Contents

Rapids Index

Generated: 2026-01-15

RTL Design Sherpa · Learning Hardware Design Through Practice GitHub · Documentation Index · MIT License

---

Memory IO Processor (RAPIDS) Specification

Chapter 1: Overview

- Overview
- Architecture Requirements
- Clocking and Reset
- Acronyms and Abbreviations

Chapter 2: Blocks

Scheduler Group

Sink Data Path

Source Data Path

MonBus AXIL Group

FSM Summary

Chapter 3: Interfaces and Busses

Top-Level Ports

Bus Definitions

- AXIL4
- AXI4
- AXIS4 (AXI4-Stream)
- MONBUS

Chapter 4: Programming Models

- RAPIDS Registers

Chapter 5: Registers

- RAPIDS Registers

RTL Design Sherpa · Learning Hardware Design Through Practice GitHub · Documentation Index · MIT License

---

Chapter 1.1: RAPIDS Overview

1.1.1 Executive Summary

RAPIDS (Rapid AXI Programmable In-band Descriptor System) is a high-performance DMA engine designed to bridge AXI4 memory-mapped interfaces with the Delta Network compute fabric. It operates under control of the HIVE-C master controller (VexRiscv RISC-V core) which provides inband descriptor injection through the Delta Network. RAPIDS intelligently routes data between external DDR memory and compute tiles while enforcing strict packet type filtering to maintain protocol integrity.

Key Features: - **Scatter-gather DMA** with linked descriptor chains - **In-band descriptor reception** via CDA (Compute Direct Access) packets - no memory polling - **Multi-priority descriptor scheduling** for efficient task execution - **16-channel S2MM architecture** (one channel per compute tile in 4×4 mesh) - **Packet type filtering** on all AXIS interfaces - **Performance monitoring** and statistics collection - **Completion interrupt** generation

1.1.2 Position in System Architecture

RAPIDS serves as the critical bridge between the memory subsystem and the Delta Network compute fabric:

```
┌─────────────────────────────────────────────────────────────┐
│                           HIVE-C                            │
│                  (VexRiscv RISC-V Master)                   │
│                                                             │
│  Generates CDA descriptors, injects via AXIS               │
└──────────────────────┬──────────────────────────────────────┘
                       │ AXIS TX (CDA packets)
                       ▼
              ┌────────────────────┐
              │   Delta Network    │
              │   Routes packets   │
              │   based on TUSER   │
              └────────┬───────────┘
                       │ CDA -> RAPIDS (tile 16)
                       │ PKT_DATA -> Compute Tiles (0-15)
                       ▼
┌──────────────────────────────────────────────────────────────┐
│                      RAPIDS DMA Engine                       │
│                    (Virtual Tile 16)                         │
│  ┌─────────────────────────────────────────────────────┐    │
│  │  AXIS CDA Input (Slave)                             │    │
│  │  - Accepts: CDA packets only                        │    │
│  │  - Rejects: PKT_DATA, PKT_CONFIG, PKT_STATUS        │    │
│  │  - Routes to: Descriptor FIFO                       │    │
│  └──────────────────┬──────────────────────────────────┘    │
│                     ▼                                        │
│  ┌─────────────────────────────────────────────────────┐    │
│  │  Descriptor Processing Engine                       │    │
│  │  - Parses 256-bit descriptors                       │    │
│  │  - Schedules by priority                            │    │
│  │  - Generates AXI4 transactions                      │    │
│  └──────────────────┬──────────────────────────────────┘    │
│                     ▼                                        │
│  ┌─────────────────────────────────────────────────────┐    │
│  │  AXI4 Memory Interface (Master)                     │    │
│  │  - Reads from DDR (activations, weights)            │    │
│  │  - Writes to DDR (results)                          │    │
│  └──────────────────┬──────────────────────────────────┘    │
│                     ▼                                        │
│  ┌─────────────────────────────────────────────────────┐    │
│  │  AXIS Data Output (Master)                          │    │
│  │  - Sends: PKT_DATA only                             │    │
│  │  - Never sends: CDA, PKT_CONFIG, PKT_STATUS         │    │
│  │  - Routes to: Delta Network -> Compute Tiles         │    │
│  └───────────────────┬──────────────────────────────────┘    │
└───────────────────┼──────────────────────────────────────────┘
                     │
                     ▼
            ┌─────────────────┐
            │ Delta Network   │
            │ (PKT_DATA only) │
            └────────┬────────┘
                      ▼
            Compute Tiles (0-15)

Memory: DDR2/DDR3 ◄──AXI4──┤ RAPIDS ├──AXIS──► Delta Network
```

1.1.3 Virtual Tile Concept

Within the Delta Network topology, RAPIDS is addressed as **Virtual Tile 16**:

```
Delta Network 4×4 Physical Mesh (Tiles 0-15)
┌────┬────┬────┬────┐
│ T0 │ T1 │ T2 │ T3 │  Row 0
├────┼────┼────┼────┤
│ T4 │ T5 │ T6 │ T7 │  Row 1
├────┼────┼────┼────┤
│ T8 │ T9 │T10 │T11 │  Row 2
├────┼────┼────┼────┤
│T12 │T13 │T14 │T15 │  Row 3
└────┴────┴────┴────┘
         │
         └─── Virtual Tile 16 (RAPIDS) connected via Router 12 south port

Virtual Tile 17 (HIVE-C) connected via Router 3 north port
```

Routing Implications: - **To RAPIDS:** TDEST = 16 routes packets from compute tiles to RAPIDS - **From RAPIDS:** TDEST = 0-15 routes data to specific compute tiles - **CDA packets:** Automatically routed to RAPIDS (tile 16) by Delta Network

1.1.4 Packet Type Handling

RAPIDS enforces strict packet type filtering based on TUSER encoding:

Interface	Direction	Accepts	Sends	Rejects
AXIS CDA Input	Slave	CDA packets	-	PKT_DATA, PKT_CONFIG, PKT_STATUS
AXIS Data Out	Master	-	PKT_DATA	CDA, PKT_CONFIG, PKT_STATUS
AXIS Data In	Slave	PKT_DATA	-	CDA, PKT_CONFIG, PKT_STATUS

Packet Type Encoding (TUSER[1:0])

TUSER[1:0]	Packet Type	Generated By	Accepted By
2'b00	PKT_DATA	RAPIDS, Compute Tiles	RAPIDS, Compute Tiles
2'b01	CDA (Compute Direct Access)	HIVE-C only	RAPIDS only
2'b10	PKT_CONFIG	HIVE-C, SERV monitors	Tile Routers
2'b11	PKT_STATUS	SERV monitors	HIVE-C

Rejection Behavior: - **Invalid packet type on input:** Assert TREADY=0, set error flag, generate interrupt - **Transmission enforcement:** Hardware prevents transmission of invalid packet types

1.1.5 Data Flow: Memory to Compute (MM2S)

```
1. HIVE-C generates CDA descriptor in firmware
2. HIVE-C writes descriptor to AXIS TX FIFO with TUSER=CDA (2'b01)
3. Delta Network routes CDA packet to RAPIDS (tile 16)
4. RAPIDS validates TUSER == CDA, writes to descriptor FIFO
5. Scheduler prioritizes and fetches descriptor
6. MM2S engine:
   a. Reads data from DDR via AXI4
   b. Buffers in internal FIFO
   c. Formats AXIS packets with TUSER=PKT_DATA (2'b00)
   d. Sets TDEST to target compute tile (from descriptor)
   e. Transmits to Delta Network
```

```
7. Delta Network routes PKT_DATA to destination tile
8. Completion interrupt generated (if enabled)
```

1.1.6 Data Flow: Compute to Memory (S2MM)

RAPIDS implements **16 independent S2MM channels**, one per compute tile:

```
1. HIVE-C pre-programs S2MM descriptors (one per active tile)
2. Compute tiles generate results, send as PKT_DATA with:
   - TUSER = PKT_DATA (2'b00)
   - TID = source tile ID (0-15)  <- CRITICAL for channel routing
   - TDEST = 16 (RAPIDS virtual tile)
3. Delta Network routes to RAPIDS
4. RAPIDS S2MM channel router:
   channel_id = TID[3:0]  // Direct mapping: TID -> Channel
5. Selected S2MM channel:
   a. Validates TUSER == PKT_DATA
   b. Buffers in per-channel FIFO
   c. Fetches descriptor from per-channel queue
   d. Writes to DDR via AXI4 (address from descriptor)
6. Completion interrupt generated (if enabled)
```

Multi-Channel Benefits: - **Concurrent writes:** Multiple tiles write simultaneously - **Independent addressing:** Each tile's data goes to different DDR region - **Per-tile flow control:** Backpressure on one tile doesn't affect others - **Simplified routing:** Direct TID -> Channel mapping (no lookup table)

1.1.7 CDA Packet Format

CDA (Compute Direct Access) packets carry descriptor information from HIVE-C to RAPIDS:

Format: 256 bits (2 AXIS beats at 128-bit width)

```
Beat 0:
TUSER[1:0] = 2'b01        // CDA packet type
TDATA[127:0] = Desc[127:0] // Descriptor bits [127:0]
TLAST = 0                 // More beats follow
TID[3:0] = Priority       // 0=highest, 15=lowest
TDEST[3:0] = 4'h0         // Always RAPIDS (routed to tile 16)

Beat 1:
TUSER[1:0] = 2'b01        // CDA packet type
TDATA[127:0] = Desc[255:128] // Descriptor bits [255:128]
TLAST = 1                 // Final beat
```

```
TID[3:0] = Priority       // Same as beat 0
TDEST[3:0] = 4'h0         // Always RAPIDS
```

Descriptor Fields (256 bits total): - Bits [255:192]: Source Address (64-bit DDR address) - Bits [191:128]: Destination Address (64-bit DDR address) - Bits [127:96]: Transfer Length (32-bit bytes) - Bits [95:64]: Control/Configuration (stride, tile ID, etc.) - Bits [63:32]: Control Flags (type, priority, interrupts) - Bits [31:0]: Next Descriptor Pointer (scatter-gather chaining)

See Chapter 2 (Descriptor Engine) for complete descriptor field definitions.

