[511:0] | I | Stream data payload |
| axis\_snk\_rx\_tstrb[63:0] | I | Byte strobes (write enables) |
| axis\_snk\_rx\_tlast | I | Last transfer in packet |
| axis\_snk\_rx\_tvalid | I | Stream data valid |
| axis\_snk\_rx\_tready | O | Stream ready (backpressure) |
| axis\_snk\_rx\_tuser[15:0] | I | User sideband (packet metadata) |

**TUSER Encoding (Sink RX):**

[15:8] - Reserved for future use  
[7:0] - Packet type/flags

**Note:** AXIS uses standard tstrb for byte-level validity instead of custom chunk\_enables. All credits and ACK mechanisms removed from streaming interface.

### Monitor Bus AXI4-Lite Group Interfaces

#### AXI4-Lite Slave Interface (Error/Interrupt Read)

| Signal | IO | Description |
| --- | --- | --- |
| axil4\_mon\_err\_ar\_valid | I | Read address valid |
| axil4\_mon\_err\_ar\_ready | O | Read address ready |
| axil4\_mon\_err\_ar\_addr[31:0] | I | Read address |
| axil4\_mon\_err\_ar\_prot[2:0] | I | Protection attributes |
| axil4\_mon\_err\_r\_valid | O | Read data valid |
| axil4\_mon\_err\_r\_ready | I | Read data ready |
| axil4\_mon\_err\_r\_data[63:0] | O | Read data (64-bit monitor packets) |
| axil4\_mon\_err\_r\_resp[1:0] | O | Read response |

#### AXI4-Lite Master Interface (Monitor Write)

| Signal | IO | Description |
| --- | --- | --- |
| axil4\_mon\_wr\_aw\_valid | O | Write address valid |
| axil4\_mon\_wr\_aw\_ready | I | Write address ready |
| axil4\_mon\_wr\_aw\_addr[31:0] | O | Write address |
| axil4\_mon\_wr\_aw\_prot[2:0] | O | Protection attributes |
| axil4\_mon\_wr\_w\_valid | O | Write data valid |
| axil4\_mon\_wr\_w\_ready | I | Write data ready |
| axil4\_mon\_wr\_w\_data[31:0] | O | Write data |
| axil4\_mon\_wr\_w\_strb[3:0] | O | Write strobes |
| axil4\_mon\_wr\_b\_valid | I | Write response valid |
| axil4\_mon\_wr\_b\_ready | O | Write response ready |
| axil4\_mon\_wr\_b\_resp[1:0] | I | Write response |

#### AXI4-Lite Slave Interface (Monitor Configuration)

| Signal | IO | Description |
| --- | --- | --- |
| axil4\_mon\_cfg\_aw\_valid | I | Write address valid |
| axil4\_mon\_cfg\_aw\_ready | O | Write address ready |
| axil4\_mon\_cfg\_aw\_addr[31:0] | I | Write address |
| axil4\_mon\_cfg\_aw\_prot[2:0] | I | Protection attributes |
| axil4\_mon\_cfg\_w\_valid | I | Write data valid |
| axil4\_mon\_cfg\_w\_ready | O | Write data ready |
| axil4\_mon\_cfg\_w\_data[31:0] | I | Write data |
| axil4\_mon\_cfg\_w\_strb[3:0] | I | Write strobes |
| axil4\_mon\_cfg\_b\_valid | O | Write response valid |
| axil4\_mon\_cfg\_b\_ready | I | Write response ready |
| axil4\_mon\_cfg\_b\_resp[1:0] | O | Write response |
| axil4\_mon\_cfg\_ar\_valid | I | Read address valid |
| axil4\_mon\_cfg\_ar\_ready | O | Read address ready |
| axil4\_mon\_cfg\_ar\_addr[31:0] | I | Read address |
| axil4\_mon\_cfg\_ar\_prot[2:0] | I | Protection attributes |
| axil4\_mon\_cfg\_r\_valid | O | Read data valid |
| axil4\_mon\_cfg\_r\_ready | I | Read data ready |
| axil4\_mon\_cfg\_r\_data[31:0] | O | Read data |
| axil4\_mon\_cfg\_r\_resp[1:0] | O | Read response |

### Key Interface Changes from v2.1 to v3.0

#### **REMOVED - Custom Network Protocol:**

* ~~network\_\*\_pkt\_valid/ready~~ - Replaced by standard axis\_\*\_tvalid/tready
* ~~network\_\*\_pkt\_data[511:0]~~ - Replaced by axis\_\*\_tdata[511:0]
* ~~network\_\*\_pkt\_type[1:0]~~ - Moved to axis\_\*\_tuser[7:0]
* ~~network\_\*\_pkt\_addr[7:0]~~ - Removed (no addressing in streaming)
* ~~network\_\*\_pkt\_addr\_par~~ - Removed (parity optional via TUSER if needed)
* ~~network\_\*\_pkt\_eos~~ - Replaced by axis\_\*\_tlast
* ~~network\_\*\_pkt\_par~~ - Removed (parity optional via TUSER if needed)
* **~~ALL ACK signals~~** - Removed completely (no credit/ACK on streaming)
  + ~~network\_\*\_ack\_valid/ready~~
  + ~~network\_\*\_ack\_ack[1:0]~~
  + ~~network\_\*\_ack\_addr[7:0]~~
  + ~~network\_\*\_ack\_addr\_par~~
  + ~~network\_\*\_ack\_par~~
* ~~Embedded chunk\_enables format~~ - Replaced by standard axis\_\*\_tstrb[63:0]

#### **ADDED - Standard AXIS4 Protocol:**

* **axis\_src\_tx\_tdata[511:0]** - Source TX data stream
* **axis\_src\_tx\_tstrb[63:0]** - Byte-level write enables (64 bytes for 512-bit bus)
* **axis\_src\_tx\_tlast** - Packet boundary marker
* **axis\_src\_tx\_tvalid/tready** - Standard handshake protocol
* **axis\_src\_tx\_tuser[15:0]** - Optional metadata sideband
* **axis\_snk\_rx\_tdata[511:0]** - Sink RX data stream
* **axis\_snk\_rx\_tstrb[63:0]** - Byte-level write enables
* **axis\_snk\_rx\_tlast** - Packet boundary marker
* **axis\_snk\_rx\_tvalid/tready** - Standard handshake protocol
* **axis\_snk\_rx\_tuser[15:0]** - Optional metadata sideband

#### **Migration Benefits:**

1. **Industry Standard**: AXIS4 is widely supported, well-documented standard protocol
2. **Simplified Flow Control**: Standard tvalid/tready backpressure, no custom ACK channels
3. **Cleaner Byte Qualification**: Standard tstrb replaces embedded chunk\_enables
4. **Packet Framing**: Standard tlast replaces custom EOS markers
5. **Reduced Complexity**: Eliminated custom packet types, addresses, parity, ACK logic
6. **Tool Support**: Better IP integration, simulation, and verification tool support
7. **No Interface Credits**: Simplified interface - credits remain only in scheduler (internal)

#### **AXIS4 vs Custom Network Protocol Mapping:**

| Custom Network v2.1 | AXIS4 v3.0 | Notes |
| --- | --- | --- |
| network\_\*\_pkt\_data[511:0] | axis\_\*\_tdata[511:0] | Direct data payload |
| network\_\*\_pkt\_chunk\_enables[15:0] (embedded) | axis\_\*\_tstrb[63:0] | Byte-level granularity |
| network\_\*\_pkt\_eos | axis\_\*\_tlast | Standard packet boundary |
| network\_\*\_pkt\_valid/ready | axis\_\*\_tvalid/tready | Standard handshake |
| network\_\*\_pkt\_type[1:0] | axis\_\*\_tuser[7:0] | Metadata in sideband |
| network\_\*\_pkt\_addr[7:0] | **REMOVED** | No addressing in streaming |
| network\_\*\_pkt\_par | **REMOVED** | Optional via TUSER if needed |
| network\_\*\_ack\_\* (all) | **REMOVED** | No ACK/credit on interface |

### Interface Summary

#### **Total AXI Interfaces:**

* **Source:** 3 AXI4 Masters (data read, ctrl write, desc read) + 1 AXI4-Lite Slave (config)
* **Sink:** 3 AXI4 Masters (data write, ctrl write, desc read) + 1 AXI4-Lite Slave (config)
* **Monitor:** 1 AXI4-Lite Master (write) + 2 AXI4-Lite Slaves (error read, config)

#### **Total AXIS Interfaces (NEW v3.0):**

* **Source:** 1 AXIS4 Master (TX streaming)
* **Sink:** 1 AXIS4 Slave (RX streaming)

#### **Key Features:**

* **Standard AXIS4 Protocol** for high-bandwidth streaming
* **Comprehensive AXI4-Lite Configuration** for all subsystems
* **Monitor Bus Aggregation** with configurable filtering
* **Error/Interrupt Handling** via dedicated AXI4-Lite interface
* **Proper Clock/Reset** with core\_clk and core\_rstn
* **Simplified Flow Control** - No custom ACK or credit mechanisms on streaming interfaces
* **Industry-Standard Interfaces** - Better tool support and IP reuse

### AXIS Data Path Integration

#### Source Data Path (Memory -> AXIS TX):

AXI4 Read Master (512-bit)  
 ↓ Read data from system memory  
Source SRAM Control  
 ↓ Buffer management  
AXIS Master (rtl/amba/axis/axis\_master.sv)  
 ↓ axis\_src\_tx\_\* signals  
External AXIS Receiver

**Key Points:** - SRAM control writes to axis\_master FUB interface (fub\_axis\_tdata/tstrb/tlast/tvalid) - AXIS master outputs external m\_axis\_\* signals - Backpressure: axis\_src\_tx\_tready=0 -> SRAM control stalls - Packet framing: SRAM sets tlast on final beat

#### Sink Data Path (AXIS RX -> Memory):

External AXIS Transmitter  
 ↓ axis\_snk\_rx\_\* signals  
AXIS Slave (rtl/amba/axis/axis\_slave.sv)  
 ↓ Internal FUB interface  
Sink SRAM Control  
 ↓ Buffer management  
AXI4 Write Master (512-bit)  
 ↓ Write to system memory

**Key Points:** - AXIS slave receives external s\_axis\_\* signals - Outputs to SRAM via FUB interface (fub\_axis\_tdata/tstrb/tlast/tvalid) - Backpressure: SRAM full -> axis\_snk\_rx\_tready=0 -> upstream stalls - Packet framing: tlast=1 triggers SRAM to finalize packet

**See: See:** - ch03\_interfaces/04\_axis4\_interface\_spec.md - Complete AXIS4 specification - rtl/amba/axis/axis\_master.sv - AXIS master RTL - rtl/amba/axis/axis\_slave.sv - AXIS slave RTL

### Bus Definitions

## AXI4-Lite Interface Specification and Assumptions

### Overview

This document defines the formal specification and assumptions for an AXI4-Lite interface implementation. AXI4-Lite is a subset of AXI4 optimized for simple, lightweight control register interfaces with inherent protocol simplifications.

### Interface Summary

#### Number of Interfaces

* **2 Master Read Interface**: Single read channel for Monitor Packets (one for each Source and Sink)
* **2 Master Write Interface**: Single write channel for Monitor Packets plus a timestamp (one for each Source and Sink)

#### Interface Parameters

| Parameter | Description | Valid Values | Default |
| --- | --- | --- | --- |
| DATA\_WIDTH | AXI data bus width in bits | 32, 64 | 32, 64 |
| ADDR\_WIDTH | AXI address bus width in bits | 32, 64 | 37 |
| STRB\_WIDTH | Write strobe width | DATA\_WIDTH/8 | 8 |

### Core Protocol Assumptions

#### Inherent AXI4-Lite Simplifications

AXI4-Lite protocol inherently provides the following constraints:

| Constraint | Description |
| --- | --- |
| **Single Transfers Only** | No burst transactions supported |
| **No Transaction IDs** | All transactions are in-order |
| **Fixed Transfer Size** | Always uses full data bus width |
| **No User Signals** | Simplified interface without user-defined extensions |

#### Implementation Assumptions

##### Assumption 1: Address Alignment to Data Bus Width

| Aspect | Requirement |
| --- | --- |
| **Alignment Rule** | All AXI4-Lite transactions aligned to data bus width |
| **32-bit bus alignment** | Address[1:0] must be 2’b00 (4-byte aligned) |
| **64-bit bus alignment** | Address[2:0] must be 3’b000 (8-byte aligned) |
| **Rationale** | Maximizes bus efficiency and eliminates unaligned access complexity |
| **Benefit** | Simplifies address decode and data steering logic |

##### Assumption 2: Fixed Transfer Size

| Aspect | Requirement |
| --- | --- |
| **Transfer Size Rule** | All transfers use maximum size equal to bus width |
| **32-bit bus** | AxSIZE = 3’b010 (4 bytes) |
| **64-bit bus** | AxSIZE = 3’b011 (8 bytes) |
| **Rationale** | Maximizes bus utilization and simplifies control logic |
| **Benefit** | No size decode logic required |

##### Assumption 3: No Address Wraparound

| Aspect | Requirement |
| --- | --- |
| **Wraparound Rule** | Transactions never wrap around top of address space |
| **Rationale** | Control register accesses never require wraparound behavior |
| **Benefit** | Simplified address boundary checking |

##### Assumption 4: Standard Protection Attributes

| Access Type | AxPROT Value | Description |
| --- | --- | --- |
| **Normal Access** | 3’b000 | Data, secure, unprivileged |
| **Privileged Access** | 3’b001 | Data, secure, privileged |
| **Rationale** |  | Covers the majority of control register access patterns |

### Master Read Interface Specification

#### Read Address Channel (AR)

| Signal | Width | Direction | Required Values | Description |
| --- | --- | --- | --- | --- |
| ar\_addr | ADDR\_WIDTH | Master->Slave | **8-byte aligned** | Read address |
| ar\_prot | 3 | Master->Slave | Implementation specific | Protection attributes |
| ar\_valid | 1 | Master->Slave | 0 or 1 | Address valid |
| ar\_ready | 1 | Slave->Master | 0 or 1 | Address ready |

#### Read Data Channel (R)

| Signal | Width | Direction | Description |
| --- | --- | --- | --- |
| r\_data | 64 | Slave->Master | Read data |
| r\_resp | 2 | Slave->Master | Read response |
| r\_valid | 1 | Slave->Master | Read data valid |
| r\_ready | 1 | Master->Slave | Read data ready |

#### AXI4-Lite Simplifications (Read)

| Removed Signal | AXI4 Usage | AXI4-Lite Reason |
| --- | --- | --- |
| **ar\_id** | Transaction ID | Single transfers, no transaction IDs |
| **ar\_len** | Burst length | Single transfers only |
| **ar\_size** | Transfer size | Fixed to bus width |
| **ar\_burst** | Burst type | Single transfers only |
| **ar\_lock** | Lock type | Simplified access model |
| **ar\_cache** | Cache attributes | Simplified memory model |
| **ar\_qos** | Quality of Service | Simplified priority model |
| **ar\_region** | Region identifier | Simplified address space |
| **ar\_user** | User-defined | Simplified interface |
| **r\_id** | Transaction ID | No transaction IDs needed |
| **r\_last** | Last transfer | Single transfers only |
| **r\_user** | User-defined | Simplified interface |

### Master Write Interface Specification

#### Write Address Channel (AW)

| Signal | Width | Direction | Required Values | Description |
| --- | --- | --- | --- | --- |
| aw\_addr | ADDR\_WIDTH | Master->Slave | **8-byte aligned** | Write address |
| aw\_prot | 3 | Master->Slave | Implementation specific | Protection attributes |
| aw\_valid | 1 | Master->Slave | 0 or 1 | Address valid |
| aw\_ready | 1 | Slave->Master | 0 or 1 | Address ready |

#### Write Data Channel (W)

| Signal | Width | Direction | Description |
| --- | --- | --- | --- |
| w\_data | 32 | Master->Slave | Write data |
| w\_strb | 4 | Master->Slave | Write strobes (byte enables) |
| w\_valid | 1 | Master->Slave | Write data valid |
| w\_ready | 1 | Slave->Master | Write data ready |

#### Write Response Channel (B)

| Signal | Width | Direction | Description |
| --- | --- | --- | --- |
| b\_resp | 2 | Slave->Master | Write response |
| b\_valid | 1 | Slave->Master | Response valid |
| b\_ready | 1 | Master->Slave | Response ready |

#### AXI4-Lite Simplifications (Write)

| Removed Signal | AXI4 Usage | AXI4-Lite Reason |
| --- | --- | --- |
| **aw\_id** | Transaction ID | Single transfers, no transaction IDs |
| **aw\_len** | Burst length | Single transfers only |
| **aw\_size** | Transfer size | Fixed to bus width |
| **aw\_burst** | Burst type | Single transfers only |
| **aw\_lock** | Lock type | Simplified access model |
| **aw\_cache** | Cache attributes | Simplified memory model |
| **aw\_qos** | Quality of Service | Simplified priority model |
| **aw\_region** | Region identifier | Simplified address space |
| **aw\_user** | User-defined | Simplified interface |
| **w\_last** | Last transfer | Single transfers only |
| **w\_user** | User-defined | Simplified interface |
| **b\_id** | Transaction ID | No transaction IDs needed |
| **b\_user** | User-defined | Simplified interface |

### Address Requirements

#### Address Alignment Rules

| Alignment Type | Formula | Description |
| --- | --- | --- |
| **Valid Address** | (Address % 4) == 0 | Must be 8-byte aligned |
| **Mandatory Alignment** | Address[2:0] must be 3’b000 | Per Assumption 1 |

#### Address Validation Examples

| Address Category | Examples | Status |
| --- | --- | --- |
| **Valid (84byte aligned)** | 0x1000, 0x1004, 0x1008, 0x100C | Accepted |
| **Invalid (unaligned)** | 0x1001, 0x1002, 0x1003 | DECERR response |

### Response Codes

#### Response Code Specification

| Value | Name | Description | Usage in Control Registers |
| --- | --- | --- | --- |
| **2’b00** | OKAY | Normal access success | Successful register access |
| **2’b01** | EXOKAY | Exclusive access success | **Not used in AXI4-Lite** |
| **2’b10** | SLVERR | Slave error | Invalid register access |
| **2’b11** | DECERR | Decode error | **Address decode failure or misalignment** |

#### Response Usage Guidelines

| Response Type | Usage | Description |
| --- | --- | --- |
| **OKAY** | Normal completion | Successful register access |
| **EXOKAY** | Not applicable | AXI4-Lite doesn’t support exclusive accesses |
| **SLVERR** | Register error | Invalid register operation |
| **DECERR** | Address error | Misalignment or decode failure per Assumption 1 |

### Protection Signal Usage

#### Protection Signal Encoding

| Bit | Name | Description | Recommended Usage |
| --- | --- | --- | --- |
| **[0]** | Privileged | 0=Normal, 1=Privileged | Set based on processor mode |
| **[1]** | Non-secure | 0=Secure, 1=Non-secure | Set based on security domain |
| **[2]** | Instruction | 0=Data, 1=Instruction | Always 0 for control registers |

#### Common Protection Patterns

| Pattern | AxPROT Value | Description |
| --- | --- | --- |
| **Normal Data Access** | 3’b000 | Standard register access |
| **Privileged Data Access** | 3’b001 | Privileged register access |
| **Debug Access** | 3’b010 | Debug register access |
| **Privileged Debug** | 3’b011 | Privileged debug access |

### Implementation Benefits

#### Simplified Control Register Interface

| Benefit Area | Simplification | Impact |
| --- | --- | --- |
| **Address Decode** | Simple 8-byte aligned address comparison | Reduced decode logic |
| **Transaction Handling** | No burst or ID tracking required | Simplified state machines |
| **Flow Control** | Straightforward valid-ready handshakes | Reduced complexity |
| **Response Generation** | Simple OKAY/SLVERR/DECERR responses | Minimal response logic |
| **Size Handling** | Fixed 64-bit transfers only | No size decode needed |

#### Address Decode Implementation

| Implementation Aspect | Method | Benefit |
| --- | --- | --- |
| **4-byte Alignment Check** | addr[1:0] == 2’b00 | Simple bit masking |
| **Address Range Check** | addr >= base && addr <= limit | Simple comparisons |
| **Combined Check** | alignment\_ok && range\_ok | Single decode decision |

#### Error Generation Logic

| Error Condition | Check | Response |
| --- | --- | --- |
| **Address Misalignment** | addr[1:0] != 2’b00 | Generate DECERR |
| **Address Out of Range** | !addr\_in\_range(addr) | Generate DECERR |
| **Register Error** | register\_error\_condition | Generate SLVERR |
| **Normal Access** | All checks pass | Generate OKAY |

### Timing Requirements

#### Handshake Protocol

| Protocol Rule | Requirement | Description |
| --- | --- | --- |
| **Valid-Ready Transfer** | Transfer occurs when both VALID and READY are high | Standard AXI handshake |
| **Valid Independence** | VALID can be asserted independently of READY | Master controls valid |
| **Ready Dependency** | READY can depend on VALID state | Slave controls ready |
| **Signal Stability** | Once VALID asserted, all signals stable until READY | Data integrity |

#### Channel Dependencies

| Dependency | Requirement | Description |
| --- | --- | --- |
| **Write Channels** | AW and W channels are independent | Can be presented in any order |
| **Write Response** | B channel waits for both AW and W completion | Response dependency |
| **Read Channels** | R channel waits for AR channel completion | Response dependency |
| **Transaction Ordering** | Multiple outstanding transactions not supported | Inherent AXI4-Lite limitation |

#### Reset Behavior

| Reset Phase | Requirement | Description |
| --- | --- | --- |
| **Active Reset** | aresetn is active-low reset signal | Standard AXI reset |
| **Reset Requirements** | All VALID signals deasserted during reset | Clean reset state |
| **Reset Recovery** | All VALID signals low after reset deassertion | Proper startup |

### Validation Requirements

#### Functional Validation

| Validation Area | Requirements |
| --- | --- |
| **Address Alignment** | Verify all accesses are 8-byte aligned per Assumption 1 |
| **Fixed Size** | Verify all transfers are full 64-bit width per Assumption 2 |
| **Response Correctness** | Verify appropriate response codes (DECERR for misaligned access) |
| **Handshake Compliance** | Verify all valid-ready handshakes |
| **Register Behavior** | Verify read/write register functionality |
| **No Wraparound** | Verify no address wraparound scenarios per Assumption 3 |

#### Timing Validation

| Validation Area | Requirements |
| --- | --- |
| **Setup/Hold** | Verify signal timing requirements |
| **Reset Behavior** | Verify proper reset sequence |
| **Back-pressure** | Verify ready signal behavior under load |

#### Error Injection Testing

| Test Type | Injection Method | Expected Response |
| --- | --- | --- |
| **Misaligned Address** | Inject addresses with addr[2:0] != 0 | DECERR response |
| **Out of Range** | Inject addresses outside valid range | DECERR response |
| **Register Errors** | Inject register-specific errors | SLVERR response |

### Example Transactions

#### 64-bit Register Write

| Parameter | Value | Description |
| --- | --- | --- |
| **Bus Width** | 64 bits (8 bytes) | Data bus configuration |
| **Target Address** | 0x1000 (8-byte aligned) | Valid aligned address |
| **Write Data** | 0xDEADBEEFCAFEBABE | 64-bit data value |
| **Required Settings** | aw\_addr=0x1000, aw\_prot=3’b000, w\_data=0xDEADBEEFCAFEBABE, w\_strb=8’b11111111 | Transaction configuration |

#### AW Transaction Flow

| Step | Action | Signal States |
| --- | --- | --- |
| **1** | Assert aw\_valid with address | aw\_valid=1, aw\_addr=0x1000 |
| **2** | Assert w\_valid with data | w\_valid=1, w\_data=0xDEADBEEFCAFEBABE |
| **3** | Wait for handshakes | aw\_ready=1, w\_ready=1 |
| **4** | Wait for response | b\_valid=1, b\_resp=OKAY |
| **5** | Complete transaction | b\_ready=1 |

#### 64-bit Register Read

| Parameter | Value | Description |
| --- | --- | --- |
| **Bus Width** | 64 bits (8 bytes) | Data bus configuration |
| **Target Address** | 0x1008 (8-byte aligned) | Valid aligned address |
| **Required Settings** | ar\_addr=0x1008, ar\_prot=3’b000 | Transaction configuration |

#### AR Transaction Flow

| Step | Action | Signal States |
| --- | --- | --- |
| **1** | Assert ar\_valid with address | ar\_valid=1, ar\_addr=0x1008 |
| **2** | Wait for address handshake | ar\_ready=1 |
| **3** | Wait for data response | r\_valid=1, r\_resp=OKAY |
| **4** | Complete transaction | r\_ready=1 |
| **5** | Capture data | r\_data (64 bits) |

#### Misaligned Address Example

| Parameter | Value | Description |
| --- | --- | --- |
| **Bus Width** | 64 bits (8 bytes) | Data bus configuration |
| **Target Address** | 0x1004 (misaligned) | Invalid address |
| **Expected Behavior** | Address decode detects misalignment -> DECERR response -> No register access | Error handling |

### Common Use Cases

#### Typical Applications

| Application | Description |
| --- | --- |
| **Control/Status Registers** | 64-bit device configuration and monitoring |
| **Memory-Mapped Peripherals** | Simple register-based devices |
| **Debug Interfaces** | Debug and trace control registers |
| **Configuration Space** | PCIe configuration space access |
| **Performance Counters** | 64-bit performance monitoring registers |

#### Performance Considerations

| Consideration | Impact | Description |
| --- | --- | --- |
| **Latency** | Single-cycle responses preferred | Simple registers |
| **Throughput** | Limited by single outstanding transaction | AXI4-Lite constraint |
| **Efficiency** | 64-bit transfers maximize data efficiency | Modern system optimization |

## AXI4 Interface Specification and Assumptions

### Overview

This document defines the formal specification and assumptions for an AXI4 interface implementation that supports two distinct transfer modes to optimize for different interface types while ensuring robust, predictable operation.

### Interface Summary

#### Number of Interfaces

* **5 Master Read Interfaces**: Descriptor sink, descriptor source, data source, flag sink, and flag source
* **3 Master Write Interfaces**: Data sink, control sink, and control source

#### Interface Parameters

| Parameter | Description | Valid Values | Default |
| --- | --- | --- | --- |
| DATA\_WIDTH | AXI data bus width in bits | 32, 64, 128, 256, 512, 1024 | 32 |
| ADDR\_WIDTH | AXI address bus width in bits | 32, 64 | 37 |
| ID\_WIDTH | AXI ID tag width in bits | 1-16 | 8 |
| USER\_WIDTH | AXI user signal width in bits (optional) | 0-16 | 1 |

#### Interface Types and Transfer Modes

| Interface Group | Channels | Transfer Mode | Address Alignment | Monitor | DCG | Notes |
| --- | --- | --- | --- | --- | --- | --- |
| **AXI4 Master Read-Split** | AR, R | **Flexible** | 4-byte | Yes | Yes | Data interfaces with chunk enables |
| **AXI4 Master Write-Split** | AW, W, B | **Flexible** | 4-byte | Yes | Yes | Data interfaces with chunk enables |
| **AXI4 Master Read** | AR, R | **Simplified** | Bus-width | No | Yes | Control interfaces, fully aligned |
| **AXI4 Master Write** | AW, W, B | **Simplified** | Bus-width | Yes | Yes | Control interfaces, fully aligned |

#### Interface Group Parameter Settings

| Interface Group | Data Width | Address Width | ID Width | User Width | Transfer Mode |
| --- | --- | --- | --- | --- | --- |
| **AXI4 Master Read-Split** | 512 bits | 37 bits | 8 bits | 1 bit | **Flexible** |
| **AXI4 Master Write-Split** | 512 bits | 37 bits | 8 bits | 1 bit | **Flexible** |
| **AXI4 Master Read** | 32 bits | 37 bits | 8 bits | 1 bit | **Simplified** |
| **AXI4 Master Write** | 32 bits | 37 bits | 8 bits | 1 bit | **Simplified** |

#### Interface Configuration Summary

| Interface Type | Interface Group | Transfer Mode | Alignment | Notes |
| --- | --- | --- | --- | --- |
| **Descriptor Sink** | **AXI4 Master Read** | **Simplified** | 32-bit aligned | Control interface |
| **Descriptor Source** | **AXI4 Master Read** | **Simplified** | 32-bit aligned | Control interface |
| **Data Source** | **AXI4 Master Read-Split** | **Flexible** | 4-byte aligned | High-bandwidth data |
| **Data Sink** | **AXI4 Master Write-Split** | **Flexible** | 4-byte aligned | High-bandwidth data |
| **Program Sink** | **AXI4 Master Write** | **Simplified** | 32-bit aligned | Control interface |
| **Program Source** | **AXI4 Master Write** | **Simplified** | 32-bit aligned | Control interface |
| **Flag Sink** | **AXI4 Master Read** | **Simplified** | 32-bit aligned | Control interface |
| **Flag Source** | **AXI4 Master Read** | **Simplified** | 32-bit aligned | Control interface |

### Transfer Mode Specifications

This specification defines two distinct transfer modes to optimize different interface types:

#### Mode 1: Simplified Transfer Mode (Control Interfaces)

Used for control interfaces (descriptors, programs, flags) that prioritize simplicity and predictable timing.

##### **Simplified Mode Assumptions**

| Aspect | Requirement |
| --- | --- |
| **Address Alignment** | All addresses aligned to full data bus width |
| **Transfer Size** | All transfers use maximum size equal to bus width |
| **Burst Type** | Incrementing bursts only (AxBURST = 2’b01) |
| **Transfer Complexity** | Maximum simplicity for predictable operation |

#### Mode 2: Flexible Transfer Mode (Data Interfaces)

Used for high-bandwidth data interfaces that need to handle arbitrary address alignment while maintaining efficiency.

##### **Flexible Mode Assumptions**

| Aspect | Requirement |
| --- | --- |
| **Address Alignment** | 4-byte aligned addresses (minimum alignment) |
| **Transfer Sizes** | Multiple sizes supported: 4, 8, 16, 32, 64 bytes |
| **Burst Type** | Incrementing bursts only (AxBURST = 2’b01) |
| **Alignment Strategy** | Progressive alignment to optimize bus utilization |

### Mode 1: Simplified Transfer Mode Specification

#### Assumption 1: Address Alignment to Data Bus Width

| Aspect | Requirement |
| --- | --- |
| **Alignment Rule** | All AXI transactions aligned to data bus width |
| **32-bit bus (4 bytes)** | Address[1:0] must be 2’b00 |
| **64-bit bus (8 bytes)** | Address[2:0] must be 3’b000 |
| **128-bit bus (16 bytes)** | Address[3:0] must be 4’b0000 |
| **256-bit bus (32 bytes)** | Address[4:0] must be 5’b00000 |
| **512-bit bus (64 bytes)** | Address[5:0] must be 6’b000000 |
| **1024-bit bus (128 bytes)** | Address[6:0] must be 7’b0000000 |
| **Rationale** | Maximizes bus efficiency and eliminates unaligned access complexity |

#### Assumption 2: Fixed Transfer Size

| Aspect | Requirement |
| --- | --- |
| **Transfer Size Rule** | All transfers use maximum size equal to bus width |
| **32-bit bus** | AxSIZE = 3’b010 (4 bytes) |
| **64-bit bus** | AxSIZE = 3’b011 (8 bytes) |
| **128-bit bus** | AxSIZE = 3’b100 (16 bytes) |
| **256-bit bus** | AxSIZE = 3’b101 (32 bytes) |
| **512-bit bus** | AxSIZE = 3’b110 (64 bytes) |
| **1024-bit bus** | AxSIZE = 3’b111 (128 bytes) |
| **Rationale** | Maximizes bus utilization and simplifies address alignment |

### Mode 2: Flexible Transfer Mode Specification

#### Assumption 1: 4-Byte Address Alignment

| Aspect | Requirement |
| --- | --- |
| **Alignment Rule** | All AXI transactions aligned to 4-byte boundaries |
| **Address Constraint** | Address[1:0] must be 2’b00 |
| **Rationale** | Balances flexibility with AXI protocol requirements |
| **Benefit** | Supports arbitrary data placement while maintaining AXI compliance |