1.1.8 System Integration Summary

RAPIDS Never Generates: - CDA packets (only HIVE-C generates these) - PKT_CONFIG (only HIVE-C or SERV monitors generate) - PKT_STATUS (only SERV monitors generate)

RAPIDS Always Validates: - CDA input packets must have TUSER = 2’b01 - Data input packets must have TUSER = 2’b00 - Any other TUSER value triggers error condition and packet rejection

Control Plane: - HIVE-C uses AXI4-Lite to access RAPIDS control/status registers - CDA packets provide descriptor injection (inband, low latency) - Interrupts notify HIVE-C of completion/errors

Data Plane: - AXI4 master interface to DDR memory (read/write) - AXIS master interface to Delta Network (PKT_DATA out) - AXIS slave interface from Delta Network (PKT_DATA in) - Multi-channel architecture supports concurrent tile operations

---

Next: Architectural Requirements

Back to: Index

Chapter 1.2: Architectural Requirements

1.2.1 System Requirements

SR-1: Delta Network Integration

RAPIDS **shall** integrate with the Delta Network as Virtual Tile 16, accepting CDA packets from HIVE-C and routing PKT_DATA packets to/from compute tiles.

SR-2: HIVE-C Control Interface

RAPIDS **shall** provide an AXI4-Lite slave interface for control/status register access by HIVE-C.

SR-3: Memory Interface

RAPIDS **shall** provide an AXI4 master interface supporting burst transactions to DDR2/DDR3/DDR4 memory controllers.

SR-4: Packet Type Enforcement

RAPIDS **shall** enforce strict packet type filtering on all AXIS interfaces, rejecting invalid packet types with error indication.

1.2.2 Descriptor Requirements

DR-1: Inband Descriptor Reception

RAPIDS **shall** accept descriptor injection via CDA packets from HIVE-C through the Delta Network (no memory polling required).

DR-2: Descriptor Queue Depth

RAPIDS **shall** support a minimum of 8 pending descriptors in the descriptor FIFO.

DR-3: Priority Scheduling

RAPIDS **shall** support priority-based descriptor scheduling with 16 priority levels (0=highest, 15=lowest).

DR-4: Scatter-Gather Support

RAPIDS **shall** support linked descriptor chains via next descriptor pointer field.

DR-5: Descriptor Format

RAPIDS **shall** accept 256-bit descriptors conforming to the format specified in Chapter 2.

1.2.3 Data Transfer Requirements

DT-1: Memory-to-Stream (MM2S) Engine

RAPIDS **shall** support reading data from DDR memory and transmitting as PKT_DATA packets to the Delta Network.

DT-2: Stream-to-Memory (S2MM) Multi-Channel Architecture

RAPIDS **shall** implement 16 independent S2MM channels, one per compute tile in a 4×4 mesh, with direct TID-to-channel mapping.

DT-3: Concurrent Channel Operation

S2MM channels **shall** operate independently, allowing multiple compute tiles to write to memory simultaneously.

DT-4: Burst Support

RAPIDS **shall** support AXI4 burst transactions with lengths up to 256 beats.

DT-5: Data Buffering

RAPIDS **shall** provide internal FIFOs to decouple memory timing from Delta Network timing: - MM2S data FIFO: Minimum 512 entries × 128 bits - S2MM per-channel FIFO: Minimum 32 entries × 128 bits

1.2.4 Performance Requirements

PR-1: Throughput

RAPIDS **shall** sustain a minimum of 1.3 GB/s for large (>1 KB) transfers under ideal conditions.

PR-2: Latency - Descriptor Injection

RAPIDS **shall** process CDA packet descriptors within 20 cycles of reception (from HIVE-C via Delta Network).

PR-3: Latency - First Data

RAPIDS **shall** deliver first data beat within 65 cycles of descriptor execution start (MM2S mode, DDR3 target).

PR-4: Burst Efficiency

RAPIDS **shall** achieve >90% burst efficiency for transfers >1 KB.

1.2.5 Interface Requirements

IF-1: AXIS CDA Input

RAPIDS **shall** provide an AXIS slave interface accepting only CDA packets (TUSER = 2'b01) from the Delta Network.

Signals: - TDATA: 128 bits - TUSER: 2 bits (must be 2'b01) - TID: 4 bits (descriptor priority) - TDEST: 4 bits (always 0 for RAPIDS routing) - TLAST: 1 bit - TVALID/TREADY: Handshake

IF-2: AXIS Data Output (MM2S)

RAPIDS **shall** provide an AXIS master interface transmitting only PKT_DATA packets (TUSER = 2'b00) to the Delta Network.

Signals: - TDATA: 128 bits - TUSER: 2 bits (always 2'b00) - TID: 4 bits (descriptor priority) - TDEST: 4 bits (target tile ID 0-15) - TLAST: 1 bit - TKEEP: 16 bits - TVALID/TREADY: Handshake

IF-3: AXIS Data Input (S2MM)

RAPIDS **shall** provide an AXIS slave interface accepting only PKT_DATA packets (TUSER = 2'b00) from Delta Network compute tiles.

Signals: - TDATA: 128 bits - TUSER: 2 bits (must be 2'b00) - TID: 4 bits (source tile ID 0-15) <- **Used for channel routing** - TDEST: 4 bits (must be 16 for RAPIDS) - TLAST: 1 bit - TKEEP: 16 bits - TVALID/TREADY: Handshake

IF-4: AXI4 Memory Interface

RAPIDS **shall** provide an AXI4 master interface with: - Address width: 32 bits (supports up to 4 GB) - Data width: 128 bits (configurable 64/128/256) - Burst types: INCR, FIXED, WRAP - Max burst length: 256 beats - Outstanding transactions: 16 (configurable)

IF-5: AXI4-Lite Control Interface

RAPIDS **shall** provide an AXI4-Lite slave interface for control/status registers with: - Address width: 12 bits (4 KB register space) - Data width: 32 bits - Single outstanding transaction

1.2.6 Error Handling Requirements

EH-1: Packet Type Validation

RAPIDS **shall** reject packets with invalid TUSER values by: - Asserting TREADY = 0 (not accepting packet) - Setting appropriate error flag in status register - Generating interrupt (if enabled)

EH-2: AXI4 Error Response

RAPIDS **shall** handle AXI4 SLVERR and DECERR responses by: - Setting error flag in status register - Optionally skipping to next descriptor - Generating interrupt (if enabled)

EH-3: Descriptor FIFO Overflow

RAPIDS **shall** prevent descriptor loss on FIFO full by: - Asserting backpressure (TREADY = 0) on CDA input - Setting FIFO full flag in status register

EH-4: Malformed Descriptor

RAPIDS **shall** handle malformed descriptors by: - Setting error flag - Skipping to next descriptor (if scatter-gather enabled) - Generating interrupt (if enabled)

1.2.7 Monitoring and Debug Requirements

MD-1: Performance Counters

RAPIDS **shall** provide read-only counters for: - Bytes read from memory - Bytes written to memory - Packets transmitted (AXIS out) - Packets received (AXIS in) - AXI4 read/write cycle counts

MD-2: Status Visibility

RAPIDS **shall** provide status indicators for: - Descriptor FIFO occupancy - Data FIFO status (full/empty flags) - Current priority being processed - Engine busy indicators (MM2S, S2MM per-channel) - Scatter-gather chain active

MD-3: Interrupt Generation

RAPIDS **shall** generate interrupts for: - Descriptor completion (configurable per-descriptor) - FIFO full conditions - AXI4 errors - Invalid packet type reception - Descriptor parse errors

1.2.8 Configuration Requirements

CF-1: Compile-Time Parameters

RAPIDS **shall** support compile-time configuration of: - AXI4 data width (64/128/256 bits) - AXI4 address width (32/64 bits) - Descriptor FIFO depth (8-256 entries) - Data FIFO depths (MM2S and S2MM) - Number of S2MM channels (4/8/16 for 2×2/3×3/4×4 meshes) - Outstanding AXI4 transaction limit

CF-2: Runtime Configuration

RAPIDS **shall** support runtime configuration via registers: - Engine enable/disable (MM2S, S2MM) - Interrupt mask - Error handling policy - Statistics counter enable

1.2.9 Reset and Initialization Requirements

RI-1: Synchronous Reset

RAPIDS **shall** support synchronous active-low reset (aresetn) for all internal state.

RI-2: Graceful Reset

RAPIDS **shall** complete in-flight AXI4 transactions before asserting internal reset (when possible).

RI-3: FIFO Flush

RAPIDS **shall** support selective FIFO flushing via control register: - Descriptor FIFO flush - Data FIFO flush (MM2S and S2MM)

RI-4: Initialization State

After reset deassertion, RAPIDS **shall**: - Clear all FIFOs - Disable all engines (MM2S, S2MM channels) - Clear all error flags - Reset performance counters

---

Next: Clocks and Reset

Previous: Overview

Back to: Index

Chapter 1.3: Clocks and Reset

1.3.1 Clock Domains

RAPIDS operates within a multi-clock domain system due to its integration with Delta Network and DDR memory:

Primary Clock Domains

1. APB Control Clock (`pclk`) - **Frequency:** Typically 50-100 MHz - **Purpose:** Control/status register access by HIVE-C - **Interface:** AXI4-Lite slave - **Requirements:** Synchronous to HIVE-C system clock

2. AXI4 Memory Clock (`aclk`) - **Frequency:** Typically 200-400 MHz (DDR2/DDR3/DDR4 dependent) - **Purpose:** High-performance memory access - **Interfaces:** - AXI4 master (read/write to DDR) - Internal RAPIDS core logic - **Requirements:** Synchronous to memory controller clock

3. Delta Network Clock (`network_clk`) - **Frequency:** Typically 250-500 MHz - **Purpose:** Packet routing and network communication - **Interfaces:** - AXIS CDA input (slave) - AXIS Data output (master) - AXIS Data input (slave) - **Requirements:** Synchronous to Delta Network fabric