#### Assumption 2: Multiple Transfer Sizes

| Transfer Size | AxSIZE Value | Use Case |
| --- | --- | --- |
| **4 bytes** | 3’b010 | Initial alignment, small transfers |
| **8 bytes** | 3’b011 | Progressive alignment |
| **16 bytes** | 3’b100 | Progressive alignment |
| **32 bytes** | 3’b101 | Progressive alignment |
| **64 bytes** | 3’b110 | Optimal full-width transfers |
| **128 bytes** | 3’b111 | Maximum efficiency (1024-bit bus) |

#### Assumption 3: Progressive Alignment Strategy

| Aspect | Requirement |
| --- | --- |
| **Alignment Goal** | Align to 64-byte boundaries for optimal bus utilization |
| **Alignment Sequence** | Use progressive sizes: 4 -> 8 -> 16 -> 32 -> 64 bytes |
| **Optimization** | Choose largest possible transfer size at each step |
| **Example** | Address 0x1004: 4-byte transfer -> aligned to 0x1008, then larger transfers |

#### Assumption 4: Chunk Enable Support

| Aspect | Requirement |
| --- | --- |
| **Chunk Granularity** | 16 chunks of 32-bits each (512-bit bus) |
| **Write Strobes** | Generated from chunk enables for precise byte control |
| **Alignment Transfers** | Chunk patterns optimized for alignment sequences |
| **Benefits** | Precise data validity, optimal memory utilization |

### Common Protocol Assumptions (Both Modes)

#### Assumption 1: Incrementing Bursts Only

| Aspect | Requirement |
| --- | --- |
| **Burst Type** | All AXI bursts use incrementing address mode (AxBURST = 2’b01) |
| **Excluded Types** | No FIXED (2’b00) or WRAP (2’b10) bursts supported |
| **Rationale** | Simplifies address generation logic and covers most use cases |
| **Benefit** | Eliminates wrap boundary calculations and fixed address handling |

#### Assumption 2: No Address Wraparound

| Aspect | Requirement |
| --- | --- |
| **Wraparound Rule** | Transactions never wrap around top of address space |
| **Example** | No 0xFFFFFFFF -> 0x00000000 transitions |
| **Rationale** | Real systems never allow this due to memory layout |
| **Benefit** | Dramatically simplified boundary crossing detection logic |

### Flexible Mode: Address Calculation Examples

#### Progressive Alignment Examples

**Example 1: Address 0x1004 -> 0x1040 (64-byte boundary)**

| Step | Address | Size | AxSIZE | Length | Bytes Transferred | Notes |
| --- | --- | --- | --- | --- | --- | --- |
| 1 | 0x1004 | 4 bytes | 3’b010 | 1 beat | 4 | Initial alignment |
| 2 | 0x1008 | 8 bytes | 3’b011 | 1 beat | 8 | Progressive alignment |
| 3 | 0x1010 | 16 bytes | 3’b100 | 1 beat | 16 | Progressive alignment |
| 4 | 0x1020 | 32 bytes | 3’b101 | 1 beat | 32 | Progressive alignment |
| 5 | 0x1040 | **64 bytes** | 3’b110 | N beats | 64xN | **Optimal transfers** |

**Example 2: Address 0x1010 -> 0x1040 (64-byte boundary)**

| Step | Address | Size | AxSIZE | Length | Bytes Transferred | Notes |
| --- | --- | --- | --- | --- | --- | --- |
| 1 | 0x1010 | 16 bytes | 3’b100 | 1 beat | 16 | Optimal initial size |
| 2 | 0x1020 | 32 bytes | 3’b101 | 1 beat | 32 | Progressive alignment |
| 3 | 0x1040 | **64 bytes** | 3’b110 | N beats | 64xN | **Optimal transfers** |

#### Chunk Enable Pattern Examples

**512-bit Bus with 16x32-bit chunks**

| Transfer Size | Address Offset | Chunk Pattern | Description |
| --- | --- | --- | --- |
| **4 bytes** | 0x04 | 16’h0002 | Chunk 1 only |
| **8 bytes** | 0x08 | 16’h000C | Chunks 2-3 |
| **16 bytes** | 0x10 | 16’h00F0 | Chunks 4-7 |
| **32 bytes** | 0x20 | 16’hFF00 | Chunks 8-15 |
| **64 bytes** | 0x00 | 16’hFFFF | All chunks |

### Master Read Interface Specification

#### Read Address Channel (AR)

| Signal | Width | Direction | Simplified Mode | Flexible Mode | Description |
| --- | --- | --- | --- | --- | --- |
| ar\_addr | ADDR\_WIDTH | Master->Slave | **Bus-width aligned** | **4-byte aligned** | Read address |
| ar\_len | 8 | Master->Slave | 0-255 | 0-255 | Burst length - 1 |
| ar\_size | 3 | Master->Slave | **Fixed per bus** | **Variable: 4-64 bytes** | Transfer size |
| ar\_burst | 2 | Master->Slave | **2’b01 (INCR only)** | **2’b01 (INCR only)** | Burst type |
| ar\_id | ID\_WIDTH | Master->Slave | Any | Any | Transaction ID |
| ar\_lock | 1 | Master->Slave | 1’b0 | 1’b0 | Lock type (normal) |
| ar\_cache | 4 | Master->Slave | Implementation specific | 4’b0011 | Cache attributes |
| ar\_prot | 3 | Master->Slave | Implementation specific | 3’b000 | Protection attributes |
| ar\_qos | 4 | Master->Slave | 4’b0000 | 4’b0000 | Quality of Service |
| ar\_region | 4 | Master->Slave | 4’b0000 | 4’b0000 | Region identifier |
| ar\_user | USER\_WIDTH | Master->Slave | Optional | Optional | User-defined |
| ar\_valid | 1 | Master->Slave | 0 or 1 | 0 or 1 | Address valid |
| ar\_ready | 1 | Slave->Master | 0 or 1 | 0 or 1 | Address ready |

#### Read Data Channel (R)

| Signal | Width | Direction | Description |
| --- | --- | --- | --- |
| r\_data | DATA\_WIDTH | Slave->Master | Read data |
| r\_id | ID\_WIDTH | Slave->Master | Transaction ID |
| r\_resp | 2 | Slave->Master | Read response |
| r\_last | 1 | Slave->Master | Last transfer in burst |
| r\_user | USER\_WIDTH | Slave->Master | User-defined (optional) |
| r\_valid | 1 | Slave->Master | Read data valid |
| r\_ready | 1 | Master->Slave | Read data ready |

### Master Write Interface Specification

#### Write Address Channel (AW)

| Signal | Width | Direction | Simplified Mode | Flexible Mode | Description |
| --- | --- | --- | --- | --- | --- |
| aw\_addr | ADDR\_WIDTH | Master->Slave | **Bus-width aligned** | **4-byte aligned** | Write address |
| aw\_len | 8 | Master->Slave | 0-255 | 0-255 | Burst length - 1 |
| aw\_size | 3 | Master->Slave | **Fixed per bus** | **Variable: 4-64 bytes** | Transfer size |
| aw\_burst | 2 | Master->Slave | **2’b01 (INCR only)** | **2’b01 (INCR only)** | Burst type |
| aw\_id | ID\_WIDTH | Master->Slave | Any | Any | Transaction ID |
| aw\_lock | 1 | Master->Slave | 1’b0 | 1’b0 | Lock type (normal) |
| aw\_cache | 4 | Master->Slave | Implementation specific | 4’b0011 | Cache attributes |
| aw\_prot | 3 | Master->Slave | Implementation specific | 3’b000 | Protection attributes |
| aw\_qos | 4 | Master->Slave | 4’b0000 | 4’b0000 | Quality of Service |
| aw\_region | 4 | Master->Slave | 4’b0000 | 4’b0000 | Region identifier |
| aw\_user | USER\_WIDTH | Master->Slave | Optional | Optional | User-defined |
| aw\_valid | 1 | Master->Slave | 0 or 1 | 0 or 1 | Address valid |
| aw\_ready | 1 | Slave->Master | 0 or 1 | 0 or 1 | Address ready |

#### Write Data Channel (W)

| Signal | Width | Direction | Simplified Mode | Flexible Mode | Description |
| --- | --- | --- | --- | --- | --- |
| w\_data | DATA\_WIDTH | Master->Slave | Write data | Write data | Write data |
| w\_strb | DATA\_WIDTH/8 | Master->Slave | **All 1’s** | **From chunk enables** | Write strobes |
| w\_last | 1 | Master->Slave | Last transfer | Last transfer | Last transfer in burst |
| w\_user | USER\_WIDTH | Master->Slave | Optional | Optional | User-defined |
| w\_valid | 1 | Master->Slave | 0 or 1 | 0 or 1 | Write data valid |
| w\_ready | 1 | Slave->Master | 0 or 1 | 0 or 1 | Write data ready |

#### Write Response Channel (B)

| Signal | Width | Direction | Description |
| --- | --- | --- | --- |
| b\_id | ID\_WIDTH | Slave->Master | Transaction ID |
| b\_resp | 2 | Slave->Master | Write response |
| b\_user | USER\_WIDTH | Slave->Master | User-defined (optional) |
| b\_valid | 1 | Slave->Master | Response valid |
| b\_ready | 1 | Master->Slave | Response ready |

### Address Calculation Rules

#### Simplified Mode Address Generation

| Parameter | Formula | Description |
| --- | --- | --- |
| **First Address** | Must be bus-width aligned | Starting address |
| **Address N** | First\_Address + (N x Bus\_Width\_Bytes) | Address for beat N |
| **Alignment Check** | (Address % Bus\_Width\_Bytes) == 0 | Must always be true |

#### Flexible Mode Address Generation

| Parameter | Formula | Description |
| --- | --- | --- |
| **First Address** | Must be 4-byte aligned | Starting address |
| **Address N** | First\_Address + (N x Transfer\_Size) | Address for beat N |
| **Alignment Check** | (Address % 4) == 0 | Must always be true |
| **Progressive Alignment** | Choose largest size <= bytes\_to\_boundary | Optimization strategy |

#### 4KB Boundary Considerations (Both Modes)

| Validation Rule | Formula | Description |
| --- | --- | --- |
| **4KB Boundary** | Bursts cannot cross 4KB (0x1000) boundaries | AXI specification |
| **Max Burst Calculation** | Max\_Beats = (4KB - (Start\_Address % 4KB)) / Transfer\_Size | Burst limit |
| **Boundary Check** | Verify no 4KB crossings in burst | Mandatory validation |

### Write Strobe Generation

#### Simplified Mode Strobe Generation

| Bus Width | Strobe Pattern | Description |
| --- | --- | --- |
| **32-bit** | 4’b1111 | All bytes valid |
| **512-bit** | 64’hFFFFFFFFFFFFFFFF | All bytes valid |

#### Flexible Mode Strobe Generation

**From Chunk Enables (512-bit bus example):**

// Convert 16x32-bit chunk enables to 64x8-bit write strobes  
for (int chunk = 0; chunk < 16; chunk++) begin  
 if (chunk\_enable[chunk]) begin  
 w\_strb[chunk\*4 +: 4] = 4'hF; // 4 bytes per chunk  
 end  
end

**Alignment Transfer Examples:**

| Transfer Size | Chunk Pattern | Strobe Pattern | Description |
| --- | --- | --- | --- |
| **4 bytes** | 16’h0001 | 64’h000000000000000F | First 4 bytes |
| **16 bytes** | 16’h000F | 64’h00000000000000FF | First 16 bytes |
| **32 bytes** | 16’h00FF | 64’h0000000000FFFFFF | First 32 bytes |
| **64 bytes** | 16’hFFFF | 64’hFFFFFFFFFFFFFFFF | All 64 bytes |

### Response Codes

#### Response Code Specification

| Value | Name | Description | Simplified Mode Usage | Flexible Mode Usage |
| --- | --- | --- | --- | --- |
| **2’b00** | OKAY | Normal access success | Bus-width aligned access | 4-byte aligned access |
| **2’b01** | EXOKAY | Exclusive access success | Bus-width aligned exclusive | 4-byte aligned exclusive |
| **2’b10** | SLVERR | Slave error | Slave-specific error | Slave-specific error |
| **2’b11** | DECERR | Decode error | **Bus-width misalignment** | **4-byte misalignment** |

### Implementation Benefits

#### Simplified Mode Benefits

| Benefit Area | Simplification | Impact |
| --- | --- | --- |
| **Address Generation** | Simple increment by bus width | Minimal logic complexity |
| **Size Checking** | No dynamic size validation | No validation logic needed |
| **Strobe Generation** | All strobes always high | Trivial implementation |
| **Timing** | Predictable single-size transfers | Optimal timing closure |

#### Flexible Mode Benefits

| Benefit Area | Capability | Impact |
| --- | --- | --- |
| **Data Placement** | Arbitrary 4-byte aligned placement | Maximum flexibility |
| **Bus Utilization** | Progressive alignment optimization | High efficiency achieved |
| **Chunk Control** | Precise byte-level validity | Optimal memory utilization |
| **Alignment Strategy** | Automatic alignment to boundaries | Performance optimization |

#### Mode Selection Guidelines

| Interface Type | Recommended Mode | Rationale |
| --- | --- | --- |
| **High-bandwidth data** | **Flexible** | Maximize throughput, handle arbitrary alignment |
| **Control/status** | **Simplified** | Predictable timing, minimal complexity |
| **Descriptors** | **Simplified** | Fixed-size structures, simple implementation |
| **Programs** | **Simplified** | Single-word writes, minimal overhead |
| **Flags** | **Simplified** | Fixed-size status, predictable behavior |

### Validation Requirements

#### Simplified Mode Validation

| Validation Area | Requirements |
| --- | --- |
| **Address Alignment** | Verify all addresses aligned to full bus width |
| **Fixed Size** | Verify AxSIZE always matches DATA\_WIDTH |
| **Full Strobes** | Verify w\_strb is always all 1’s |
| **Burst Type** | Verify AxBURST is always 2’b01 |

#### Flexible Mode Validation

| Validation Area | Requirements |
| --- | --- |
| **Address Alignment** | Verify all addresses are 4-byte aligned |
| **Size Validation** | Verify AxSIZE matches actual transfer size |
| **Chunk Consistency** | Verify chunk enables match transfer size |
| **Strobe Generation** | Verify strobes generated correctly from chunks |
| **Progressive Alignment** | Verify alignment strategy optimization |
| **Boundary Checking** | Verify no 4KB boundary crossings |

#### Common Validation

| Validation Area | Requirements |
| --- | --- |
| **No Wraparound** | Verify addresses never wrap around |
| **Incrementing Only** | Verify AxBURST is always 2’b01 |
| **Response Handling** | Verify proper response generation |
| **Error Conditions** | Verify alignment violation responses |

### Performance Characteristics

#### Simplified Mode Performance

| Metric | Typical Value | Description |
| --- | --- | --- |
| **Latency** | 3 cycles | Address + Data + Response |
| **Throughput** | 1 transfer per clock | Sustained rate |
| **Efficiency** | 100% | Perfect bus utilization |
| **Complexity** | Minimal | Simple implementation |

#### Flexible Mode Performance

| Metric | Alignment Phase | Optimized Phase | Description |
| --- | --- | --- | --- |
| **Latency** | 3-15 cycles | 3 cycles | Variable based on alignment |
| **Throughput** | Variable | 1 transfer per clock | Depends on alignment pattern |
| **Efficiency** | 25-100% | 100% | Improves with alignment |
| **Complexity** | Moderate | Minimal | Progressive optimization |

#### Performance Optimization Strategy

**Flexible Mode Alignment Strategy:** 1. **Initial Phase**: Use largest possible transfer size for current alignment 2. **Progressive Phase**: Incrementally align to larger boundaries  
3. **Optimized Phase**: Use full bus-width transfers once aligned 4. **Result**: Achieve maximum efficiency while handling arbitrary starting addresses

This dual-mode approach provides the best of both worlds: simplified, predictable operation for control interfaces and flexible, high-performance operation for data interfaces. ## AXI4-Stream (AXIS4) Interface Specification and Assumptions

### Overview

This document defines the formal specification and assumptions for AXI4-Stream (AXIS4) interface implementations used in the RAPIDS system. AXIS4 provides high-bandwidth, unidirectional streaming data transfer with built-in flow control and packet framing capabilities.