Clock Domain Relationships

```
┌─────────────────────────────────────────────────────────────────┐
│                           HIVE-C                                │
│                   (Control Plane Clock)                         │
│                         50-100 MHz                              │
└──────────────────────────┬──────────────────────────────────────┘
                           │ pclk
                           ▼
              ┌────────────────────────────┐
              │   APB Control Interface    │
              │   (AXI4-Lite Slave)       │
              └────────────┬───────────────┘
                           │ CDC if async
                           ▼
┌─────────────────────────────────────────────────────────────────┐
│                        RAPIDS Core                              │
│                    (aclk - Memory Clock)                        │
│                       200-400 MHz                               │
│  ┌──────────────┐  ┌──────────────┐  ┌──────────────┐         │
│  │ Descriptor   │  │    MM2S      │  │    S2MM      │         │
│  │   Engine     │  │   Engine     │  │  Channels    │         │
│  └──────────────┘  └──────────────┘  └──────────────┘         │
└─────────────────────────────────────────────────────────────────┘
             │                │                  │
             │                │                  │
             ▼                ▼                  ▼
   ┌─────────────────┐  ┌──────────────┐  ┌──────────────┐
   │ AXI4 Memory     │  │  CDC FIFO    │  │  CDC FIFO    │
   │   Interface     │  │  MM2S->AXIS   │  │  AXIS->S2MM   │
   │ (aclk domain)   │  └──────────────┘  └──────────────┘
   └─────────────────┘         │ CDC              │ CDC
             │                 ▼                  ▼
             ▼          ┌────────────────────────────────────┐
   ┌─────────────┐     │      Delta Network                 │
   │     DDR     │     │   (network_clk - 250-500 MHz)      │
   │   Memory    │     │                                     │
   └─────────────┘     │  AXIS TX (PKT_DATA)                │
                        │  AXIS RX (PKT_DATA)                │
                        │  AXIS CDA (descriptors)            │
                        └────────────────────────────────────┘
```

1.3.2 Clock Domain Crossing (CDC)

CDC Requirements

RAPIDS requires CDC for these interface crossings:

1. **APB <-> Core (Optional)** - **When:** If HIVE-C APB clock differs from RAPIDS core clock - **Method:** APB slave CDC (handshake synchronization) - **Latency Impact:** 4-6 clock cycles for register access - **Parameter:** `CDC_ENABLE` (0=synchronous, 1=async)

2. **Core <-> Delta Network (Typical)** - **Crossing:** `aclk` (memory) <-> `network_clk` (Delta Network) - **Method:** Asynchronous FIFOs with gray-coded pointers - **Locations:** - MM2S data FIFO (memory -> network) - S2MM data FIFOs (network -> memory, per-channel) - CDA descriptor FIFO (network -> core) - **Depth:** Minimum 32 entries to absorb clock ratio variations

3. **Core <-> Memory (Synchronous)** - **Assumption:** RAPIDS core runs at memory clock (`aclk`) - **No CDC Required:** Direct AXI4 connection - **Benefit:** Lowest latency for memory access

CDC FIFO Specifications

MM2S Data FIFO:

```
Write Side: aclk (RAPIDS core)
Read Side: network_clk (Delta Network)
Depth: 512 entries × DATA_WIDTH
Empty/Full: Gray-coded flags with multi-stage synchronization
```

S2MM Data FIFO (per channel):

```
Write Side: network_clk (Delta Network)
Read Side: aclk (RAPIDS core)
Depth: 32 entries × DATA_WIDTH
Empty/Full: Gray-coded flags with multi-stage synchronization
```

CDA Descriptor FIFO:

```
Write Side: network_clk (Delta Network CDA packets)
Read Side: aclk (RAPIDS descriptor processing)
Depth: 8 descriptors × 256 bits
Empty/Full: Gray-coded flags with multi-stage synchronization
```

1.3.3 Reset Strategy

Reset Signals

RAPIDS uses **synchronous active-low reset** for all clock domains:

1. APB Reset (`presetn`) - **Domain:** APB control interface - **Type:** Active-low synchronous - **Source:** HIVE-C system reset - **Scope:** APB slave interface only

2. Core Reset (`aresetn`) - **Domain:** RAPIDS core logic + AXI4 memory interface - **Type:** Active-low synchronous (AXI4 standard) - **Source:** System reset - **Scope:** All RAPIDS internal state, descriptor engine, scheduler, data paths

3. Network Reset (`network_rst_n`) - **Domain:** Delta Network interfaces (AXIS) - **Type:** Active-low synchronous - **Source:** Delta Network fabric reset - **Scope:** AXIS interfaces, CDC FIFOs (network side)

Reset Sequencing

Recommended power-on reset sequence:

```
1. Assert all resets (presetn=0, aresetn=0, network_rst_n=0)
   - Hold for minimum 10 cycles in slowest domain

2. Deassert network_rst_n (Delta Network reset)
```

```
   - Allow Delta Network fabric to initialize
   - Wait 100 cycles (network_clk)

3. Deassert aresetn (RAPIDS core reset)
   - RAPIDS core initializes
   - CDC FIFOs reset
   - Wait 50 cycles (aclk)

4. Deassert presetn (APB control reset)
   - APB interface ready
   - HIVE-C can access control registers

5. Software initialization
   - HIVE-C configures RAPIDS via APB
   - Enable engines, set initial credits
```

Reset Requirements

RI-1: Synchronous Reset - All reset signals synchronized to respective clock domains - No asynchronous reset (metastability risk)

RI-2: Graceful Reset - RAPIDS completes in-flight AXI4 transactions before internal reset (when possible) - CDC FIFOs flushed cleanly (no partial packets)

RI-3: FIFO Flush - Selective FIFO flushing via control register: - Descriptor FIFO flush - MM2S data FIFO flush - S2MM data FIFO flush (per-channel)

RI-4: Initialization State

After reset deassertion, RAPIDS guarantees: - [x] All FIFOs empty - [x] All engines disabled (MM2S, S2MM channels) - [x] All error flags cleared - [x] Performance counters reset to 0 - [x] Descriptor queue empty - [x] AXI4 interfaces idle

1.3.4 Clock Constraints

Frequency Relationships

Minimum Ratios: - `network_clk` : `aclk` >= 1:1 (network can be slower, same, or faster) - `aclk` : `pclk` >= 2:1 (core typically 2-4× faster than APB)

Recommended: - `pclk` = 50-100 MHz (APB control) - `aclk` = 200-400 MHz (DDR3/DDR4 memory) - `network_clk` = 250-500 MHz (Delta Network fabric)

CDC FIFO Sizing: - For `network_clk` : `aclk` = 2:1 (network faster): Larger write-side headroom - For `network_clk` : `aclk` = 1:2 (memory faster): Larger read-side headroom

Jitter and Skew

Clock Jitter Tolerance: - ±100 ps peak-to-peak jitter (typical for on-chip PLLs) - CDC FIFOs absorb jitter via async crossing

Clock Skew: - Same-domain: <200 ps (within RAPIDS core on `aclk`) - Cross-domain: Not applicable (async FIFOs tolerate arbitrary skew)

1.3.5 Power Management (Optional)

Clock Gating: - Individual engine clock gating when disabled - Descriptor engine clock gated when queue empty - S2MM channel clock gating when inactive

Power Domains: - Core domain: aclk (always on during operation) - Control domain: pclk (can be gated when HIVE-C idle) - Network domain: network_clk (controlled by Delta Network fabric)

Next: Acronyms

Previous: Architectural Requirements

Back to: Index

Chapter 1.4: Acronyms and Abbreviations

A

APB - Advanced Peripheral Bus (AMBA 3 specification for low-bandwidth control interfaces)

AR - Address Read (AXI4 read address channel)

AMBA - Advanced Microcontroller Bus Architecture (ARM interconnect standard)

AW - Address Write (AXI4 write address channel)

AXI4 - Advanced eXtensible Interface version 4 (high-performance memory-mapped protocol)

AXI4-Lite (AXIL) - Subset of AXI4 for simple control/status register access

AXIS - AXI4-Stream (packet-oriented streaming protocol)

ASIC - Application-Specific Integrated Circuit

B

BFM - Bus Functional Model (testbench component)

B Channel - AXI4 write response channel

C

CDA - Compute Direct Access (packet type for descriptor injection from HIVE-C to RAPIDS)

CDC - Clock Domain Crossing (asynchronous interface between different clock domains)

Comparator - Timer comparison value in HPET peripheral

D

DDR - Double Data Rate (SDRAM memory technology: DDR2, DDR3, DDR4)

DDR2/DDR3/DDR4 - Generations of DDR memory (DDR2: 400-800 MT/s, DDR3: 800-1600 MT/s, DDR4: 1600-3200 MT/s)

Delta Network - 4×4 mesh Network-on-Chip for compute fabric interconnection

Descriptor - Data structure containing DMA transfer parameters (source, destination, length, control)

DMA - Direct Memory Access (hardware-managed memory transfers without CPU intervention)

DRAM - Dynamic Random Access Memory

DUT - Device Under Test (module being verified in testbench)

F

FIFO - First In First Out (queue buffer structure)

FPGA - Field-Programmable Gate Array

FSM - Finite State Machine (sequential logic with discrete states)

H

HIVE-C - High-performance Integrated VexRiscv Engine for Control (RISC-V master controller)

HPET - High Precision Event Timer (multi-timer peripheral)

I

ID - Transaction identifier (AXI4 signal for out-of-order response routing)

INCR - Incrementing burst type (AXI4 burst mode)

M

Mesh - Network topology with regular grid structure (4×4 for Delta Network)

MM2S - Memory-to-Stream (read from DDR memory, transmit to Delta Network)

MonBus - Monitor Bus (standardized 64-bit monitoring protocol)

N

Network Slave - RAPIDS interface receiving packets from Delta Network

Network Master - RAPIDS interface transmitting packets to Delta Network

NoC - Network-on-Chip (on-chip interconnect fabric)

P

Packet - Unit of data transmission on AXIS/Delta Network interfaces

PKT_CONFIG - Packet type for configuration (TUSER = 2’b10)

PKT_DATA - Packet type for compute data (TUSER = 2’b00)

PKT_STATUS - Packet type for monitoring/status (TUSER = 2’b11)

Priority - Descriptor scheduling weight (0=highest, 15=lowest in RAPIDS)

R

R Channel - AXI4 read data channel

RAPIDS - Rapid AXI Programmable In-band Descriptor System (DMA engine)

RISC-V - Open-standard instruction set architecture (used in HIVE-C)

Router - Delta Network component for packet routing between tiles

S

S2MM - Stream-to-Memory (receive from Delta Network, write to DDR memory)

Scatter-Gather - DMA technique using linked descriptor chains for non-contiguous transfers

SERV - System Event Reporting and Verification (monitoring subsystem)

SRAM - Static Random Access Memory (on-chip buffer)

T

TDEST - Transaction Destination (AXIS signal indicating target tile/destination)

TID - Transaction ID (AXIS signal for source identification or priority)

Tile - Compute element in Delta Network mesh (physical: 0-15, virtual: 16-17)

TLAST - Transaction Last (AXIS signal indicating final beat of packet)

TREADY - Transaction Ready (AXIS backpressure signal)

TUSER - Transaction User (AXIS signal encoding packet type)

TVALID - Transaction Valid (AXIS signal indicating valid data)

V

VexRiscv - RISC-V softcore CPU implementation (used in HIVE-C)

Virtual Tile - Non-physical network endpoint (RAPIDS=16, HIVE-C=17)

VC - Virtual Channel (independent flow control paths in router)

W

W Channel - AXI4 write data channel

W1C - Write-1-to-Clear (register bit cleared by writing 1, not 0)

WRAP - Wrapping burst type (AXI4 burst mode for circular addressing)

System-Specific Terms

Compute Tile - Processing element in Delta Network (tiles 0-15)

Descriptor Chain - Linked list of descriptors for scatter-gather DMA

Descriptor FIFO - Queue storing pending DMA descriptors (minimum 8 entries)

Inband Descriptor - Descriptor injected via CDA packets (no memory polling)

Virtual Tile 16 - RAPIDS DMA engine network address

Virtual Tile 17 - HIVE-C control processor network address

XY Routing - Dimension-ordered routing (X-dimension first, then Y-dimension)

Packet Type Summary

Acronym	TUSER	Full Name	Source	Destination	Purpose
PKT_DATA	2'b00	Data Packet	RAPIDS/Tiles	RAPIDS/Tiles	Compute data transfers
CDA	2'b01	Compute Direct Access	HIVE-C	RAPIDS	Descriptor injection
PKT_CONFIG	2'b10	Configuration Packet	HIVE-C/SERV	Routers/Tiles	Configuration
PKT_STATUS	2'b11	Status Packet	SERV	HIVE-C	Monitoring/status

Next: References

Previous: Clocks and Reset

Back to: Index

RTL Design Sherpa · Learning Hardware Design Through Practice GitHub · Documentation Index · MIT License

---

Chapter 1.5: References

Internal Specifications

RAPIDS Project Documentation

[1] RAPIDS Product Requirements Document `projects/components/rapids/PRD.md` Complete requirements specification for RAPIDS subsystem

[2] RAPIDS CLAUDE Guide `projects/components/rapids/CLAUDE.md` AI-specific guidance for working with RAPIDS subsystem

[3] RAPIDS Implementation Status `projects/components/rapids/IMPLEMENTATION_STATUS.md` Current development status and test results

[4] RAPIDS Specification Context Document `projects/components/rapids/docs/rapids_specific` Original HIVE/Delta Network integration specification (source for this update)

Delta Network Documentation

[5] Delta Network Specification v0.3 `projects/components/delta/docs/delta_spec/delta_index` Complete specification for Delta Network mesh Network-on-Chip

[6] Delta Router Architecture `projects/components/delta/docs/delta_spec/ch02_blocks/01_r` Detailed router architecture including XY routing and virtual channels

[7] Delta Packet Type Routing `projects/components/delta/docs/delta_spec/ch01_overview/03` Packet type definitions (PKT_DATA, CDA, PKT_CONFIG, PKT_STATUS)

[8] Delta External Entities `projects/components/delta/docs/delta_spec/ch03_interfaces/03` Integration specifications for RAPIDS and HIVE-C as virtual tiles

HIVE-C Documentation

[9] HIVE-C System Specification *(Location TBD - VexRiscv RISC-V control processor specification)*

[10] HIVE-C Software Guide *(Location TBD - Firmware development guide for descriptor generation)*

RTL Design Sherpa Project

[11] RTL Design Sherpa Root PRD `/PRD.md` Master product requirements document for repository

[12] RTL Design Sherpa CLAUDE Guide `/CLAUDE.md` Repository-wide guidance for AI-assisted development

[13] AMBA Protocol Documentation `rtl/amba/PRD.md` AMBA protocol infrastructure (AXI4, APB, AXIS monitors)

[14] Verification Architecture Guide `docs/VERIFICATION_ARCHITECTURE_GUIDE.md` Complete testbench architecture patterns and best practices

External Standards and Specifications

ARM AMBA Specifications

[15] AMBA AXI and ACE Protocol Specification (AXI4) ARM IHI 0022E, 2013 Complete specification for AXI4 memory-mapped protocol https://developer.arm.com/documentation/ihi0022/latest

[16] AMBA AXI4-Stream Protocol Specification ARM IHI 0051A, 2010 Specification for packet-oriented streaming protocol (AXIS) https://developer.arm.com/documentation/ihi0051/latest

[17] AMBA APB Protocol Specification v2.0 ARM IHI 0024C, 2010 Specification for Advanced Peripheral Bus (control/status registers) https://developer.arm.com/documentation/ihi0024/latest

[18] AMBA AXI4-Lite Protocol Subset of AXI4 specification (Chapter E1 of [15]) Simplified protocol for control/status register access

Memory Specifications

[19] DDR2 SDRAM Specification JESD79-2F, JEDEC Standard Double Data Rate 2 SDRAM specification

[20] DDR3 SDRAM Specification JESD79-3F, JEDEC Standard Double Data Rate 3 SDRAM specification (commonly used with RAPIDS)

[21] DDR4 SDRAM Specification JESD79-4C, JEDEC Standard Double Data Rate 4 SDRAM specification

RISC-V Architecture

[22] The RISC-V Instruction Set Manual, Volume I: User-Level ISA Version 20191213, RISC-V Foundation https://riscv.org/technical/specifications/

[23] VexRiscv RISC-V CPU Core GitHub: SpinalHDL/VexRiscv Open-source RISC-V implementation (used in HIVE-C) https://github.com/SpinalHDL/VexRiscv

Verification and Testbench

[24] CocoTB Documentation Coroutine-based cosimulation testbench environment for Python https://docs.cocotb.org/

[25] Verilator Manual Fast Verilog/SystemVerilog simulator https://verilator.org/guide/latest/

Design Patterns and Best Practices

Network-on-Chip Design

[26] Dally, W.J. and Towles, B., “Principles and Practices of Interconnection Networks” Morgan Kaufmann, 2004 Foundational text for NoC design principles

[27] Duato, J., Yalamanchili, S., and Ni, L., “Interconnection Networks: An Engineering Approach” Morgan Kaufmann, 2002 Deadlock-free routing algorithms (XY routing used in Delta Network)

DMA and Scatter-Gather

[28] PCI Express Base Specification Revision 4.0 PCI-SIG, 2017 Reference for descriptor-based DMA patterns

[29] Intel I/O Acceleration Technology (I/OAT) Specification Intel Corporation Advanced DMA engine architecture (industry reference)

Clock Domain Crossing

[30] Cummings, C.E., “Clock Domain Crossing (CDC) Design & Verification Techniques Using SystemVerilog” SNUG 2008 Best practices for CDC FIFO design

[31] Cummings, C.E., “Synthesis and Scripting Techniques for Designing Multi-Asynchronous Clock Designs” SNUG 2001 Gray code counters for async FIFO pointers

Related Projects

RTL Design Examples

[32] OpenPiton Research Platform Princeton University Open-source many-core processor with mesh NoC https://parallel.princeton.edu/openpiton/

[33] LowRISC Ibex Core lowRISC CIC Open-source RISC-V processor (similar class to VexRiscv) https://github.com/lowRISC/ibex

[34] Ariane RISC-V CPU ETH Zurich High-performance 64-bit RISC-V processor https://github.com/openhwgroup/cva6

Verification Frameworks

[35] PyUVM Python implementation of UVM (Universal Verification Methodology) https://github.com/pyuvm/pyuvm

[36] cocotb-coverage Functional coverage library for CocoTB https://github.com/mciepluc/cocotb-coverage

Document Revision History

Version	Date	Author	Changes
0.1	2025-10-19	Documentation Team	Initial creation with HIVE/Delta Network context

---

Next: Block Overview

Previous: Acronyms

Back to: Index

RTL Design Sherpa · Learning Hardware Design Through Practice GitHub · Documentation Index · MIT License

---

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

Figure 1: 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.

RTL Design Sherpa · Learning Hardware Design Through Practice GitHub · Documentation Index · MIT License

---

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

Figure 2: 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

Signal Name	Type	Width	Direction	Required	Description
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

Signal Name	Type	Width	Direction	Required	Description
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

Signal Name	Type	Width	Direction	Status	Description
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 & schedule
```

```verilog
// 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`

RTL Design Sherpa · Learning Hardware Design Through Practice GitHub · Documentation Index · MIT License

---

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

Figure 3: 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

Signal Name	Type	Width	Direction	Required	Description
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

Signal Name	Type	Width	Direction	Required	Description
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

Signal Name	Type	Width	Direction	Required	Description
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.