### Interface Summary

#### Number of Interfaces

* **2 Master (Transmit) Interfaces**: Source data path and network master output
* **2 Slave (Receive) Interfaces**: Sink data path and network slave input

#### Interface Parameters

| Parameter | Description | Valid Values | Default |
| --- | --- | --- | --- |
| TDATA\_WIDTH | Stream data bus width in bits | 32, 64, 128, 256, 512, 1024 | 512 |
| TID\_WIDTH | Stream ID width in bits (optional) | 0-8 | 0 |
| TDEST\_WIDTH | Stream destination width in bits (optional) | 0-8 | 0 |
| TUSER\_WIDTH | User-defined sideband width in bits (optional) | 0-128 | 0 |
| TKEEP\_ENABLE | Enable TKEEP byte qualifier signals | 0, 1 | 1 |
| TSTRB\_ENABLE | Enable TSTRB byte strobe signals | 0, 1 | 0 |

#### Interface Configuration Summary

| Interface Type | Direction | TDATA Width | TID | TDEST | TUSER | TKEEP | Purpose |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **Network Master (TX)** | Output | 512 bits | No | Optional | Optional | Yes | Transmit packets to network |
| **Network Slave (RX)** | Input | 512 bits | No | Optional | Optional | Yes | Receive packets from network |
| **Source Data Stream** | Output | 512 bits | No | No | Optional | Yes | Memory-to-network streaming |
| **Sink Data Stream** | Input | 512 bits | No | No | Optional | Yes | Network-to-memory streaming |

### Core Protocol Assumptions

#### AXI4-Stream Fundamentals

| Aspect | Requirement |
| --- | --- |
| **Transfer Protocol** | Valid-ready handshake on every transfer |
| **Data Flow** | Unidirectional streaming (master -> slave) |
| **Packet Framing** | TLAST signal indicates packet boundaries |
| **Flow Control** | Backpressure via TREADY signal |
| **Byte Granularity** | TKEEP indicates valid bytes within transfer |

#### Implementation Assumptions

##### Assumption 1: TKEEP-Based Byte Qualification

| Aspect | Requirement |
| --- | --- |
| **Byte Validity Rule** | TKEEP indicates which bytes of TDATA contain valid data |
| **512-bit bus (64 bytes)** | TKEEP[63:0] - one bit per byte |
| **Contiguous Bytes** | Valid bytes are always contiguous (no gaps) |
| **Alignment** | Valid bytes always start from TDATA[7:0] (byte 0) |
| **Rationale** | Simplifies data alignment and reduces complexity |
| **Benefit** | Eliminates data steering logic for non-contiguous bytes |

**TKEEP Encoding Examples (64-byte bus):**

| Transfer Size | TKEEP Value | Description |
| --- | --- | --- |
| **64 bytes** | 64’hFFFFFFFFFFFFFFFF | All bytes valid (full transfer) |
| **32 bytes** | 64’h00000000FFFFFFFF | First 32 bytes valid |
| **16 bytes** | 64’h000000000000FFFF | First 16 bytes valid |
| **8 bytes** | 64’h00000000000000FF | First 8 bytes valid |
| **4 bytes** | 64’h000000000000000F | First 4 bytes valid |
| **1 byte** | 64’h0000000000000001 | First byte valid |

##### Assumption 2: TLAST for Packet Boundaries

| Aspect | Requirement |
| --- | --- |
| **Packet Delimiter** | TLAST=1 on final transfer of packet |
| **Single-Beat Packets** | TLAST=1 for single-transfer packets |
| **Multi-Beat Packets** | TLAST=0 for all transfers except last |
| **Mandatory Signal** | TLAST required for all packet-based protocols |
| **Rationale** | Enables downstream packet processing and buffering |

##### Assumption 3: No TSTRB Usage

| Aspect | Requirement |
| --- | --- |
| **TSTRB Disabled** | TSTRB signals not used (TSTRB\_ENABLE=0) |
| **Byte Qualification** | TKEEP provides sufficient byte-level control |
| **Rationale** | RAPIDS data is always read-oriented (no write strobes needed) |
| **Benefit** | Reduces interface width and complexity |

##### Assumption 4: Optional TID and TDEST

| Aspect | Requirement |
| --- | --- |
| **TID Usage** | Transaction ID not required for RAPIDS use cases |
| **TDEST Usage** | Destination routing optional (set by interface type) |
| **Default Configuration** | TID\_WIDTH=0, TDEST\_WIDTH=0 for simple streaming |
| **Rationale** | RAPIDS uses point-to-point connections, no routing needed |

### Master (Transmit) Interface Specification

#### AXIS Master Signals

| Signal | Width | Direction | Required | Description |
| --- | --- | --- | --- | --- |
| m\_axis\_tdata | TDATA\_WIDTH | Master->Slave | Yes | Stream data payload |
| m\_axis\_tvalid | 1 | Master->Slave | Yes | Data valid indicator |
| m\_axis\_tready | 1 | Slave->Master | Yes | Ready for data (backpressure) |
| m\_axis\_tlast | 1 | Master->Slave | Yes | Last transfer in packet |
| m\_axis\_tkeep | TDATA\_WIDTH/8 | Master->Slave | Yes | Byte qualifier (valid bytes) |
| m\_axis\_tid | TID\_WIDTH | Master->Slave | Optional | Transaction ID |
| m\_axis\_tdest | TDEST\_WIDTH | Master->Slave | Optional | Routing destination |
| m\_axis\_tuser | TUSER\_WIDTH | Master->Slave | Optional | User sideband data |

#### Master Transfer Rules

| Rule | Requirement | Description |
| --- | --- | --- |
| **Transfer Occurrence** | Transfer occurs when TVALID=1 AND TREADY=1 | Standard AXIS handshake |
| **TVALID Assertion** | Master can assert TVALID independently of TREADY | Master controls data availability |
| **TVALID Stability** | Once TVALID=1, all T\* signals must remain stable until TREADY=1 | Data integrity requirement |
| **TREADY Dependency** | Slave can assert TREADY based on TVALID state | Backpressure control |
| **TKEEP Alignment** | Valid bytes start from byte 0, must be contiguous | Per Assumption 1 |
| **TLAST Requirement** | TLAST=1 on final beat of every packet | Per Assumption 2 |

### Slave (Receive) Interface Specification

#### AXIS Slave Signals

| Signal | Width | Direction | Required | Description |
| --- | --- | --- | --- | --- |
| s\_axis\_tdata | TDATA\_WIDTH | Master->Slave | Yes | Stream data payload |
| s\_axis\_tvalid | 1 | Master->Slave | Yes | Data valid indicator |
| s\_axis\_tready | 1 | Slave->Master | Yes | Ready for data (backpressure) |
| s\_axis\_tlast | 1 | Master->Slave | Yes | Last transfer in packet |
| s\_axis\_tkeep | TDATA\_WIDTH/8 | Master->Slave | Yes | Byte qualifier (valid bytes) |
| s\_axis\_tid | TID\_WIDTH | Master->Slave | Optional | Transaction ID |
| s\_axis\_tdest | TDEST\_WIDTH | Master->Slave | Optional | Routing destination |
| s\_axis\_tuser | TUSER\_WIDTH | Master->Slave | Optional | User sideband data |

#### Slave Flow Control

| Aspect | Behavior | Description |
| --- | --- | --- |
| **TREADY=1** | Slave ready to accept data | Normal operation |
| **TREADY=0** | Slave cannot accept data (backpressure) | Flow control active |
| **TREADY Timing** | Can be asserted/deasserted on any cycle | Dynamic flow control |
| **Buffer Management** | TREADY reflects downstream buffer availability | Prevents overflow |

### Packet Structure and Framing

#### Single-Beat Packet

**Packet with data <= 64 bytes (fits in one transfer):**

Clock: -+ +-+ +-  
 +-+ +-+  
  
TVALID: --+ +---  
 +-+  
  
TREADY: --------  
 (always ready)  
  
TDATA: [Packet Data (64 bytes max)]  
  
TKEEP: [Valid byte mask (e.g., 64'h00000000FFFFFFFF for 32 bytes)]  
  
TLAST: --+ +---  
 +-+  
  
Transfer: Single beat completes entire packet

#### Multi-Beat Packet

**Packet with data > 64 bytes (requires multiple transfers):**

Clock: -+ +-+ +-+ +-+ +-  
 +-+ +-+ +-+ +-+  
  
TVALID: --+ +---  
 +---------+  
  
TREADY: ----------------  
 (always ready)  
  
TDATA: [Beat 0][Beat 1][Beat 2][Beat 3]  
 64B 64B 64B 32B (partial)  
  
TKEEP: [64'hFFFF...][64'hFFFF...][64'hFFFF...][64'h00000000FFFFFFFF]  
 All valid All valid All valid 32 bytes valid  
  
TLAST: --------------+ +---  
 +-+  
  
Transfer: 4 beats (256 bytes total)  
- Beats 0-2: Full 64-byte transfers, TLAST=0  
- Beat 3: Partial 32-byte transfer, TLAST=1

#### Packet with Backpressure

**Backpressure applied mid-packet:**

Clock: -+ +-+ +-+ +-+ +-+ +-+ +-  
 +-+ +-+ +-+ +-+ +-+ +-+  
  
TVALID: --+ +---  
 +-------------------+  
  
TREADY: --+ +-+ +---+ +-----  
 +-+ +-+ +-+  
  
TDATA: [Beat 0] (stall) [Beat 1] (stall) [Beat 2]  
  
TKEEP: [Valid] (held) [Valid] (held) [Valid]  
  
TLAST: --------------------------+ +---  
 +-+  
  
Transfer: 3 beats with backpressure  
- Beat 0: Transfers immediately  
- Stall: TREADY=0, TVALID held high, data held stable  
- Beat 1: Transfers after 2 stall cycles  
- Stall: Another 1-cycle stall  
- Beat 2: Final beat transfers, TLAST=1

**Key Observations:** 1. Master must hold TVALID=1 and all T\* signals stable during backpressure 2. Slave controls flow via TREADY signal 3. Transfer only occurs when both TVALID=1 AND TREADY=1

### TUSER Sideband Signaling

#### Purpose and Usage

| Aspect | Description |
| --- | --- |
| **Purpose** | Carry packet metadata alongside data stream |
| **Scope** | Valid on first beat of packet only (when new packet starts) |
| **Width** | Application-specific (0-128 bits) |
| **Examples** | Packet type, priority, timestamp, sequence number |

#### RAPIDS-Specific TUSER Encoding (Example)

**64-bit TUSER encoding for network packets:**

| Bits | Field | Description |
| --- | --- | --- |
| [7:0] | **Packet Type** | 0x01=Data, 0x02=Control, 0x03=Status |
| [15:8] | **Priority** | 0-255 priority level |
| [31:16] | **Sequence Number** | Per-channel sequence tracking |
| [47:32] | **Channel ID** | Source/destination channel identifier |
| [63:48] | **Reserved** | Future use |

**Usage Pattern:** - TUSER valid only on first beat of packet (can be ignored on subsequent beats) - Downstream logic captures TUSER on first beat, uses for entire packet - Simplifies metadata handling (no per-beat metadata processing)

### Flow Control and Backpressure

#### Backpressure Mechanisms

| Mechanism | Implementation | Description |
| --- | --- | --- |
| **Immediate Backpressure** | TREADY=0 for N cycles | Slave cannot accept data |
| **Conditional Backpressure** | TREADY toggles based on buffer state | Dynamic flow control |
| **Sustained Backpressure** | TREADY=0 for extended period | Upstream buffer fills |

#### Backpressure Propagation

Source SRAM Control -> AXIS Master -> AXIS Slave -> Sink SRAM Control  
 ↑ ↓  
 TREADY Backpressure  
 (flow control)

**Propagation Rules:** 1. Slave asserts TREADY=0 when downstream buffer nearly full 2. Master stops sending data (TVALID may remain high, data held) 3. Backpressure propagates to source (SRAM read stalls) 4. System self-regulates to prevent buffer overflow

#### Buffer Depth Considerations

| Buffer Type | Recommended Depth | Rationale |
| --- | --- | --- |
| **AXIS Input FIFO** | 16-32 beats | Absorb backpressure latency |
| **AXIS Output FIFO** | 16-32 beats | Smooth bursty traffic |
| **Packet Buffer** | 4-8 packets | Handle multi-packet scenarios |

### Reset Behavior

#### Reset Requirements

| Reset Phase | Requirement | Description |
| --- | --- | --- |
| **Active Reset** | aresetn is active-low reset signal | Standard AXI reset |
| **TVALID During Reset** | TVALID=0 when aresetn=0 | No spurious transfers |
| **TREADY During Reset** | TREADY can be 0 or 1 (don’t care) | Slave state undefined |
| **State Clearing** | All FIFOs/buffers flushed during reset | Clean startup |
| **Post-Reset** | TVALID=0 for at least 1 cycle after reset release | Stable initialization |

#### Reset Timing

Clock: -+ +-+ +-+ +-+ +-+ +-+ +-  
 +-+ +-+ +-+ +-+ +-+ +-+  
  
aresetn: ------+ +-----  
 +-----------+  
  
TVALID: ------------------+ +---  
 +-+  
 (stable after reset)  
  
TREADY: ------------------? ? ?--  
 (don't care initially)  
  
Note: TVALID must be 0 during and immediately after reset

### Implementation Benefits

#### Streaming Efficiency

| Benefit Area | Advantage | Impact |
| --- | --- | --- |
| **Continuous Streaming** | No address overhead (unlike AXI4) | Maximum throughput |
| **Simple Handshake** | Valid-ready protocol only | Minimal control logic |
| **Burst Transfers** | Multi-beat packets for efficiency | High bandwidth utilization |
| **Flow Control** | Built-in backpressure mechanism | Prevents data loss |

#### Packet-Based Processing

| Benefit Area | Advantage | Impact |
| --- | --- | --- |
| **Packet Framing** | TLAST delimits packets | Enables packet-level processing |
| **Byte Granularity** | TKEEP handles partial transfers | Supports variable-length packets |
| **Metadata Support** | TUSER carries packet info | Rich packet classification |

#### Resource Efficiency

| Benefit Area | Simplification | Impact |
| --- | --- | --- |
| **No Address Logic** | Streaming-only interface | Reduced logic complexity |
| **No Transaction IDs** | Point-to-point connections | Simplified state machines |
| **Optional Signals** | TID/TDEST/TUSER as needed | Minimal interface width |
| **Byte Alignment** | Contiguous bytes only | No complex data steering |

### Timing Requirements

#### Setup and Hold Times

| Timing Parameter | Requirement | Description |
| --- | --- | --- |
| **TVALID to TREADY** | Setup time: 0 ns | Combinational ready allowed |
| **TREADY to TVALID** | Setup time: 1 clock cycle | TVALID registered before TREADY check |
| **T\* Signal Stability** | Hold until TREADY=1 | Data integrity during backpressure |

#### Clock Domain Considerations

| Scenario | Requirement | Solution |
| --- | --- | --- |
| **Synchronous Operation** | Single clock domain | Direct connection |
| **Asynchronous Operation** | Different clock domains | Insert AXIS async FIFO |
| **Clock Frequency Ratio** | Producer faster than consumer | Backpressure handles rate mismatch |

### Validation Requirements

#### Functional Validation

| Validation Area | Requirements |
| --- | --- |
| **Valid-Ready Handshake** | Verify all transfers occur only when TVALID=1 AND TREADY=1 |
| **TKEEP Encoding** | Verify byte validity matches TKEEP pattern |
| **TLAST Assertion** | Verify TLAST=1 on final beat of every packet |
| **Backpressure Handling** | Verify master holds TVALID and T\* signals during TREADY=0 |
| **Packet Integrity** | Verify multi-beat packets reconstruct correctly |
| **Byte Alignment** | Verify valid bytes are contiguous starting from byte 0 |

#### Timing Validation

| Validation Area | Requirements |
| --- | --- |
| **Signal Stability** | Verify T\* signals stable when TVALID=1 until TREADY=1 |
| **Reset Behavior** | Verify TVALID=0 during and after reset |
| **Clock Crossing** | Verify async FIFO metastability protection (if used) |

#### Stress Testing

| Test Type | Description | Expected Behavior |
| --- | --- | --- |
| **Sustained Backpressure** | TREADY=0 for 100+ cycles | Master waits without data corruption |
| **Rapid Backpressure Toggle** | TREADY toggles every cycle | Transfer rate adapts correctly |
| **Maximum Throughput** | TREADY=1 always | Full bandwidth utilization |
| **Single-Beat Packets** | All packets fit in one beat | TLAST=1 on every transfer |
| **Large Multi-Beat Packets** | Packets spanning 100+ beats | Correct packet reconstruction |

### Example Transactions

#### Example 1: Single-Beat Packet (32 bytes)

**Configuration:** - TDATA\_WIDTH = 512 bits (64 bytes) - Packet size = 32 bytes - Single transfer

**Signals:**

Clock cycle: -+ +-  
 +-+  
  
TVALID: --+ +---  
 +-+  
  
TREADY: ----------  
 (ready)  
  
TDATA: [32 bytes of packet data | 32 bytes unused]  
  
TKEEP: 64'h00000000FFFFFFFF (first 32 bytes valid)  
  
TLAST: --+ +---  
 +-+  
  
TUSER: [Packet metadata - type, priority, etc.]

**Result:** Entire packet transfers in single beat.

#### Example 2: Multi-Beat Packet with Backpressure (200 bytes)

**Configuration:** - TDATA\_WIDTH = 512 bits (64 bytes) - Packet size = 200 bytes - Requires 4 beats: 64 + 64 + 64 + 8 bytes

**Signals:**

Clock cycle: -+ +-+ +-+ +-+ +-+ +-+ +-  
 +-+ +-+ +-+ +-+ +-+ +-+  
  
TVALID: --+ +---  
 +-----------------------+  
  
TREADY: --+ +-----+ +-+ +-+ +---  
 +-+ +-+ +-+ +-+  
 (backpressure cycles 2-3, 5)  
  
Beat 0: [64 bytes] TKEEP=64'hFFFF..., TLAST=0  
 Transfers immediately  
  
Stall: (cycles 2-3, TREADY=0, master holds data)  
  
Beat 1: [64 bytes] TKEEP=64'hFFFF..., TLAST=0  
 Transfers after stall  
  
Stall: (cycle 5, TREADY=0 again)  
  
Beat 2: [64 bytes] TKEEP=64'hFFFF..., TLAST=0  
 Transfers after stall  
  
Beat 3: [8 bytes] TKEEP=64'h00000000000000FF, TLAST=1  
 Final beat transfers  
  
TUSER: [Valid on beat 0 only]

**Result:** 200-byte packet transfers across 4 beats with intermittent backpressure. Total transfer takes 7 clock cycles (4 beats + 3 stall cycles).

#### Example 3: Back-to-Back Packets

**Configuration:** - Two packets: Packet A (64 bytes), Packet B (128 bytes) - No gaps between packets

**Signals:**

Clock cycle: -+ +-+ +-+ +-  
 +-+ +-+ +-+  
  
TVALID: --+ +---  
 +---------+  
  
TREADY: ----------------  
  
Beat 0: [Packet A - 64 bytes] TKEEP=64'hFFFF..., TLAST=1  
 TUSER=[Packet A metadata]  
  
Beat 1: [Packet B beat 0 - 64 bytes] TKEEP=64'hFFFF..., TLAST=0  
 TUSER=[Packet B metadata]  
  
Beat 2: [Packet B beat 1 - 64 bytes] TKEEP=64'hFFFF..., TLAST=1  
 TUSER=(ignored, mid-packet)

**Result:** Back-to-back packets transfer efficiently without idle cycles. TUSER updates on first beat of each new packet.

### Common Use Cases

#### Network Interface Applications

| Use Case | Configuration | Description |
| --- | --- | --- |
| **Packet Transmission** | 512-bit TDATA, TKEEP, TLAST, TUSER | High-bandwidth network TX |
| **Packet Reception** | 512-bit TDATA, TKEEP, TLAST, TUSER | High-bandwidth network RX |
| **Streaming DMA** | 512-bit TDATA, TKEEP, TLAST | Memory-to-network data transfer |
| **Flow-Controlled Streaming** | Dynamic TREADY | Backpressure-aware streaming |

#### Data Path Integration

| Integration Pattern | Description |
| --- | --- |
| **Source Path** | AXI4 Read -> SRAM -> AXIS Master -> Network |
| **Sink Path** | Network -> AXIS Slave -> SRAM -> AXI4 Write |
| **Loopback** | AXIS Master -> AXIS Slave (testing) |
| **Multi-Stage Pipeline** | AXIS -> Processing -> AXIS (chained) |

#### Performance Characteristics

| Metric | Typical Value | Description |
| --- | --- | --- |
| **Latency** | 1-2 cycles | TVALID assertion to TREADY response |
| **Throughput** | 1 beat per clock | Sustained rate (no backpressure) |
| **Efficiency** | 95-100% | With occasional backpressure |
| **Packet Rate** | Dependent on size | 64-byte packets: ~10Gbps @ 200MHz |

### RAPIDS-Specific Considerations

#### Interface Assignments

| RAPIDS Block | AXIS Role | Direction | Width | Notes |
| --- | --- | --- | --- | --- |
| **Network Slave** | Slave (RX) | Input | 512-bit | Receives packets from network |
| **Network Master** | Master (TX) | Output | 512-bit | Transmits packets to network |
| **Source SRAM Control** | Master (TX) | Output | 512-bit | Streams data from SRAM |
| **Sink SRAM Control** | Slave (RX) | Input | 512-bit | Receives data to SRAM |

#### Packet Processing Flow

**Sink Path (Network -> Memory):**

Network AXIS Input (512-bit packets)  
 ↓ TVALID/TREADY/TLAST/TKEEP  
Network Slave (packet validation)  
 ↓ Internal handshake  
Sink SRAM Control (buffering)  
 ↓ AXI4 Write  
System Memory

**Source Path (Memory -> Network):**

System Memory  
 ↓ AXI4 Read  
Source SRAM Control (buffering)  
 ↓ Internal handshake  
Network Master (packet formation)  
 ↓ TVALID/TREADY/TLAST/TKEEP  
Network AXIS Output (512-bit packets)

#### Buffer Sizing Guidelines

| Buffer Location | Recommended Size | Rationale |
| --- | --- | --- |
| **Network Input FIFO** | 32 beats (2KB) | Absorb network burst traffic |
| **Network Output FIFO** | 32 beats (2KB) | Smooth AXI4 read latency |
| **SRAM Depth** | 1024-4096 entries | Match typical packet sizes |

### Comparison with Other AXIS Variants

#### AXIS vs Full AXI4

| Feature | AXIS | Full AXI4 |
| --- | --- | --- |
| **Addressing** | No addressing (streaming) | Full address bus |
| **Channels** | Single data channel | 5 independent channels (AR, R, AW, W, B) |
| **Transaction IDs** | Optional (rarely used) | Mandatory for out-of-order |
| **Burst Support** | Continuous streaming | Fixed-length bursts |
| **Complexity** | Low | High |
| **Use Case** | Streaming data | Random-access memory |

#### AXIS Configuration Trade-offs

| Configuration | Advantages | Disadvantages |
| --- | --- | --- |
| **Wide Data Path (512-bit)** | High throughput, fewer transfers | More routing resources |
| **Narrow Data Path (64-bit)** | Simpler routing, lower resource | Lower throughput, more transfers |
| **With TUSER** | Rich metadata support | Increased interface width |
| **Without TUSER** | Minimal interface | Limited metadata capability |

### Appendix: Signal Quick Reference

#### Mandatory Signals

| Signal | Width | Source | Description |
| --- | --- | --- | --- |
| TDATA | TDATA\_WIDTH | Master | Streaming data payload |
| TVALID | 1 | Master | Data valid indicator |
| TREADY | 1 | Slave | Ready for data (backpressure) |
| TLAST | 1 | Master | Last transfer in packet |

#### Optional Signals

| Signal | Width | Source | When to Use |
| --- | --- | --- | --- |
| TKEEP | TDATA\_WIDTH/8 | Master | Byte-level data validity (recommended) |
| TSTRB | TDATA\_WIDTH/8 | Master | Write strobes (rarely used) |
| TID | TID\_WIDTH | Master | Transaction routing/identification |
| TDEST | TDEST\_WIDTH | Master | Destination routing |
| TUSER | TUSER\_WIDTH | Master | User-defined sideband metadata |

#### Signal Relationships

TVALID=1 + TREADY=1 -> Transfer occurs  
TVALID=1 + TREADY=0 -> Master waits (holds all T\* signals)  
TVALID=0 + TREADY=? -> No transfer (TREADY don't care)  
TLAST=1 -> Final beat of packet  
TKEEP[n]=1 -> Byte n of TDATA is valid  
TKEEP[n]=0 -> Byte n of TDATA is invalid (ignored)

**Next:** [Chapter 3 - Interface 5: MonBus](05_monbus_interface_spec.md) ## Monitor Bus Architecture and Event Code Organization

### Overview

The Monitor Bus architecture provides a unified, scalable framework for monitoring and error reporting across multiple bus protocols in complex SoC designs. This system supports AXI, APB, Network (Mesh Network on Chip), ARB (Arbiter), CORE, and custom protocols through a standardized 64-bit packet format with protocol-aware event categorization.

### Interface Summary

#### Number of Interfaces

* **1 Monitor Bus Output Interface**: Unified 64-bit packet stream
* **Multiple Protocol Input Interfaces**: AXI, APB, Network, ARB, CORE, Custom protocol monitors
* **Local Memory Interface**: Error/interrupt packet storage
* **External Memory Interface**: Bulk packet storage

#### Interface Parameters

| Parameter | Description | Valid Values | Default |
| --- | --- | --- | --- |
| PACKET\_WIDTH | Monitor bus packet width | 64 | 64 |
| PROTOCOL\_WIDTH | Protocol identifier width | 3 | 3 |
| EVENT\_CODE\_WIDTH | Event code width | 4 | 4 |
| PACKET\_TYPE\_WIDTH | Packet type width | 4 | 4 |
| CHANNEL\_ID\_WIDTH | Channel identifier width | 6 | 6 |
| UNIT\_ID\_WIDTH | Unit identifier width | 4 | 4 |
| AGENT\_ID\_WIDTH | Agent identifier width | 8 | 8 |
| EVENT\_DATA\_WIDTH | Event data width | 35 | 35 |

### Core Design Assumptions

#### Assumption 1: Hierarchical Event Organization

| Aspect | Requirement |
| --- | --- |
| **Organization Rule** | Protocol -> Packet Type -> Event Code hierarchy |
| **Event Space** | Each protocol x packet type combination has exactly 16 event codes |
| **Mapping** | 1:1 mapping between packet types and event codes |
| **Rationale** | Provides clear, scalable event organization |

#### Assumption 2: Protocol Isolation

| Aspect | Requirement |
| --- | --- |
| **Isolation Rule** | Each protocol owns its event space |
| **Conflict Prevention** | No cross-protocol event conflicts |
| **Independent Evolution** | Protocols can evolve independently |
| **Rationale** | Prevents interference and enables protocol-specific optimization |

#### Assumption 3: Two-Tier Memory Architecture

| Aspect | Requirement |
| --- | --- |
| **Local Storage** | Critical events (errors/interrupts) stored locally |
| **External Storage** | Non-critical events routed to external memory |
| **Routing Decision** | Based on packet type configuration |
| **Rationale** | Balances immediate access with bulk storage needs |

#### Assumption 4: Configurable Packet Routing

| Aspect | Requirement |
| --- | --- |
| **Routing Rule** | Different packet types can route to different destinations |
| **Configuration** | Base/limit registers define routing per packet type |
| **Priority Support** | Configurable priority levels per packet type |
| **Rationale** | Enables flexible memory allocation and access patterns |

### Interface Signal Specification

#### Monitor Bus Output Interface

| Signal | Width | Direction | Description |
| --- | --- | --- | --- |
| mon\_packet | 64 | Monitor->System | Monitor packet data |
| mon\_valid | 1 | Monitor->System | Packet valid signal |
| mon\_ready | 1 | System->Monitor | Ready to accept packet |
| mon\_error | 1 | Monitor->System | Monitor error condition |

#### Protocol Input Interfaces

| Signal | Width | Direction | Description |
| --- | --- | --- | --- |
| axi\_event | 64 | AXI Monitor->Bus | AXI event packet |
| axi\_event\_valid | 1 | AXI Monitor->Bus | AXI event valid |
| axi\_event\_ready | 1 | Bus->AXI Monitor | Ready for AXI event |
| apb\_event | 64 | APB Monitor->Bus | APB event packet |
| apb\_event\_valid | 1 | APB Monitor->Bus | APB event valid |
| apb\_event\_ready | 1 | Bus->APB Monitor | Ready for APB event |
| network\_event | 64 | Network Monitor->Bus | Network event packet |
| network\_event\_valid | 1 | Network Monitor->Bus | Network event valid |
| network\_event\_ready | 1 | Bus->Network Monitor | Ready for Network event |
| arb\_event | 64 | ARB Monitor->Bus | ARB event packet |
| arb\_event\_valid | 1 | ARB Monitor->Bus | ARB event valid |
| arb\_event\_ready | 1 | Bus->ARB Monitor | Ready for ARB event |
| core\_event | 64 | CORE Monitor->Bus | CORE event packet |
| core\_event\_valid | 1 | CORE Monitor->Bus | CORE event valid |
| core\_event\_ready | 1 | Bus->CORE Monitor | Ready for CORE event |

#### Control and Status Signals

| Signal | Width | Direction | Description |
| --- | --- | --- | --- |
| clk | 1 | Input | System clock |
| resetn | 1 | Input | Active-low reset |
| monitor\_enable | 1 | Input | Global monitor enable |
| packet\_type\_enables | 16 | Input | Per-type enable bits |
| local\_memory\_full | 1 | Output | Local memory full flag |
| external\_memory\_error | 1 | Output | External memory error |

### Packet Format and Field Allocation

#### 64-bit Monitor Bus Packet Structure

| Field | Bits | Width | Description |
| --- | --- | --- | --- |
| **Packet Type** | [63:60] | 4 | Event category (Error, Completion, etc.) |
| **Protocol** | [59:57] | 3 | Bus protocol (AXI=0, Network=1, APB=2, ARB=3, CORE=4) |
| **Event Code** | [56:53] | 4 | Specific events within category |
| **Channel ID** | [52:47] | 6 | Transaction/channel identifier |
| **Unit ID** | [46:43] | 4 | Subsystem identifier |
| **Agent ID** | [42:35] | 8 | Module identifier |
| **Event Data** | [34:0] | 35 | Event-specific payload |

#### Packet Type Definitions

| Value | Name | Purpose | Applicable Protocols |
| --- | --- | --- | --- |
| **0x0** | Error | Protocol violations, response errors | All |
| **0x1** | Completion | Successful transaction completion | All |
| **0x2** | Threshold | Threshold crossed events | All |
| **0x3** | Timeout | Timeout conditions | All |
| **0x4** | Performance | Performance metrics | All |
| **0x5** | Credit | Credit management | Network only |
| **0x6** | Channel | Channel status | Network only |
| **0x7** | Stream | Stream events | Network only |
| **0x8** | Address Match | Address matching | AXI only |
| **0x9** | APB Specific | APB protocol events | APB only |
| **0xA-0xE** | Reserved | Future expansion | - |
| **0xF** | Debug | Debug and trace events | All |

### Protocol-Specific Event Codes

#### AXI Protocol Events

##### Error Events (PktTypeError + PROTOCOL\_AXI)

| Code | Event Name | Description |
| --- | --- | --- |
| **0x0** | AXI\_ERR\_RESP\_SLVERR | Slave error response |
| **0x1** | AXI\_ERR\_RESP\_DECERR | Decode error response |
| **0x2** | AXI\_ERR\_DATA\_ORPHAN | Data without command |
| **0x3** | AXI\_ERR\_RESP\_ORPHAN | Response without transaction |
| **0x4** | AXI\_ERR\_PROTOCOL | Protocol violation |
| **0x5** | AXI\_ERR\_BURST\_LENGTH | Invalid burst length |
| **0x6** | AXI\_ERR\_BURST\_SIZE | Invalid burst size |
| **0x7** | AXI\_ERR\_BURST\_TYPE | Invalid burst type |
| **0x8** | AXI\_ERR\_ID\_COLLISION | ID collision detected |
| **0x9** | AXI\_ERR\_WRITE\_BEFORE\_ADDR | Write data before address |
| **0xA** | AXI\_ERR\_RESP\_BEFORE\_DATA | Response before data complete |
| **0xB** | AXI\_ERR\_LAST\_MISSING | Missing LAST signal |
| **0xC** | AXI\_ERR\_STROBE\_ERROR | Write strobe error |
| **0xD** | AXI\_ERR\_RESERVED\_D | Reserved |
| **0xE** | AXI\_ERR\_RESERVED\_E | Reserved |
| **0xF** | AXI\_ERR\_USER\_DEFINED | User-defined error |