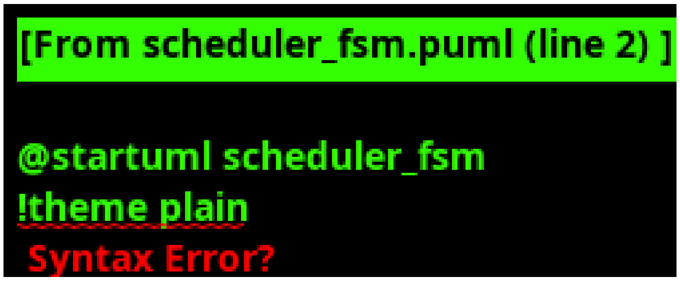
RDA Completion Interface

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



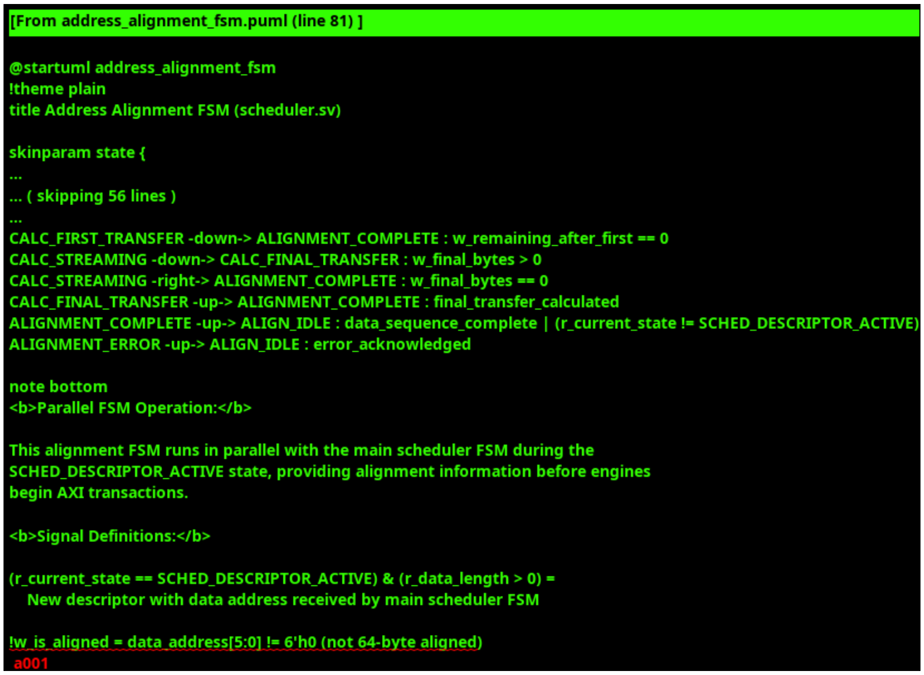
```
[From scheduler_fsm.puml (line 2) ]

@startuml scheduler_fsm
!theme plain
 Syntax Error?
```

Figure 4: Address Alignment FSM

Main Scheduler 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) ->
```



```
[From address_alignment_fsm.puml (line 81) ]

@startuml address_alignment_fsm
!theme plain
title Address Alignment FSM (scheduler.sv)

skinparam state {
...
... ( skipping 56 lines )
...
CALC_FIRST_TRANSFER -down-> ALIGNMENT_COMPLETE : w_remaining_after_first == 0
CALC_STREAMING -down-> CALC_FINAL_TRANSFER : w_final_bytes > 0
CALC_STREAMING -right-> ALIGNMENT_COMPLETE : w_final_bytes == 0
CALC_FINAL_TRANSFER -up-> ALIGNMENT_COMPLETE : final_transfer_calculated
ALIGNMENT_COMPLETE -up-> ALIGN_IDLE : data_sequence_complete | (r_current_state != SCHED_DESCRIPTOR_ACTIVE)
ALIGNMENT_ERROR -up-> ALIGN_IDLE : error_acknowledged

note bottom
<b>Parallel FSM Operation:</b>

This alignment FSM runs in parallel with the main scheduler FSM during the
SCHED_DESCRIPTOR_ACTIVE state, providing alignment information before engines
begin AXI transactions.

<b>Signal Definitions:</b>

(r_current_state == SCHED_DESCRIPTOR_ACTIVE) & (r_data_length > 0) =
   New descriptor with data address received by main scheduler FSM

!w_is_aligned = data_address[5:0] != 6'h0 (not 64-byte aligned)
 a001
```

Figure 5: Address Alignment FSM

Address Alignment FSM (Parallel Operation) **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)) beg
    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 boun
    logic [31:0]              first_transfer_bytes; // Bytes in alignment transfe
    logic [NUM_CHUNKS-1:0]    first_chunk_enables;  // Chunk pattern for alignmen
    logic [7:0]               optimal_burst_len;    // Optimal burst after alignm
    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

Bit Range	Field Name	Width	Description
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) beg
    // 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

    // 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

Signal Name	Width	Description
`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'h000000
                                       (cfg_initial_credit == 4'h0) ? 32'h0000000
                                       (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
```

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 :
assign ctrlwr_data = (r_current_state == SCHED_ISSUE_CTRLWR0) ? r_ctrlwr0_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=
           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_ctrlw
    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_c

// Error state recovery to IDLE
if ((r_current_state == SCHED_ERROR) && w_all_errors_clear && w_credit_available
    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 nee
    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.

RTL Design Sherpa · Learning Hardware Design Through Practice GitHub · Documentation Index · MIT License

---

Descriptor Engine

Overview The Descriptor Engine manages descriptor fetching and buffering operations with sophisticated dual-path processing for APB and CDA (Compute Direct Access packet) 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.

Delta Network Integration: - **CDA Interface:** The CDA (Compute Direct Access) packet interface receives CDA packets from HIVE-C via the Delta Network - **Inband Descriptor Injection:** HIVE-C injects descriptors as CDA packets (TUSER=2’b01) through Delta Network virtual tile 16 - **Priority Processing:** CDA packets processed with higher priority than APB requests for low-latency descriptor delivery - **Packet Type Validation:** CDA packets validated for TUSER=2’b01 before descriptor extraction

descriptor engine

Figure 6: descriptor engine

Key Features

- **Dual-Path Processing**: Handles both APB programming interface and CDA 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**: CDA 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

Signal Name	Type	Width	Direction	Required	Description
apb_addr	logic	`ADDR_WIDTH`	Input	Yes	Descriptor address

CDA Packet Interface (CDA Packets from HIVE-C via Delta Network)
Note: CDA (Compute Direct Access) interface receives CDA packets with TUSER=2'b01 from HIVE-C routed through Delta Network to RAPIDS virtual tile 16.

Signal Name	Type	Width	Direction	Required	Description
cda_valid	logic	1	Input	Yes	CDA packet valid (from HIVE-C)
cda_ready	logic	1	Output	Yes	Ready to accept CDA packet
cda_packet	logic	`DATA_WIDTH`	Input	Yes	CDA packet data (256-bit descriptor)
cda_channel	logic	`CHAN_WIDTH`	Input	Yes	Channel identifier from CDA packet

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_cda	logic	1	Output	Yes	CDA packet indicator

Signal Name	Type	Width	Direction	Required	Description
descriptor_cda_channel	logic	`CHAN_WIDTH`	Output	Yes	CDA 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

Signal Name	Type	Width	Direction	Required	Description
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 CDA 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:

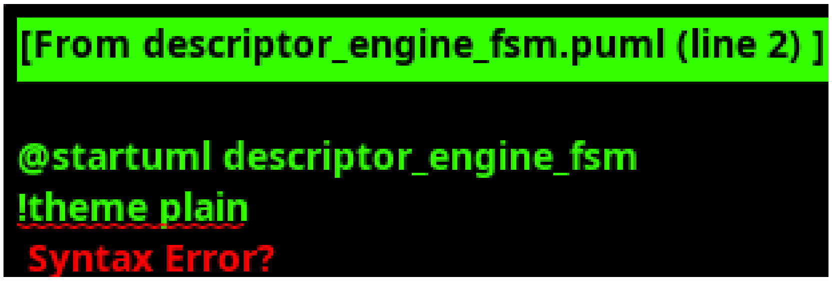
1. **CDA Packet Priority**: CDA packets from HIVE-C always processed before APB requests
    - **Rationale:** Low-latency descriptor injection from HIVE-C control processor
    - **Mechanism:** CDA packets bypass APB queue when both interfaces active
2. **Operation Exclusivity**: Only one operation type active at a time
3. **Channel Reset Coordination**: Graceful shutdown with AXI completion

Delta Network Integration Benefits: - **Zero-polling overhead:** HIVE-C pushes descriptors via CDA packets instead of RAPIDS polling memory - **Sub-microsecond latency:** CDA packets traverse Delta Network in ~10-20 cycles - **Priority-based scheduling:** CDA packet TID field carries descriptor priority (0=highest, 15=lowest)

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 CDA packet interface operations. The FSM manages the complete descriptor lifecycle from initial request validation through AXI read transactions, descriptor parsing, and final output generation.

Key States: - **DESC_IDLE**: Ready for APB or CDA 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 -



```
[From descriptor_engine_fsm.puml (line 2) ]

@startuml descriptor_engine_fsm
!theme plain
 Syntax Error?
```

Figure 7: Descriptor Engine FSM

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 CDA 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()
);
```

CDA Packet Skid Buffer

```
// CDA packet skid buffer (4-deep)
```

```
gaxi_skid_buffer #(
    .DATA_WIDTH(DATA_WIDTH + CHAN_WIDTH),
    .DEPTH(4),
    .INSTANCE_NAME("CDA_PKT_SKID")
) i_cda_packet_skid_buffer (
    .axi_aclk(clk),
    .axi_aresetn(rst_n),
    .wr_valid(cda_valid),
    .wr_ready(cda_ready),
    .wr_data({cda_packet, cda_channel}),
    .rd_valid(w_cda_skid_valid_out),
    .rd_ready(w_cda_skid_ready_out),
    .rd_data(w_cda_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
- **CDA 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 CDA + 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

RTL Design Sherpa · Learning Hardware Design Through Practice GitHub · Documentation Index · MIT License

---

Ctrlwr Engine

Overview The Ctrlwr Engine manages post-processing write operations for each virtual channel after descriptor completion. The module implements a four-state state machine that handles ctrlwr address writes with configurable data values to support notification, control, and cleanup operations with comprehensive timeout mechanisms and robust error handling.

ctrlwr engine

Figure 8: ctrlwr engine

Key Features

- **Four-State State Machine**: Comprehensive ctrlwr operation lifecycle management

- **Standardized Skid Buffer**: Uses gaxi_skid_buffer for consistent flow control
- **Null Address Support**: Graceful handling of conditional ctrlwr operations
- **Multi-Channel AXI Support**: Proper shared AXI interface with ID-based response routing
- **Address Validation**: 4-byte alignment checking with graceful error handling
- **Channel Reset Support**: Graceful shutdown with AXI transaction completion
- **Monitor Integration**: Comprehensive event reporting for ctrlwr operations
- **Flexible Control Writes**: Support for various use cases and data patterns

Interface Specification

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
`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

Scheduler FSM Interface

Signal Name	Type	Width	Direction	Required	Description
ctrlwr_valid	logic	1	Input	Yes	Ctrlwr operation request from FSM
ctrlwr_ready	logic	1	Output	Yes	Ctrlwr operation complete
ctrlwr_pkt_addr	logic	ADDR_WIDTH	Input	Yes	Ctrlwr write address
ctrlwr_pkt_data	logic	32	Input	Yes	Ctrlwr write data
ctrlwr_error	logic	1	Output	Yes	Ctrlwr operation error