##### Timeout Events (PktTypeTimeout + PROTOCOL\_AXI)

| Code | Event Name | Description |
| --- | --- | --- |
| **0x0** | AXI\_TIMEOUT\_CMD | Command/Address timeout |
| **0x1** | AXI\_TIMEOUT\_DATA | Data timeout |
| **0x2** | AXI\_TIMEOUT\_RESP | Response timeout |
| **0x3** | AXI\_TIMEOUT\_HANDSHAKE | Handshake timeout |
| **0x4** | AXI\_TIMEOUT\_BURST | Burst completion timeout |
| **0x5** | AXI\_TIMEOUT\_EXCLUSIVE | Exclusive access timeout |
| **0x6-0xE** | Reserved | Future expansion |
| **0xF** | AXI\_TIMEOUT\_USER\_DEFINED | User-defined timeout |

##### Performance Events (PktTypePerf + PROTOCOL\_AXI)

| Code | Event Name | Description |
| --- | --- | --- |
| **0x0** | AXI\_PERF\_ADDR\_LATENCY | Address phase latency |
| **0x1** | AXI\_PERF\_DATA\_LATENCY | Data phase latency |
| **0x2** | AXI\_PERF\_RESP\_LATENCY | Response phase latency |
| **0x3** | AXI\_PERF\_TOTAL\_LATENCY | Total transaction latency |
| **0x4** | AXI\_PERF\_THROUGHPUT | Transaction throughput |
| **0x5** | AXI\_PERF\_ERROR\_RATE | Error rate |
| **0x6** | AXI\_PERF\_ACTIVE\_COUNT | Active transaction count |
| **0x7** | AXI\_PERF\_BANDWIDTH\_UTIL | Bandwidth utilization |
| **0x8** | AXI\_PERF\_QUEUE\_DEPTH | Average queue depth |
| **0x9** | AXI\_PERF\_BURST\_EFFICIENCY | Burst efficiency metric |
| **0xA-0xE** | Reserved | Future expansion |
| **0xF** | AXI\_PERF\_USER\_DEFINED | User-defined performance |

#### APB Protocol Events

##### Error Events (PktTypeError + PROTOCOL\_APB)

| Code | Event Name | Description |
| --- | --- | --- |
| **0x0** | APB\_ERR\_PSLVERR | Peripheral slave error |
| **0x1** | APB\_ERR\_SETUP\_VIOLATION | Setup phase protocol violation |
| **0x2** | APB\_ERR\_ACCESS\_VIOLATION | Access phase protocol violation |
| **0x3** | APB\_ERR\_STROBE\_ERROR | Write strobe error |
| **0x4** | APB\_ERR\_ADDR\_DECODE | Address decode error |
| **0x5** | APB\_ERR\_PROT\_VIOLATION | Protection violation (PPROT) |
| **0x6** | APB\_ERR\_ENABLE\_ERROR | Enable phase error |
| **0x7** | APB\_ERR\_READY\_ERROR | PREADY protocol error |
| **0x8-0xE** | Reserved | Future expansion |
| **0xF** | APB\_ERR\_USER\_DEFINED | User-defined error |

##### Timeout Events (PktTypeTimeout + PROTOCOL\_APB)

| Code | Event Name | Description |
| --- | --- | --- |
| **0x0** | APB\_TIMEOUT\_SETUP | Setup phase timeout |
| **0x1** | APB\_TIMEOUT\_ACCESS | Access phase timeout |
| **0x2** | APB\_TIMEOUT\_ENABLE | Enable phase timeout (PREADY stuck) |
| **0x3** | APB\_TIMEOUT\_PREADY\_STUCK | PREADY stuck low |
| **0x4** | APB\_TIMEOUT\_TRANSFER | Overall transfer timeout |
| **0x5-0xE** | Reserved | Future expansion |
| **0xF** | APB\_TIMEOUT\_USER\_DEFINED | User-defined timeout |

##### Completion Events (PktTypeCompletion + PROTOCOL\_APB)

| Code | Event Name | Description |
| --- | --- | --- |
| **0x0** | APB\_COMPL\_TRANS\_COMPLETE | Transaction completed |
| **0x1** | APB\_COMPL\_READ\_COMPLETE | Read transaction complete |
| **0x2** | APB\_COMPL\_WRITE\_COMPLETE | Write transaction complete |
| **0x3-0xE** | Reserved | Future expansion |
| **0xF** | APB\_COMPL\_USER\_DEFINED | User-defined completion |

##### Threshold Events (PktTypeThreshold + PROTOCOL\_APB)

| Code | Event Name | Description |
| --- | --- | --- |
| **0x0** | APB\_THRESH\_LATENCY | APB latency threshold |
| **0x1** | APB\_THRESH\_ERROR\_RATE | APB error rate threshold |
| **0x2** | APB\_THRESH\_ACCESS\_COUNT | Access count threshold |
| **0x3** | APB\_THRESH\_BANDWIDTH | Bandwidth threshold |
| **0x4-0xE** | Reserved | Future expansion |
| **0xF** | APB\_THRESH\_USER\_DEFINED | User-defined threshold |

##### Performance Events (PktTypePerf + PROTOCOL\_APB)

| Code | Event Name | Description |
| --- | --- | --- |
| **0x0** | APB\_PERF\_READ\_LATENCY | Read transaction latency |
| **0x1** | APB\_PERF\_WRITE\_LATENCY | Write transaction latency |
| **0x2** | APB\_PERF\_THROUGHPUT | Transaction throughput |
| **0x3** | APB\_PERF\_ERROR\_RATE | Error rate |
| **0x4** | APB\_PERF\_ACTIVE\_COUNT | Active transaction count |
| **0x5** | APB\_PERF\_COMPLETED\_COUNT | Completed transaction count |
| **0x6-0xE** | Reserved | Future expansion |
| **0xF** | APB\_PERF\_USER\_DEFINED | User-defined performance |

##### Debug Events (PktTypeDebug + PROTOCOL\_APB)

| Code | Event Name | Description |
| --- | --- | --- |
| **0x0** | APB\_DEBUG\_STATE\_CHANGE | APB state changed |
| **0x1** | APB\_DEBUG\_SETUP\_PHASE | Setup phase event |
| **0x2** | APB\_DEBUG\_ACCESS\_PHASE | Access phase event |
| **0x3** | APB\_DEBUG\_ENABLE\_PHASE | Enable phase event |
| **0x4** | APB\_DEBUG\_PSEL\_TRACE | PSEL trace |
| **0x5** | APB\_DEBUG\_PENABLE\_TRACE | PENABLE trace |
| **0x6** | APB\_DEBUG\_PREADY\_TRACE | PREADY trace |
| **0x7** | APB\_DEBUG\_PPROT\_TRACE | PPROT trace |
| **0x8** | APB\_DEBUG\_PSTRB\_TRACE | PSTRB trace |
| **0x9-0xE** | Reserved | Future expansion |
| **0xF** | APB\_DEBUG\_USER\_DEFINED | User-defined debug |

#### Network Protocol Events

##### Error Events (PktTypeError + PROTOCOL\_MNOC)

| Code | Event Name | Description |
| --- | --- | --- |
| **0x0** | NETWORK\_ERR\_PARITY | Parity error |
| **0x1** | NETWORK\_ERR\_PROTOCOL | Protocol violation |
| **0x2** | NETWORK\_ERR\_OVERFLOW | Buffer/Credit overflow |
| **0x3** | NETWORK\_ERR\_UNDERFLOW | Buffer/Credit underflow |
| **0x4** | NETWORK\_ERR\_ORPHAN | Orphaned packet/ACK |
| **0x5** | NETWORK\_ERR\_INVALID | Invalid type/channel/payload |
| **0x6** | NETWORK\_ERR\_HEADER\_CRC | Header CRC error |
| **0x7** | NETWORK\_ERR\_PAYLOAD\_CRC | Payload CRC error |
| **0x8** | NETWORK\_ERR\_SEQUENCE | Sequence number error |
| **0x9** | NETWORK\_ERR\_ROUTE | Routing error |
| **0xA** | NETWORK\_ERR\_DEADLOCK | Deadlock detected |
| **0xB-0xE** | Reserved | Future expansion |
| **0xF** | NETWORK\_ERR\_USER\_DEFINED | User-defined error |

##### Credit Events (PktTypeCredit + PROTOCOL\_MNOC)

| Code | Event Name | Description |
| --- | --- | --- |
| **0x0** | NETWORK\_CREDIT\_ALLOCATED | Credits allocated |
| **0x1** | NETWORK\_CREDIT\_CONSUMED | Credits consumed |
| **0x2** | NETWORK\_CREDIT\_RETURNED | Credits returned |
| **0x3** | NETWORK\_CREDIT\_OVERFLOW | Credit overflow detected |
| **0x4** | NETWORK\_CREDIT\_UNDERFLOW | Credit underflow detected |
| **0x5** | NETWORK\_CREDIT\_EXHAUSTED | All credits exhausted |
| **0x6** | NETWORK\_CREDIT\_RESTORED | Credits restored |
| **0x7** | NETWORK\_CREDIT\_EFFICIENCY | Credit efficiency metric |
| **0x8** | NETWORK\_CREDIT\_LEAK | Credit leak detected |
| **0x9-0xE** | Reserved | Future expansion |
| **0xF** | NETWORK\_CREDIT\_USER\_DEFINED | User-defined credit event |

##### Channel Events (PktTypeChannel + PROTOCOL\_MNOC)

| Code | Event Name | Description |
| --- | --- | --- |
| **0x0** | NETWORK\_CHANNEL\_OPEN | Channel opened |
| **0x1** | NETWORK\_CHANNEL\_CLOSE | Channel closed |
| **0x2** | NETWORK\_CHANNEL\_STALL | Channel stalled |
| **0x3** | NETWORK\_CHANNEL\_RESUME | Channel resumed |
| **0x4** | NETWORK\_CHANNEL\_CONGESTION | Channel congestion detected |
| **0x5** | NETWORK\_CHANNEL\_PRIORITY | Channel priority change |
| **0x6-0xE** | Reserved | Future expansion |
| **0xF** | NETWORK\_CHANNEL\_USER\_DEFINED | User-defined channel event |

##### Stream Events (PktTypeStream + PROTOCOL\_MNOC)

| Code | Event Name | Description |
| --- | --- | --- |
| **0x0** | NETWORK\_STREAM\_START | Stream started |
| **0x1** | NETWORK\_STREAM\_END | Stream ended (EOS) |
| **0x2** | NETWORK\_STREAM\_PAUSE | Stream paused |
| **0x3** | NETWORK\_STREAM\_RESUME | Stream resumed |
| **0x4** | NETWORK\_STREAM\_OVERFLOW | Stream buffer overflow |
| **0x5** | NETWORK\_STREAM\_UNDERFLOW | Stream buffer underflow |
| **0x6-0xE** | Reserved | Future expansion |
| **0xF** | NETWORK\_STREAM\_USER\_DEFINED | User-defined stream event |

#### ARB Protocol Events

##### Error Events (PktTypeError + PROTOCOL\_ARB)

| Code | Event Name | Description |
| --- | --- | --- |
| **0x0** | ARB\_ERR\_STARVATION | Client request starvation |
| **0x1** | ARB\_ERR\_ACK\_TIMEOUT | Grant ACK timeout |
| **0x2** | ARB\_ERR\_PROTOCOL\_VIOLATION | ACK protocol violation |
| **0x3** | ARB\_ERR\_CREDIT\_VIOLATION | Credit system violation |
| **0x4** | ARB\_ERR\_FAIRNESS\_VIOLATION | Weighted fairness violation |
| **0x5** | ARB\_ERR\_WEIGHT\_UNDERFLOW | Weight credit underflow |
| **0x6** | ARB\_ERR\_CONCURRENT\_GRANTS | Multiple simultaneous grants |
| **0x7** | ARB\_ERR\_INVALID\_GRANT\_ID | Invalid grant ID detected |
| **0x8** | ARB\_ERR\_ORPHAN\_ACK | ACK without pending grant |
| **0x9** | ARB\_ERR\_GRANT\_OVERLAP | Overlapping grant periods |
| **0xA** | ARB\_ERR\_MASK\_ERROR | Round-robin mask error |
| **0xB** | ARB\_ERR\_STATE\_MACHINE | FSM state error |
| **0xC** | ARB\_ERR\_CONFIGURATION | Invalid configuration |
| **0xD-0xE** | Reserved | Future expansion |
| **0xF** | ARB\_ERR\_USER\_DEFINED | User-defined error |

##### Timeout Events (PktTypeTimeout + PROTOCOL\_ARB)

| Code | Event Name | Description |
| --- | --- | --- |
| **0x0** | ARB\_TIMEOUT\_GRANT\_ACK | Grant ACK timeout |
| **0x1** | ARB\_TIMEOUT\_REQUEST\_HOLD | Request held too long |
| **0x2** | ARB\_TIMEOUT\_WEIGHT\_UPDATE | Weight update timeout |
| **0x3** | ARB\_TIMEOUT\_BLOCK\_RELEASE | Block release timeout |
| **0x4** | ARB\_TIMEOUT\_CREDIT\_UPDATE | Credit update timeout |
| **0x5** | ARB\_TIMEOUT\_STATE\_CHANGE | State machine timeout |
| **0x6-0xE** | Reserved | Future expansion |
| **0xF** | ARB\_TIMEOUT\_USER\_DEFINED | User-defined timeout |

##### Completion Events (PktTypeCompletion + PROTOCOL\_ARB)

| Code | Event Name | Description |
| --- | --- | --- |
| **0x0** | ARB\_COMPL\_GRANT\_ISSUED | Grant successfully issued |
| **0x1** | ARB\_COMPL\_ACK\_RECEIVED | ACK successfully received |
| **0x2** | ARB\_COMPL\_TRANSACTION | Complete transaction (grant+ack) |
| **0x3** | ARB\_COMPL\_WEIGHT\_UPDATE | Weight update completed |
| **0x4** | ARB\_COMPL\_CREDIT\_CYCLE | Credit cycle completed |
| **0x5** | ARB\_COMPL\_FAIRNESS\_PERIOD | Fairness analysis period |
| **0x6** | ARB\_COMPL\_BLOCK\_PERIOD | Block period completed |
| **0x7-0xE** | Reserved | Future expansion |
| **0xF** | ARB\_COMPL\_USER\_DEFINED | User-defined completion |

##### Threshold Events (PktTypeThreshold + PROTOCOL\_ARB)

| Code | Event Name | Description |
| --- | --- | --- |
| **0x0** | ARB\_THRESH\_REQUEST\_LATENCY | Request-to-grant latency threshold |
| **0x1** | ARB\_THRESH\_ACK\_LATENCY | Grant-to-ACK latency threshold |
| **0x2** | ARB\_THRESH\_FAIRNESS\_DEV | Fairness deviation threshold |
| **0x3** | ARB\_THRESH\_ACTIVE\_REQUESTS | Active request count threshold |
| **0x4** | ARB\_THRESH\_GRANT\_RATE | Grant rate threshold |
| **0x5** | ARB\_THRESH\_EFFICIENCY | Grant efficiency threshold |
| **0x6** | ARB\_THRESH\_CREDIT\_LOW | Low credit threshold |
| **0x7** | ARB\_THRESH\_WEIGHT\_IMBALANCE | Weight imbalance threshold |
| **0x8** | ARB\_THRESH\_STARVATION\_TIME | Starvation time threshold |
| **0x9-0xE** | Reserved | Future expansion |
| **0xF** | ARB\_THRESH\_USER\_DEFINED | User-defined threshold |