Configuration Interface

Signal Name	Type	Width	Direction	Required	Description
cfg_channel_reset	logic	1	Input	Yes	Dynamic channel reset request

Status Interface

Signal Name	Type	Width	Direction	Required	Description
ctrlwr_engine_idle	logic	1	Output	Yes	Engine idle status indicator

Shared AXI4 Master Write Interface (32-bit)

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 (arbitrated)

Signal Name	Type	Width	Direction	Required	Description
aw_addr	logic	ADDR_WIDTH	Output	Yes	Write address
aw_len	logic	8	Output	Yes	Burst length - 1 (always 0)
aw_size	logic	3	Output	Yes	Transfer size (3’b010 for 4 bytes)
aw_burst	logic	2	Output	Yes	Burst type (2’b01 INCR)
aw_id	logic	AXI_ID_WIDTH	Output	Yes	Transaction ID (channel-based)
aw_lock	logic	1	Output	Yes	Lock type (always 0)
aw_cache	logic	4	Output	Yes	Cache attributes
aw_prot	logic	3	Output	Yes	Protection attributes
aw_qos	logic	4	Output	Yes	Quality of service
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	32	Output	Yes	Write data
w_strb	logic	4	Output	Yes	Write strobes (always 4’hF)
w_last	logic	1	Output	Yes	Write last (always 1)
b_valid	logic	1	Input	Yes	Write response valid
b_ready	logic	1	Output	Yes	Write response ready
b_id	logic	AXI_ID_WIDTH	Input	Yes	Response ID (channel identification)

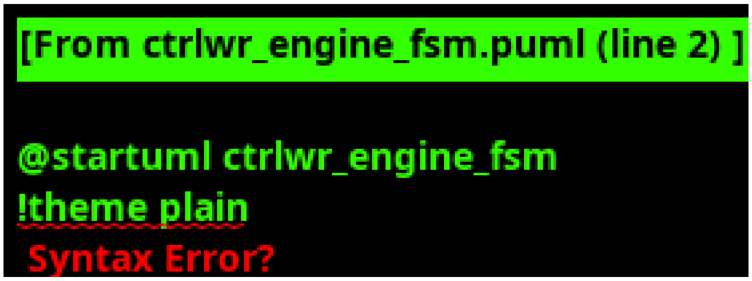
b_resp	logic	2	Input	Yes	Write response

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

Ctrlwr Engine FSM The Ctrlwr 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 ctrlwr address writes with configurable data values to support notification, control, and cleanup operations with comprehensive timeout mechanisms and robust error handling.

The FSM supports conditional ctrlwr through graceful null address handling, where operations are skipped immediately when ctrlwr addresses are 64'h0, enabling descriptor-driven conditional execution. Address validation ensures 4-byte alignment compliance, while the shared AXI interface uses ID-based response routing for proper multi-channel operation. Channel reset coordination provides graceful shutdown capabilities, completing any in-flight AXI write transactions before asserting idle status, enabling runtime reconfiguration without system disruption.



```
[From ctrlwr_engine_fsm.puml (line 2) ]

@startuml ctrlwr_engine_fsm
!theme plain
 Syntax Error?
```

Figure 9: Ctrlwr Engine FSM

State Definitions

State	Description
WRITE_IDLE	Ready for new ctrlwr operation request
WRITE_ISSUE_ADDR	Issue AXI write address and data phases
WRITE_WAIT_RESP	Wait for AXI write response
WRITE_ERROR	Handle error conditions

State Transitions

```
WRITE_IDLE -> WRITE_ISSUE_ADDR: Valid ctrlwr request with non-
null address
WRITE_IDLE -> WRITE_IDLE: Valid ctrlwr request with null address (skip operation
WRITE_IDLE -> WRITE_ERROR: Valid ctrlwr request with address error
WRITE_ISSUE_ADDR -> WRITE_WAIT_RESP: Both AXI address and data phases complete
WRITE_ISSUE_ADDR -> WRITE_ERROR: AXI transaction timeout
WRITE_WAIT_RESP -> WRITE_IDLE: AXI response received successfully
WRITE_WAIT_RESP -> WRITE_ERROR: AXI response indicates error
WRITE_ERROR -> WRITE_IDLE: Error acknowledged or channel reset
```

Operation Flow

1. **Request Reception**: Scheduler provides ctrlwr address and data
2. **Address Validation**: Check for null address and 4-byte alignment
3. **AXI Transaction**: Issue write address and data simultaneously
4. **Response Monitoring**: Wait for AXI response with correct channel ID
5. **Completion**: Signal ready to scheduler when operation complete

Ctrlwr Operation Types

Conditional Control Write (Null Address)

- **Detection**: Ctrlwr address is 64’h0
- **Behavior**: Operation is skipped, ready asserted immediately
- **Use Case**: Conditional ctrlwr based on descriptor content
- **Monitor Event**: Null ctrlwr operation completion

Standard Control Write

- **Detection**: Non-zero, 4-byte aligned address
- **Behavior**: Standard AXI write transaction executed
- **Use Case**: Notification, control register updates, cleanup operations
- **Monitor Event**: Ctrlwr write completion with success/error status

Address Error Handling

- **Detection**: Non-zero address not 4-byte aligned
- **Behavior**: Error state entered, no AXI transaction issued
- **Use Case**: Invalid descriptor content detection
- **Monitor Event**: Address validation error

AXI Transaction Management

Channel ID Encoding

```
// AXI ID construction for channel identification
assign r_expected_axi_id = {{(AXI_ID_WIDTH-CHAN_WIDTH){1'b0}}, CHANNEL_ID[CHAN_W
assign aw_id = r_expected_axi_id;
```

Write Transaction Properties

```
// AXI write attributes for 32-bit ctrlwr operations
assign aw_len = 8'h00;          // Single beat
assign aw_size = 3'b010;        // 4 bytes (32 bits)
assign aw_burst = 2'b01;        // INCR burst type
assign aw_lock = 1'b0;          // Normal access
assign w_strb = 4'hF;           // All bytes valid
assign w_last = 1'b1;           // Single beat transaction
```

Response Monitoring

```
// Response monitoring for correct channel
assign w_our_axi_response = b_valid && (b_id == r_expected_axi_id);
assign w_axi_response_error = (b_resp != 2'b00); // Not OKAY response

// Ready when waiting for our response
assign b_ready = (r_current_state == WRITE_WAIT_RESP) && w_our_axi_response;
```

Channel Reset Coordination The ctrlwr engine supports graceful channel reset:

Reset Behavior

1. **Block New Requests**: Stop accepting ctrlwr requests during reset
2. **Complete AXI Transaction**: Finish any in-flight AXI write operation
3. **Clear State**: Reset internal state and skid buffer
4. **Signal Completion**: Assert `ctrlwr_engine_idle` when complete

Reset Timing

- **AXI Safety**: Waits for AXI write completion in WRITE_WAIT_RESP state
- **FIFO Cleanup**: Clears ctrlwr request skid buffer
- **State Reset**: Returns FSM to WRITE_IDLE state
- **Idle Indication**: Provides clean idle signal for system coordination

```
// Channel reset coordination
assign w_safe_to_reset = (r_current_state == WRITE_IDLE) &&
                        w_fifo_empty &&
                        w_no_active_transaction;

assign ctrlwr_engine_idle = w_safe_to_reset && !r_channel_reset_active;
```

Error Handling

Error Sources

1. **Address Errors**: Non-4-byte aligned addresses
2. **AXI Response Errors**: SLVERR, DECERR from AXI infrastructure
3. **Timeout Errors**: AXI transaction timeout detection
4. **Protocol Errors**: Invalid AXI handshake sequences

Error Recovery

```
// Error state recovery conditions
assign w_error_recovery = error_acknowledged || r_channel_reset_active;

// Error persistence for debugging
always_ff @(posedge clk) begin
    if (!rst_n || r_channel_reset_active) begin
        r_ctrlwr_error <= 1'b0;
    end else if (w_error_condition_detected) begin
        r_ctrlwr_error <= 1'b1;
    end
end
```

Monitor Bus Events The ctrlwr engine generates comprehensive monitor events:

Error Events

- **Address Error**: Invalid address alignment detected
- **AXI Error**: AXI write response error conditions
- **Timeout**: AXI transaction timeout detection

- **Protocol Error**: Invalid operation sequence

Performance Events

- **Ctrlwr Write**: Successful ctrlwr write completion
- **Null Ctrlwr**: Null address ctrlwr operation skipped
- **Processing Latency**: Time from request to completion
- **Throughput**: Ctrlwr operation rate

Completion Events

- **Write Completion**: Standard ctrlwr write complete
- **Skip Completion**: Null address operation complete
- **Error Recovery**: Error condition resolution
- **Channel Reset**: Channel reset sequence completion

Input Buffer Management

Skid Buffer Architecture

```
// Standardized skid buffer for ctrlwr requests
gaxi_skid_buffer #(
    .DATA_WIDTH(ADDR_WIDTH + 32),  // Address + Data
    .SKID_DEPTH(4)                 // Configurable depth
) u_ctrlwr_req_skid (
    .axi_aclk(clk),
    .axi_aresetn(rst_n),
    .s_axis_tvalid(ctrlwr_valid),
    .s_axis_tready(ctrlwr_ready),
    .s_axis_tdata({ctrlwr_pkt_addr, ctrlwr_pkt_data}),
    .m_axis_tvalid(w_ctrlwr_req_skid_valid_out),
    .m_axis_tready(w_ctrlwr_req_skid_ready_out),
    .m_axis_tdata(w_ctrlwr_req_skid_dout)
);
```

Flow Control

```
// Ready signal generation based on state and reset
assign w_ctrlwr_req_skid_ready_out = (r_current_state == WRITE_IDLE) &&
                                     w_ctrlwr_req_skid_valid_out &&
                                     !r_channel_reset_active;
```

Performance Characteristics

Latency Analysis

- **Null Ctrlwr**: 1 cycle (immediate completion)
- **Standard Ctrlwr**: 4-8 cycles (AXI address + data + response)
- **Error Handling**: 2-4 cycles (error detection + recovery)
- **Channel Reset**: Variable (depends on AXI completion)

Throughput Analysis

- **Peak Rate**: 1 ctrlwr operation per 4-8 cycles
- **Sustained Rate**: Limited by AXI write bandwidth and arbitration
- **Multi-Channel**: Independent operation across channels
- **Efficiency**: >90% for back-to-back ctrlwr operations

AXI Utilization

- **Address Utilization**: 100% for valid operations
- **Data Utilization**: 100% (32-bit data on 32-bit interface)
- **Response Handling**: Selective response monitoring by channel ID
- **Error Rate**: <0.1% under normal operating conditions

Usage Guidelines

Control Write Patterns **Notification Control Write:**

```
// Notify completion to external processor
ctrlwr_addr = COMPLETION_REGISTER_BASE + (channel_id * 4);
ctrlwr_data = descriptor_completion_status;
```

Control Register Update:

```
// Update control registers based on stream state
ctrlwr_addr = CONTROL_REGISTER_BASE + (channel_id * 4);
ctrlwr_data = {reserved, eos_detected, error_flags, completion_count};
```

Cleanup Control Write:

```
// Clear status registers after processing
ctrlwr_addr = STATUS_REGISTER_BASE + (channel_id * 4);
ctrlwr_data = 32'h0000_0000;
```

Error Monitoring The ctrlwr engine provides comprehensive error detection: - Monitor address alignment errors for descriptor validation - Check AXI response codes for system health - Track timeout conditions for performance analysis - Use monitor events for debugging and optimization

Performance Optimization

- Use null addresses for conditional ctrlwr to reduce AXI traffic
- Batch multiple ctrlwr operations where possible
- Monitor AXI arbitration delays and adjust system priorities
- Configure skid buffer depth based on scheduler burst characteristics

RTL Design Sherpa · Learning Hardware Design Through Practice GitHub · Documentation Index · MIT License

---

Ctrlrd Engine

Overview The Ctrlrd Engine manages pre-processing read operations for each virtual channel before descriptor execution. The module implements a retry-capable read-and-compare mechanism that polls a control address until the read value matches an expected value, or a maximum retry count is exceeded. This enables synchronization, flag polling, and pre-condition validation before data operations begin.

ctrlrd engine

Figure 10: ctrlrd engine

Key Features

- **Read-and-Compare Mechanism**: Automatic retry until read data matches expected value
- **Configurable Retry Count**: Up to 511 retries via `cfg_ctrlrd_max_try[8:0]`
- **Mask Support**: Flexible bit-wise masking for partial value comparison
- **1µs Timeout Counter**: Microsecond-resolution timeout using scheduler_group 1µs tick
- **Standardized Skid Buffer**: Uses gaxi_skid_buffer for consistent flow control
- **Null Address Support**: Graceful handling of conditional ctrlrd operations
- **AXI4 Read Interface**: Standard 32-bit AXI4-Lite read transactions
- **Channel Reset Support**: Graceful shutdown with AXI transaction completion
- **Monitor Integration**: Comprehensive event reporting for ctrlrd operations

Interface Specification

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
`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

Scheduler FSM Interface

Signal Name	Type	Width	Direction	Required	Description
ctrlrd_valid	logic	1	Input	Yes	Ctrlrd operation request from scheduler
ctrlrd_ready	logic	1	Output	Yes	Ctrlrd operation complete (match or error)
ctrlrd_pkt_addr	logic	ADDR_WIDTH	Input	Yes	Ctrlrd read address
ctrlrd_pkt_data	logic	32	Input	Yes	Ctrlrd expected/comparison data
ctrlrd_pkt_mask	logic	32	Input	Yes	Ctrlrd bit mask for comparison

Signal Name	Type	Width	Direction	Required	Description
ctrlrd_error	logic	1	Output	Yes	Ctrlrd operation error (valid with ctrlrd_ready)
ctrlrd_result	logic	32	Output	Yes	Ctrlrd actual read result data

Configuration Interface

Signal Name	Type	Width	Direction	Required	Description
cfg_ctrlrd_max_try	logic	9	Input	Yes	Maximum retry count (0-511)
cfg_channel_reset	logic	1	Input	Yes	Dynamic channel reset request
tick_1us	logic	1	Input	Yes	1 microsecond tick from scheduler_group

Status Interface

Signal Name	Type	Width	Direction	Required	Description
ctrlrd_engine_idle	logic	1	Output	Yes	Engine idle status indicator
ctrlrd_retry_count	logic	9	Output	Yes	Current retry count value

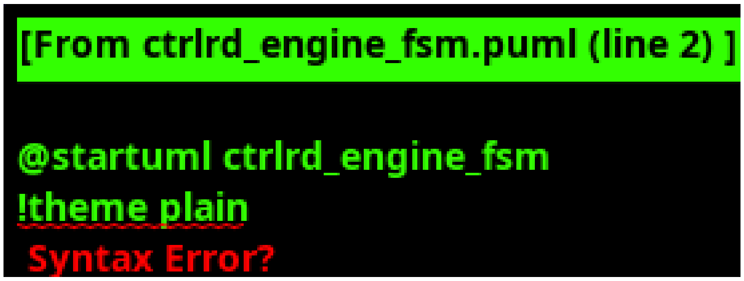
Shared AXI4 Read Interface (32-bit)

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'b010 for 4 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	32	Input	Yes	Read data
r_resp	logic	2	Input	Yes	Read response
r_last	logic	1	Input	Yes	Read last indicator
r_id	logic	AXI_ID_WIDTH	Input	Yes	Response ID (channel identification)

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

Ctrlrd Engine FSM The Ctrlrd Engine implements a retry-capable read-compare-retry finite state machine that manages control read operations before descriptor execution.



```
[From ctrlrd_engine_fsm.puml (line 2) ]

@startuml ctrlrd_engine_fsm
!theme plain
 Syntax Error?
```

Figure 11: Ctrlrd Engine FSM

State Definitions

State	Description
READ_IDLE	Ready for new ctrlrd operation request
READ_ISSUE_ADDR	Issue AXI read address phase
READ_WAIT_DATA	Wait for AXI read data response
READ_COMPARE	Compare read data with expected value
READ_RETRY_WAIT	Wait 1µs before retry attempt
READ_MATCH	Read data matched expected value (success)
READ_ERROR	Handle error conditions (max retries or AXI error)

State Transitions

```
READ_IDLE -> READ_ISSUE_ADDR: Valid ctrlrd request with non-
null address
READ_IDLE -> READ_MATCH: Valid ctrlrd request with null address (skip operation)
```

```
READ_ISSUE_ADDR -> READ_WAIT_DATA: AXI address phase complete
READ_WAIT_DATA -> READ_COMPARE: AXI read data received
READ_WAIT_DATA -> READ_ERROR: AXI response error (SLVERR, DECERR)
READ_COMPARE -> READ_MATCH: Read data matches expected value (success)
READ_COMPARE -> READ_RETRY_WAIT: Mismatch and retries remaining
READ_COMPARE -> READ_ERROR: Mismatch and max retries exceeded
READ_RETRY_WAIT -> READ_ISSUE_ADDR: 1μs elapsed, retry read
READ_MATCH -> READ_IDLE: Success acknowledged by scheduler
READ_ERROR -> READ_IDLE: Error acknowledged or channel reset
```

Operation Flow

1. **Request Reception**: Scheduler provides ctrlrd address, expected data, and mask
2. **Address Validation**: Check for null address (skip operation if 64’h0)
3. **AXI Read Transaction**: Issue read address and wait for data
4. **Data Comparison**: Compare `(r_data & mask) == (expected_data & mask)`
5. **Match Success**: Assert `ctrlrd_ready` without `ctrlrd_error`
6. **Mismatch Handling**:
    - If retries remaining: Wait 1μs, retry read
    - If max retries exceeded: Assert `ctrlrd_ready` with `ctrlrd_error`

Retry Mechanism

Retry Counter Management

```
// Retry counter (0 to cfg_ctrlrd_max_try)
logic [8:0] r_retry_count;
logic w_retries_remaining;

assign w_retries_remaining = (r_retry_count < cfg_ctrlrd_max_try);

// Increment on each retry attempt
if (r_current_state == READ_COMPARE && !w_data_match && w_retries_remaining) beg
    r_retry_count <= r_retry_count + 1;
end

// Reset on new operation or match
if (r_current_state == READ_IDLE || r_current_state == READ_MATCH) begin
    r_retry_count <= 9'h0;
end
```

1μs Retry Delay

```
// 1µs delay between retry attempts
// tick_1us provided by scheduler_group counter_freq_invariant
logic r_retry_wait_complete;

always_ff @(posedge clk) begin
    if (!rst_n) begin
        r_retry_wait_complete <= 1'b0;
    end else if (r_current_state == READ_RETRY_WAIT) begin
        if (tick_1us) begin
            r_retry_wait_complete <= 1'b1;
        end
    end else begin
        r_retry_wait_complete <= 1'b0;
    end
end

// Transition to retry after 1µs
if (r_current_state == READ_RETRY_WAIT && r_retry_wait_complete) begin
    w_next_state = READ_ISSUE_ADDR;
end
```

Data Comparison Logic

Masked Comparison

```
// Compare with mask support
logic [31:0] w_masked_expected;
logic [31:0] w_masked_actual;
logic w_data_match;

assign w_masked_expected = r_expected_data & r_mask;
assign w_masked_actual = r_axi_read_data & r_mask;
assign w_data_match = (w_masked_expected == w_masked_actual);

// Comparison in READ_COMPARE state
if (r_current_state == READ_COMPARE) begin
    if (w_data_match) begin
        w_next_state = READ_MATCH;  // Success
    end else if (w_retries_remaining) begin
        w_next_state = READ_RETRY_WAIT;  // Retry after 1µs
    end else begin
        w_next_state = READ_ERROR;  // Max retries exceeded
    end
end
```

Null Address Handling

```
// Null address detection (skip operation)
logic w_null_address;
assign w_null_address = (r_ctrlrd_addr == 64'h0);

// Skip to READ_MATCH immediately for null addresses
if ((r_current_state == READ_IDLE) && ctrlrd_valid && w_null_address) begin
    w_next_state = READ_MATCH;  // Skip operation, no error
end
```

AXI Transaction Management

Channel ID Encoding

```
// AXI ID construction for channel identification
assign r_expected_axi_id = {{(AXI_ID_WIDTH-CHAN_WIDTH){1'b0}}, CHANNEL_ID[CHAN_W
assign ar_id = r_expected_axi_id;
```