##### Performance Events (PktTypePerf + PROTOCOL\_ARB)

| Code | Event Name | Description |
| --- | --- | --- |
| **0x0** | ARB\_PERF\_GRANT\_ISSUED | Grant issued event |
| **0x1** | ARB\_PERF\_ACK\_RECEIVED | ACK received event |
| **0x2** | ARB\_PERF\_GRANT\_EFFICIENCY | Grant completion efficiency |
| **0x3** | ARB\_PERF\_FAIRNESS\_METRIC | Fairness compliance metric |
| **0x4** | ARB\_PERF\_THROUGHPUT | Arbitration throughput |
| **0x5** | ARB\_PERF\_LATENCY\_AVG | Average latency measurement |
| **0x6** | ARB\_PERF\_WEIGHT\_COMPLIANCE | Weight compliance metric |
| **0x7** | ARB\_PERF\_CREDIT\_UTILIZATION | Credit utilization efficiency |
| **0x8** | ARB\_PERF\_CLIENT\_ACTIVITY | Per-client activity metric |
| **0x9** | ARB\_PERF\_STARVATION\_COUNT | Starvation event count |
| **0xA** | ARB\_PERF\_BLOCK\_EFFICIENCY | Block/unblock efficiency |
| **0xB-0xE** | Reserved | Future expansion |
| **0xF** | ARB\_PERF\_USER\_DEFINED | User-defined performance |

##### Debug Events (PktTypeDebug + PROTOCOL\_ARB)

| Code | Event Name | Description |
| --- | --- | --- |
| **0x0** | ARB\_DEBUG\_STATE\_CHANGE | Arbiter state machine change |
| **0x1** | ARB\_DEBUG\_MASK\_UPDATE | Round-robin mask update |
| **0x2** | ARB\_DEBUG\_WEIGHT\_CHANGE | Weight configuration change |
| **0x3** | ARB\_DEBUG\_CREDIT\_UPDATE | Credit level update |
| **0x4** | ARB\_DEBUG\_CLIENT\_MASK | Client enable/disable mask |
| **0x5** | ARB\_DEBUG\_PRIORITY\_CHANGE | Priority level change |
| **0x6** | ARB\_DEBUG\_BLOCK\_EVENT | Block/unblock event |
| **0x7** | ARB\_DEBUG\_QUEUE\_STATUS | Request queue status |
| **0x8** | ARB\_DEBUG\_COUNTER\_SNAPSHOT | Counter values snapshot |
| **0x9** | ARB\_DEBUG\_FIFO\_STATUS | FIFO status change |
| **0xA** | ARB\_DEBUG\_FAIRNESS\_STATE | Fairness tracking state |
| **0xB** | ARB\_DEBUG\_ACK\_STATE | ACK protocol state |
| **0xC-0xE** | Reserved | Future expansion |
| **0xF** | ARB\_DEBUG\_USER\_DEFINED | User-defined debug |

#### CORE Protocol Events

##### Error Events (PktTypeError + PROTOCOL\_CORE)

| Code | Event Name | Description |
| --- | --- | --- |
| **0x0** | CORE\_ERR\_DESCRIPTOR\_MALFORMED | Missing magic number (0x900dc0de) |
| **0x1** | CORE\_ERR\_DESCRIPTOR\_BAD\_ADDR | Invalid descriptor address |
| **0x2** | CORE\_ERR\_DATA\_BAD\_ADDR | Invalid data address (fetch or runtime) |
| **0x3** | CORE\_ERR\_FLAG\_COMPARISON | Flag mask/compare mismatch |
| **0x4** | CORE\_ERR\_CREDIT\_UNDERFLOW | Credit system violation |
| **0x5** | CORE\_ERR\_STATE\_MACHINE | Invalid FSM state transition |
| **0x6** | CORE\_ERR\_DESCRIPTOR\_ENGINE | Descriptor engine FSM error |
| **0x7** | CORE\_ERR\_FLAG\_ENGINE | Flag engine FSM error |
| **0x8** | CORE\_ERR\_PROGRAM\_ENGINE | Program engine FSM error |
| **0x9** | CORE\_ERR\_DATA\_ENGINE | Data engine error |
| **0xA** | CORE\_ERR\_CHANNEL\_INVALID | Invalid channel ID |
| **0xB** | CORE\_ERR\_CONTROL\_VIOLATION | Control register violation |
| **0xC-0xE** | Reserved | Future expansion |
| **0xF** | CORE\_ERR\_USER\_DEFINED | User-defined error |

##### Timeout Events (PktTypeTimeout + PROTOCOL\_CORE)

| Code | Event Name | Description |
| --- | --- | --- |
| **0x0** | CORE\_TIMEOUT\_DESCRIPTOR\_FETCH | Descriptor fetch timeout |
| **0x1** | CORE\_TIMEOUT\_FLAG\_RETRY | Flag comparison retry timeout |
| **0x2** | CORE\_TIMEOUT\_PROGRAM\_WRITE | Program write timeout |
| **0x3** | CORE\_TIMEOUT\_DATA\_TRANSFER | Data transfer timeout |
| **0x4** | CORE\_TIMEOUT\_CREDIT\_WAIT | Credit wait timeout |
| **0x5** | CORE\_TIMEOUT\_CONTROL\_WAIT | Control enable wait timeout |
| **0x6** | CORE\_TIMEOUT\_ENGINE\_RESPONSE | Sub-engine response timeout |
| **0x7** | CORE\_TIMEOUT\_STATE\_TRANSITION | FSM state transition timeout |
| **0x8-0xE** | Reserved | Future expansion |
| **0xF** | CORE\_TIMEOUT\_USER\_DEFINED | User-defined timeout |

##### Completion Events (PktTypeCompletion + PROTOCOL\_CORE)

| Code | Event Name | Description |
| --- | --- | --- |
| **0x0** | CORE\_COMPL\_DESCRIPTOR\_LOADED | Descriptor successfully loaded |
| **0x1** | CORE\_COMPL\_DESCRIPTOR\_CHAIN | Descriptor chain completed |
| **0x2** | CORE\_COMPL\_FLAG\_MATCHED | Flag comparison successful |
| **0x3** | CORE\_COMPL\_PROGRAM\_COMPLETED | Post-programming completed |
| **0x4** | CORE\_COMPL\_DATA\_TRANSFER | Data transfer completed |
| **0x5** | CORE\_COMPL\_CREDIT\_CYCLE | Credit cycle completed |
| **0x6** | CORE\_COMPL\_CHANNEL\_COMPLETE | Channel processing complete |
| **0x7** | CORE\_COMPL\_ENGINE\_READY | Sub-engine ready |
| **0x8-0xE** | Reserved | Future expansion |
| **0xF** | CORE\_COMPL\_USER\_DEFINED | User-defined completion |

##### Threshold Events (PktTypeThreshold + PROTOCOL\_CORE)

| Code | Event Name | Description |
| --- | --- | --- |
| **0x0** | CORE\_THRESH\_DESCRIPTOR\_QUEUE | Descriptor queue depth threshold |
| **0x1** | CORE\_THRESH\_CREDIT\_LOW | Credit low threshold |
| **0x2** | CORE\_THRESH\_FLAG\_RETRY\_COUNT | Flag retry count threshold |
| **0x3** | CORE\_THRESH\_LATENCY | Processing latency threshold |
| **0x4** | CORE\_THRESH\_ERROR\_RATE | Error rate threshold |
| **0x5** | CORE\_THRESH\_THROUGHPUT | Throughput threshold |
| **0x6** | CORE\_THRESH\_ACTIVE\_CHANNELS | Active channel count threshold |
| **0x7** | CORE\_THRESH\_PROGRAM\_LATENCY | Program write latency threshold |
| **0x8** | CORE\_THRESH\_DATA\_RATE | Data transfer rate threshold |
| **0x9-0xE** | Reserved | Future expansion |
| **0xF** | CORE\_THRESH\_USER\_DEFINED | User-defined threshold |

##### Performance Events (PktTypePerf + PROTOCOL\_CORE)

| Code | Event Name | Description |
| --- | --- | --- |
| **0x0** | CORE\_PERF\_END\_OF\_DATA | Stream continuation signal |
| **0x1** | CORE\_PERF\_END\_OF\_STREAM | Stream termination signal |
| **0x2** | CORE\_PERF\_ENTERING\_IDLE | FSM returning to idle |
| **0x3** | CORE\_PERF\_CREDIT\_INCREMENTED | Credit added by software |
| **0x4** | CORE\_PERF\_CREDIT\_EXHAUSTED | Credit blocking execution |
| **0x5** | CORE\_PERF\_STATE\_TRANSITION | FSM state change |
| **0x6** | CORE\_PERF\_DESCRIPTOR\_ACTIVE | Data processing started |
| **0x7** | CORE\_PERF\_FLAG\_RETRY | Flag comparison retry |
| **0x8** | CORE\_PERF\_CHANNEL\_ENABLE | Channel enabled by software |
| **0x9** | CORE\_PERF\_CHANNEL\_DISABLE | Channel disabled by software |
| **0xA** | CORE\_PERF\_CREDIT\_UTILIZATION | Credit utilization metric |
| **0xB** | CORE\_PERF\_PROCESSING\_LATENCY | Total processing latency |
| **0xC** | CORE\_PERF\_QUEUE\_DEPTH | Current queue depth |
| **0xD-0xE** | Reserved | Future expansion |
| **0xF** | CORE\_PERF\_USER\_DEFINED | User-defined performance |

##### Debug Events (PktTypeDebug + PROTOCOL\_CORE)

| Code | Event Name | Description |
| --- | --- | --- |
| **0x0** | CORE\_DEBUG\_FSM\_STATE\_CHANGE | Descriptor FSM state change |
| **0x1** | CORE\_DEBUG\_DESCRIPTOR\_CONTENT | Descriptor content trace |
| **0x2** | CORE\_DEBUG\_FLAG\_ENGINE\_STATE | Flag engine state trace |
| **0x3** | CORE\_DEBUG\_PROGRAM\_ENGINE\_STATE | Program engine state trace |
| **0x4** | CORE\_DEBUG\_CREDIT\_OPERATION | Credit system operation |
| **0x5** | CORE\_DEBUG\_CONTROL\_REGISTER | Control register access |
| **0x6** | CORE\_DEBUG\_ENGINE\_HANDSHAKE | Engine handshake trace |
| **0x7** | CORE\_DEBUG\_QUEUE\_STATUS | Queue status change |
| **0x8** | CORE\_DEBUG\_COUNTER\_SNAPSHOT | Counter values snapshot |
| **0x9** | CORE\_DEBUG\_ADDRESS\_TRACE | Address progression trace |
| **0xA** | CORE\_DEBUG\_PAYLOAD\_TRACE | Payload content trace |
| **0xB-0xE** | Reserved | Future expansion |
| **0xF** | CORE\_DEBUG\_USER\_DEFINED | User-defined debug |

### Memory Architecture and Packet Routing

#### Two-Tier Memory Architecture

##### Local Error/Interrupt Memory

| Characteristic | Description |
| --- | --- |
| **Storage Types** | Error Packets (Type 0x0) and Timeout Packets (Type 0x3) |
| **Access Method** | Immediate CPU access without memory subsystem delays |
| **Capacity** | Large enough to prevent overflow during error bursts |
| **Priority** | Critical events requiring immediate attention |
| **Indexing** | Fast search and retrieval mechanisms |

##### Configurable External Memory

| Characteristic | Description |
| --- | --- |
| **Storage Types** | Performance, Completion, Threshold, Debug packets |
| **Access Method** | Base and limit registers define memory regions |
| **Capacity** | Bulk storage for non-critical events |
| **DMA Support** | Can be accessed via DMA for efficient transfer |
| **Time Stamping** | 32-bit timestamp appended when routing externally |

#### Routing Configuration

##### Base and Limit Registers

| Register Set | Purpose | Configuration |
| --- | --- | --- |
| **Completion Config** | Type 0x1 routing | base\_addr, limit\_addr, enable, priority |
| **Threshold Config** | Type 0x2 routing | base\_addr, limit\_addr, enable, priority |
| **Performance Config** | Type 0x4 routing | base\_addr, limit\_addr, enable, priority |
| **Debug Config** | Type 0xF routing | base\_addr, limit\_addr, enable, priority |

##### Routing Decision Logic

| Packet Type | Destination | Address Calculation |
| --- | --- | --- |
| **Error (0x0)** | Local Memory | local\_error\_write\_pointer |
| **Timeout (0x3)** | Local Memory | local\_error\_write\_pointer |
| **Completion (0x1)** | External Memory | completion\_config.base\_addr + offset |
| **Performance (0x4)** | External Memory | performance\_config.base\_addr + offset |
| **Debug (0xF)** | External Memory | debug\_config.base\_addr + offset |

#### Address Space Management

##### Memory Layout Example

| Address Range | Usage | Description |
| --- | --- | --- |
| **0x1000\_0000 - 0x1000\_FFFF** | Local Error Memory | Immediate access storage |
| **0x2000\_0000 - 0x2001\_FFFF** | Performance Packets | External bulk storage |
| **0x2010\_0000 - 0x2011\_FFFF** | Completion Packets | External bulk storage |
| **0x2020\_0000 - 0x202F\_FFFF** | Debug Packets | External bulk storage |

#### Transaction State and Bus Transaction Structure

##### Transaction State Enumeration

| State | Value | Description | Usage |
| --- | --- | --- | --- |
| **TRANS\_EMPTY** | 3’b000 | Unused entry | Available slot |
| **TRANS\_ADDR\_PHASE** | 3’b001 | Address phase active (AXI) / Packet sent (Network) / Setup phase (APB) | Initial phase |
| **TRANS\_DATA\_PHASE** | 3’b010 | Data phase active (AXI) / Waiting for ACK (Network) / Access phase (APB) | Data transfer |
| **TRANS\_RESP\_PHASE** | 3’b011 | Response phase active (AXI) / ACK received (Network) / Enable phase (APB) | Response handling |
| **TRANS\_COMPLETE** | 3’b100 | Transaction complete | Successful completion |
| **TRANS\_ERROR** | 3’b101 | Transaction has error | Error condition |
| **TRANS\_ORPHANED** | 3’b110 | Orphaned transaction | Missing components |
| **TRANS\_CREDIT\_STALL** | 3’b111 | Credit stall (Network only) | Network-specific stall |

##### Enhanced Transaction Structure

| Field | Width | Description | Protocol Usage |
| --- | --- | --- | --- |
| **valid** | 1 | Entry is valid | All protocol’s |
| **protocol** | 3 | Protocol type (AXI/Network/APB/ARB/CORE) | All protocols |
| **state** | 3 | Transaction state | All protocols |
| **id** | 32 | Transaction ID (AXI) / Sequence (Network) / PSEL encoding (APB) | All protocols |
| **addr** | 64 | Transaction address / Channel addr / PADDR | All protocols |
| **len** | 8 | Burst length (AXI) / Packet count (Network) / Always 0 (APB) | AXI, Network |
| **size** | 3 | Access size (AXI) / Reserved (Network) / Transfer size (APB) | AXI, APB |
| **burst** | 2 | Burst type (AXI) / Payload type (Network) / PPROT[1:0] (APB) | All protocols |

##### Phase Completion Flags

| Flag | Description | Protocol Usage |
| --- | --- | --- |
| **cmd\_received** | Address phase received / Packet sent / Setup phase | All protocols |
| **data\_started** | Data phase started / ACK expected / Access phase | All protocols |
| **data\_completed** | Data phase completed / ACK received / Enable phase | All protocols |
| **resp\_received** | Response received / Final ACK / PREADY asserted | All protocols |

##### Protocol-Specific Tracking Fields

| Field | Width | Description | Protocol |
| --- | --- | --- | --- |
| **channel** | 6 | Channel ID (AXI ID / Network channel / PSEL bit position) | All protocols |
| **eos\_seen** | 1 | EOS marker seen | Network only |
| **parity\_error** | 1 | Parity error detected | Network only |
| **credit\_at\_start** | 8 | Credits available at start | Network only |
| **retry\_count** | 3 | Number of retries | Network only |
| **desc\_addr\_match** | 1 | Descriptor address match detected | AXI only |
| **data\_addr\_match** | 1 | Data address match detected | AXI only |
| **apb\_phase** | 2 | Current APB phase | APB only |
| **pslverr\_seen** | 1 | PSLVERR detected | APB only |
| **pprot\_value** | 3 | PPROT value | APB only |
| **pstrb\_value** | 4 | PSTRB value for writes | APB only |
| **arb\_grant\_id** | 8 | Current grant ID | ARB only |
| **arb\_weight** | 8 | Current weight value | ARB only |
| **core\_fsm\_state** | 3 | Current CORE FSM state | CORE only |
| **core\_channel\_id** | 6 | CORE channel identifier | CORE only |

#### APB Transaction Phases

| Phase | Value | Description |
| --- | --- | --- |
| **APB\_PHASE\_IDLE** | 2’b00 | Bus idle |
| **APB\_PHASE\_SETUP** | 2’b01 | Setup phase (PSEL asserted) |
| **APB\_PHASE\_ACCESS** | 2’b10 | Access phase (PENABLE asserted) |
| **APB\_PHASE\_ENABLE** | 2’b11 | Enable phase (waiting for PREADY) |

#### APB Protection Types

| Protection | Value | Description |
| --- | --- | --- |
| **APB\_PROT\_NORMAL** | 3’b000 | Normal access |
| **APB\_PROT\_PRIVILEGED** | 3’b001 | Privileged access |
| **APB\_PROT\_SECURE** | 3’b010 | Secure access |
| **APB\_PROT\_INSTRUCTION** | 3’b100 | Instruction access |

#### Network Payload Types

| Payload | Value | Description |
| --- | --- | --- |
| **NETWORK\_PAYLOAD\_CONFIG** | 2’b00 | CONFIG\_PKT |
| **NETWORK\_PAYLOAD\_TS** | 2’b01 | TS\_PKT |
| **NETWORK\_PAYLOAD\_RDA** | 2’b10 | RDA\_PKT |
| **NETWORK\_PAYLOAD\_RAW** | 2’b11 | RAW\_PKT |

#### Network ACK Types

| ACK Type | Value | Description |
| --- | --- | --- |
| **NETWORK\_ACK\_STOP** | 2’b00 | MSAP\_STOP |
| **NETWORK\_ACK\_START** | 2’b01 | MSAP\_START |
| **NETWORK\_ACK\_CREDIT\_ON** | 2’b10 | MSAP\_CREDIT\_ON |
| **NETWORK\_ACK\_STOP\_AT\_EOS** | 2’b11 | MSAP\_STOP\_AT\_EOS |

#### ARB State Types

| State | Value | Description |
| --- | --- | --- |
| **ARB\_STATE\_IDLE** | 3’b000 | Idle state |
| **ARB\_STATE\_ARBITRATE** | 3’b001 | Performing arbitration |
| **ARB\_STATE\_GRANT** | 3’b010 | Grant issued, waiting for ACK |
| **ARB\_STATE\_BLOCKED** | 3’b011 | Arbitration blocked |
| **ARB\_STATE\_WEIGHT\_UPD** | 3’b100 | Weight update in progress |
| **ARB\_STATE\_ERROR** | 3’b101 | Error state |

#### CORE State Types

| State | Value | Description |
| --- | --- | --- |
| **CORE\_STATE\_IDLE** | 3’b000 | Idle state |
| **CORE\_STATE\_DESC\_FETCH** | 3’b001 | Fetching descriptor |
| **CORE\_STATE\_FLAG\_CHECK** | 3’b010 | Checking flag condition |
| **CORE\_STATE\_PROGRAM\_WRITE** | 3’b011 | Writing program |
| **CORE\_STATE\_DATA\_TRANSFER** | 3’b100 | Transferring data |
| **CORE\_STATE\_CREDIT\_WAIT** | 3’b101 | Waiting for credits |
| **CORE\_STATE\_ERROR** | 3’b110 | Error state |

### Configuration and Control

#### Monitor Configuration Registers

##### Global Configuration

| Field | Width | Description |
| --- | --- | --- |
| **monitor\_enable** | 1 | Global monitor enable |
| **error\_local\_enable** | 1 | Enable local error storage |
| **external\_route\_enable** | 1 | Enable external routing |
| **unit\_id** | 4 | Unit identifier |
| **agent\_id** | 8 | Agent identifier |
| **packet\_type\_enables** | 16 | Per-type enable bits |

##### Packet Type Enable Mapping

| Bit | Enable | Description |
| --- | --- | --- |
| **0** | PKT\_ENABLE\_ERROR | Enable error packets |
| **1** | PKT\_ENABLE\_COMPLETION | Enable completion packets |
| **2** | PKT\_ENABLE\_THRESHOLD | Enable threshold packets |
| **3** | PKT\_ENABLE\_TIMEOUT | Enable timeout packets |
| **4** | PKT\_ENABLE\_PERF | Enable performance packets |
| **5** | PKT\_ENABLE\_CREDIT | Enable credit packets (Network) |
| **6** | PKT\_ENABLE\_CHANNEL | Enable channel packets (Network) |
| **7** | PKT\_ENABLE\_STREAM | Enable stream packets (Network) |
| **8** | PKT\_ENABLE\_ADDR\_MATCH | Enable address match (AXI) |
| **9** | PKT\_ENABLE\_APB | Enable APB packets |
| **15** | PKT\_ENABLE\_DEBUG | Enable debug packets |

#### Protocol-Specific Configuration

##### AXI Monitor Configuration

| Field | Width | Description |
| --- | --- | --- |
| **active\_trans\_threshold** | 16 | Active transaction threshold |
| **latency\_threshold** | 32 | Latency threshold (cycles) |
| **addr\_timeout\_cnt** | 4 | Address timeout count |
| **data\_timeout\_cnt** | 4 | Data timeout count |
| **resp\_timeout\_cnt** | 4 | Response timeout count |
| **burst\_boundary\_check** | 1 | Enable burst boundary checking |
| **address\_match\_enable** | 1 | Enable address matching |
| **desc\_addr\_match\_base** | 64 | Descriptor address match base |
| **desc\_addr\_match\_mask** | 64 | Descriptor address match mask |
| **data\_addr\_match\_base** | 64 | Data address match base |
| **data\_addr\_match\_mask** | 64 | Data address match mask |

##### Network Monitor Configuration

| Field | Width | Description |
| --- | --- | --- |
| **credit\_low\_threshold** | 8 | Credit low threshold |
| **packet\_rate\_threshold** | 16 | Packet rate threshold |
| **max\_route\_hops** | 8 | Maximum routing hops |
| **enable\_credit\_tracking** | 1 | Enable credit tracking |
| **enable\_deadlock\_detect** | 1 | Enable deadlock detection |
| **deadlock\_timeout** | 4 | Deadlock detection timeout |

##### ARB Monitor Configuration

| Field | Width | Description |
| --- | --- | --- |
| **grant\_timeout\_cnt** | 16 | Grant ACK timeout count |
| **fairness\_window** | 32 | Fairness analysis window |
| **weight\_update\_enable** | 1 | Enable weight tracking |
| **starvation\_threshold** | 16 | Starvation detection threshold |
| **efficiency\_threshold** | 8 | Grant efficiency threshold |

##### CORE Monitor Configuration

| Field | Width | Description |
| --- | --- | --- |
| **descriptor\_timeout\_cnt** | 16 | Descriptor fetch timeout count |
| **flag\_retry\_limit** | 8 | Maximum flag retry count |
| **credit\_low\_threshold** | 8 | Credit low threshold |
| **processing\_timeout\_cnt** | 32 | Processing timeout count |
| **enable\_descriptor\_trace** | 1 | Enable descriptor content tracing |
| **enable\_fsm\_trace** | 1 | Enable FSM state tracing |

### Validation Requirements

#### Functional Validation

| Validation Area | Requirements |
| --- | --- |
| **Packet Format** | Verify 64-bit packet structure and field encoding |
| **Event Organization** | Verify hierarchical event code organization |
| **Protocol Isolation** | Verify independent protocol event spaces |
| **Routing Logic** | Verify packet routing based on type and configuration |
| **Memory Management** | Verify local and external memory operations |
| **Configuration** | Verify register configuration and enable controls |

#### Performance Validation

| Validation Area | Requirements |
| --- | --- |
| **Throughput** | Verify monitor bus can handle peak event rates |
| **Latency** | Verify low-latency path for critical events |
| **Memory Efficiency** | Verify efficient memory usage patterns |
| **Power Consumption** | Verify power-efficient operation |

#### Error Handling Validation

| Validation Area | Requirements |
| --- | --- |
| **Error Injection** | Verify error detection and reporting |
| **Overflow Handling** | Verify behavior when memories fill |
| **Configuration Errors** | Verify invalid configuration detection |
| **Recovery Mechanisms** | Verify error recovery procedures |

### Usage Examples

#### Creating Monitor Packets

| Packet Type | Example Usage |
| --- | --- |
| **AXI Error** | Protocol=AXI, Type=Error, Code=AXI\_ERR\_RESP\_SLVERR |
| **Network Credit** | Protocol=Network, Type=Credit, Code=NETWORK\_CREDIT\_EXHAUSTED |
| **APB Performance** | Protocol=APB, Type=Performance, Code=APB\_PERF\_TOTAL\_LATENCY |
| **ARB Threshold** | Protocol=ARB, Type=Threshold, Code=ARB\_THRESH\_FAIRNESS\_DEV |
| **CORE Completion** | Protocol=CORE, Type=Completion, Code=CORE\_COMPL\_DESCRIPTOR\_LOADED |

#### Packet Decoding

| Decoding Step | Method |
| --- | --- |
| **Extract Type** | packet[63:60] |
| **Extract Protocol** | packet[59:57] |
| **Extract Event Code** | packet[56:53] |
| **Extract Channel ID** | packet[52:47] |
| **Extract Event Data** | packet[34:0] |

#### Monitor Bus Packet Helper Functions

##### Packet Field Extraction

| Function | Return Type | Description |
| --- | --- | --- |
| **get\_packet\_type(pkt)** | logic [3:0] | Extract packet type [63:60] |
| **get\_protocol\_type(pkt)** | protocol\_type\_t | Extract protocol [59:57] |
| **get\_event\_code(pkt)** | logic [3:0] | Extract event code [56:53] |
| **get\_channel\_id(pkt)** | logic [5:0] | Extract channel ID [52:47] |
| **get\_unit\_id(pkt)** | logic [3:0] | Extract unit ID [46:43] |
| **get\_agent\_id(pkt)** | logic [7:0] | Extract agent ID [42:35] |
| **get\_event\_data(pkt)** | logic [34:0] | Extract event data [34:0] |

##### Packet Creation Function

| Function | Parameters | Description |
| --- | --- | --- |
| **create\_monitor\_packet()** | packet\_type, protocol, event\_code, channel\_id, unit\_id, agent\_id, event\_data | Create complete 64-bit packet |

##### Event Code Creation Functions

| Function | Parameter | Description |
| --- | --- | --- |
| **create\_axi\_error\_event()** | axi\_error\_code\_t | Create AXI error event code |
| **create\_axi\_timeout\_event()** | axi\_timeout\_code\_t | Create AXI timeout event code |
| **create\_axi\_completion\_event()** | axi\_completion\_code\_t | Create AXI completion event code |
| **create\_axi\_threshold\_event()** | axi\_threshold\_code\_t | Create AXI threshold event code |
| **create\_axi\_performance\_event()** | axi\_performance\_code\_t | Create AXI performance event code |
| **create\_axi\_addr\_match\_event()** | axi\_addr\_match\_code\_t | Create AXI address match event code |
| **create\_axi\_debug\_event()** | axi\_debug\_code\_t | Create AXI debug event code |
| **create\_apb\_error\_event()** | apb\_error\_code\_t | Create APB error event code |
| **create\_apb\_timeout\_event()** | apb\_timeout\_code\_t | Create APB timeout event code |
| **create\_apb\_completion\_event()** | apb\_completion\_code\_t | Create APB completion event code |
| **create\_network\_error\_event()** | network\_error\_code\_t | Create Network error event code |
| **create\_network\_timeout\_event()** | network\_timeout\_code\_t | Create Network timeout event code |
| **create\_network\_completion\_event()** | network\_completion\_code\_t | Create Network completion event code |
| **create\_network\_credit\_event()** | network\_credit\_code\_t | Create Network credit event code |
| **create\_network\_channel\_event()** | network\_channel\_code\_t | Create Network channel event code |
| **create\_network\_stream\_event()** | network\_stream\_code\_t | Create Network stream event code |
| **create\_arb\_error\_event()** | arb\_error\_code\_t | Create ARB error event code |
| **create\_arb\_timeout\_event()** | arb\_timeout\_code\_t | Create ARB timeout event code |
| **create\_arb\_completion\_event()** | arb\_completion\_code\_t | Create ARB completion event code |
| **create\_arb\_threshold\_event()** | arb\_threshold\_code\_t | Create ARB threshold event code |
| **create\_arb\_performance\_event()** | arb\_performance\_code\_t | Create ARB performance event code |
| **create\_arb\_debug\_event()** | arb\_debug\_code\_t | Create ARB debug event code |
| **create\_core\_error\_event()** | core\_error\_code\_t | Create CORE error event code |
| **create\_core\_timeout\_event()** | core\_timeout\_code\_t | Create CORE timeout event code |
| **create\_core\_completion\_event()** | core\_completion\_code\_t | Create CORE completion event code |
| **create\_core\_threshold\_event()** | core\_threshold\_code\_t | Create CORE threshold event code |
| **create\_core\_performance\_event()** | core\_performance\_code\_t | Create CORE performance event code |
| **create\_core\_debug\_event()** | core\_debug\_code\_t | Create CORE debug event code |

##### Validation Functions

| Function | Parameters | Description |
| --- | --- | --- |
| **is\_valid\_event\_for\_packet\_type()** | packet\_type, protocol, event\_code | Validate event code for packet type and protocol |

##### String Functions for Debugging

| Function | Parameter | Description |
| --- | --- | --- |
| **get\_axi\_error\_name()** | axi\_error\_code\_t | Get human-readable AXI error name |
| **get\_arb\_error\_name()** | arb\_error\_code\_t | Get human-readable ARB error name |
| **get\_core\_error\_name()** | core\_error\_code\_t | Get human-readable CORE error name |
| **get\_packet\_type\_name()** | logic [3:0] | Get packet type name string |
| **get\_protocol\_name()** | protocol\_type\_t | Get protocol name string |
| **get\_event\_name()** | packet\_type, protocol, event\_code | Get comprehensive event name |

### Debug and Monitoring Signals

#### Essential Debug Signals

| Signal | Width | Purpose |
| --- | --- | --- |
| **debug\_packet\_counts** | 32 x 16 | Packet count per type |
| **debug\_protocol\_counts** | 32 x 5 | Packet count per protocol |
| **debug\_error\_counts** | 32 | Total error packet count |
| **debug\_local\_memory\_level** | 16 | Local memory usage level |
| **debug\_external\_memory\_level** | 16 | External memory usage level |

#### Performance Counters

| Counter | Width | Purpose |
| --- | --- | --- |
| **total\_packets\_processed** | 32 | Total packets processed |
| **packets\_dropped** | 32 | Packets dropped due to overflow |
| **routing\_errors** | 32 | Routing configuration errors |
| **memory\_full\_events** | 32 | Memory full occurrences |

### Protocol Coverage Summary

#### Complete Protocol Event Matrix

| Protocol | Error | Timeout | Completion | Threshold | Performance | Debug | Protocol-Specific |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **AXI** | [PASS] 16 | [PASS] 16 | [PASS] 16 | [PASS] 16 | [PASS] 16 | [PASS] 16 | AddrMatch [PASS] 16 |
| **Network** | [PASS] 16 | [PASS] 16 | [PASS] 16 | [FAIL] 0 | [FAIL] 0 | [FAIL] 0 | Credit/Channel/Stream [PASS] 48 |
| **APB** | [PASS] 16 | [PASS] 16 | [PASS] 16 | [PASS] 16 | [PASS] 16 | [PASS] 16 | None |
| **ARB** | [PASS] 16 | [PASS] 16 | [PASS] 16 | [PASS] 16 | [PASS] 16 | [PASS] 16 | None |
| **CORE** | [PASS] 16 | [PASS] 16 | [PASS] 16 | [PASS] 16 | [PASS] 16 | [PASS] 16 | None |

**Total Event Codes**: 544 defined across all protocols and packet types.

## Chapter 4: Programming Models

## Programming

TBD ## Chapter 5: Registers