Read Transaction Properties

```
// AXI read attributes for 32-bit ctrlrd operations
assign ar_len = 8'h00;          // Single beat
assign ar_size = 3'b010;        // 4 bytes (32 bits)
assign ar_burst = 2'b01;        // INCR burst type
assign ar_lock = 1'b0;          // Normal access
assign ar_cache = 4'b0000;      // Device non-bufferable
assign ar_prot = 3'b000;        // Unprivileged, non-secure, data
assign ar_qos = 4'h0;           // No QoS
assign ar_region = 4'h0;        // Default region
```

Response Monitoring

```
// Response monitoring for correct channel
assign w_our_axi_response = r_valid && (r_id == r_expected_axi_id);
assign w_axi_response_error = (r_resp != 2'b00); // Not OKAY response

// Ready when waiting for our response
assign r_ready = (r_current_state == READ_WAIT_DATA) && w_our_axi_response;

// Error detection
if (r_current_state == READ_WAIT_DATA && w_our_axi_response) begin
    if (w_axi_response_error) begin
        w_next_state = READ_ERROR;  // AXI error (SLVERR, DECERR)
    end else begin
        r_axi_read_data <= r_data;
        w_next_state = READ_COMPARE;
    end
end
```

Channel Reset Coordination The ctrlrd engine supports graceful channel reset:

Reset Behavior

1. **Block New Requests**: Stop accepting ctrlrd requests during reset
2. **Complete AXI Transaction**: Finish any in-flight AXI read operation
3. **Clear State**: Reset internal state and skid buffer
4. **Signal Completion**: Assert `ctrlrd_engine_idle` when complete

Reset Timing

```
// Channel reset coordination
assign w_safe_to_reset = (r_current_state == READ_IDLE) &&
                        w_fifo_empty &&
                        w_no_active_transaction;

assign ctrlrd_engine_idle = w_safe_to_reset && !r_channel_reset_active;
```

Error Handling

Error Sources

1. **AXI Response Errors**: SLVERR, DECERR from AXI infrastructure
2. **Max Retries Exceeded**: Read data never matched expected value
3. **Timeout Errors**: Combined with retry count for overall timeout
4. **Protocol Errors**: Invalid AXI handshake sequences

Error Recovery

```
// Error state outputs
assign ctrlrd_ready = (r_current_state == READ_MATCH) ||
                     (r_current_state == READ_ERROR);
assign ctrlrd_error = (r_current_state == READ_ERROR);

// Error recovery on acknowledgment
if (r_current_state == READ_ERROR && ctrlrd_ready && ctrlrd_valid) begin
    w_next_state = READ_IDLE;
end

// Or on channel reset
```

```
if (r_channel_reset_active && r_current_state == READ_ERROR) begin
    w_next_state = READ_IDLE;
end
```

Monitor Bus Events The ctrlrd engine generates comprehensive monitor events:

Error Events

- **AXI Read Error**: AXI read response error (SLVERR, DECERR)
- **Max Retries Exceeded**: Retry count exceeded without match
- **Timeout**: Overall operation timeout
- **Protocol Error**: Invalid operation sequence

Performance Events

- **Ctrlrd Match**: Successful read-compare match
- **Retry Count**: Number of retry attempts before success
- **Null Ctrlrd**: Null address ctrlrd operation skipped
- **Processing Latency**: Time from request to completion

Completion Events

- **Match Completion**: Read data matched expected value
- **Skip Completion**: Null address operation complete
- **Error Recovery**: Error condition resolution
- **Channel Reset**: Channel reset sequence completion

Input Buffer Management

Skid Buffer Architecture

```
// Standardized skid buffer for ctrlrd requests
gaxi_skid_buffer #(
    .DATA_WIDTH(ADDR_WIDTH + 32 + 32),  // Address + Data + Mask
    .SKID_DEPTH(4)                      // Configurable depth
) u_ctrlrd_req_skid (
    .axi_aclk(clk),
    .axi_aresetn(rst_n),
    .s_axis_tvalid(ctrlrd_valid),
    .s_axis_tready(ctrlrd_ready),
    .s_axis_tdata({ctrlrd_pkt_addr, ctrlrd_pkt_data, ctrlrd_pkt_mask}),
    .m_axis_tvalid(w_ctrlrd_req_skid_valid_out),
    .m_axis_tready(w_ctrlrd_req_skid_ready_out),
    .m_axis_tdata(w_ctrlrd_req_skid_dout)
);
```

Flow Control

```
// Ready signal generation based on state and reset
assign w_ctrlrd_req_skid_ready_out = (r_current_state == READ_IDLE) &&
                                      w_ctrlrd_req_skid_valid_out &&
                                      !r_channel_reset_active;
```

Performance Characteristics

Latency Analysis

- **Null Ctrlrd**: 1 cycle (immediate completion)
- **Match on First Try**: 6-10 cycles (AXI address + data + compare)
- **Match After Retries**: (6-10 + N x 1µs) where N = retry count
- **Max Retries Exceeded**: (6-10 + cfg_ctrlrd_max_try x 1µs) + error handling

Retry Behavior

- **1µs Delay**: Fixed 1 microsecond delay between retry attempts
- **Max Retries**: Configurable 0-511 via `cfg_ctrlrd_max_try[8:0]`
- **Total Timeout**: Up to 511µs maximum retry duration
- **Success Rate**: Depends on external flag update frequency

AXI Utilization

- **Address Utilization**: 100% for valid operations
- **Data Utilization**: 100% (32-bit data on 32-bit interface)
- **Response Handling**: Selective response monitoring by channel ID
- **Retry Overhead**: 1µs minimum between attempts (prevents bus saturation)

Usage Guidelines

Control Read Patterns   Flag Polling:

```
// Poll for ready flag before descriptor execution
ctrlrd_addr = STATUS_REGISTER_BASE + (channel_id * 4);
ctrlrd_data = 32'h0000_0001;  // Expect bit 0 set
ctrlrd_mask = 32'h0000_0001;  // Check only bit 0
```

Multi-bit Status Check:

```
// Check multiple status bits before proceeding
ctrlrd_addr = CONTROL_REGISTER_BASE + (channel_id * 4);
ctrlrd_data = 32'h0000_00F0;  // Expect bits [7:4] all set
ctrlrd_mask = 32'h0000_00F0;  // Check only bits [7:4]
```

Synchronization Check:

```
// Wait for external synchronization event
ctrlrd_addr = SYNC_REGISTER_BASE;
ctrlrd_data = EXPECTED_SYNC_VALUE;
ctrlrd_mask = 32'hFFFF_FFFF;  // Check all bits
```

Retry Configuration Configure retry count based on expected flag update frequency:

```
// Fast flags (updated every few microseconds)
cfg_ctrlrd_max_try = 9'd10;   // 10µs maximum wait

// Medium flags (updated every ~100µs)
cfg_ctrlrd_max_try = 9'd100;  // 100µs maximum wait

// Slow flags (updated every millisecond)
cfg_ctrlrd_max_try = 9'd511;  // 511µs maximum wait (max supported)
```

Error Monitoring The ctrlrd engine provides comprehensive error detection: - Monitor AXI response codes for bus errors - Track retry count for performance analysis - Check for max retries exceeded to detect stuck conditions - Use monitor events for debugging and optimization

Performance Optimization

- Use appropriate masks to check only relevant bits
- Configure retry count based on expected flag update frequency
- Monitor retry statistics to optimize polling intervals
- Use null addresses for conditional ctrlrd to reduce AXI traffic

Integration with Scheduler Group

1µs Counter Integration The scheduler_group provides a shared 1µs tick signal to all ctrlrd engines:

```
// At scheduler_group level (scheduler_group.sv)
counter_freq_invariant #(
    .WIDTH(32),
    .FREQ_HZ(1_000_000)  // 1 MHz = 1µs period
) u_1us_counter (
    .clk(clk),
    .rst_n(rst_n),
    .tick(tick_1us)  // Shared 1μs tick to all ctrlrd engines
);

// Broadcast to all channels
for (genvar i = 0; i < NUM_CHANNELS; i++) begin
    assign ctrlrd_engine[i].tick_1us = tick_1us;
end
```

Channel Array Structure

```
// Scheduler group instantiates ctrlrd engines for each channel
generate
    for (genvar i = 0; i < NUM_CHANNELS; i++) begin : gen_ctrlrd_engines
        ctrlrd_engine #(
            .CHANNEL_ID(i),
            .NUM_CHANNELS(NUM_CHANNELS),
            .ADDR_WIDTH(ADDR_WIDTH),
            .AXI_ID_WIDTH(AXI_ID_WIDTH)
        ) u_ctrlrd_engine (
            .clk(clk),
            .rst_n(rst_n),

            // From scheduler
            .ctrlrd_valid(scheduler[i].ctrlrd_valid),
            .ctrlrd_ready(scheduler[i].ctrlrd_ready),
            .ctrlrd_pkt_addr(scheduler[i].ctrlrd_addr),
            .ctrlrd_pkt_data(scheduler[i].ctrlrd_data),
            .ctrlrd_pkt_mask(scheduler[i].ctrlrd_mask),
            .ctrlrd_error(scheduler[i].ctrlrd_error),
            .ctrlrd_result(scheduler[i].ctrlrd_result),

            // Configuration
            .cfg_ctrlrd_max_try(cfg_ctrlrd_max_try[i]),
            .cfg_channel_reset(cfg_channel_reset[i]),
            .tick_1us(tick_1us),  // Shared 1μs tick

            // Shared AXI read interface (arbitrated)
            .ar_valid(ctrlrd_ar_valid[i]),
            .ar_ready(ctrlrd_ar_ready[i]),
            // ... AXI signals ...

            // Monitor bus
            .mon_valid(ctrlrd_mon_valid[i]),
            .mon_ready(ctrlrd_mon_ready[i]),
            .mon_packet(ctrlrd_mon_packet[i])
        );
    end
endgenerate
```

This ctrlrd engine architecture provides robust flag polling and synchronization capabilities with configurable retry mechanisms and comprehensive error handling.

RTL Design Sherpa · Learning Hardware Design Through Practice GitHub · Documentation Index · MIT License

---

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

Figure 12: 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

Signal Name	Type	Width	Direction	Required	Description
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

Signal Name	Type	Width	Direction	Required	Description
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

Signal Name	Type	Width	Direction	Required	Description
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)

Parameter	Default Value	Description
`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

Signal Name	Type	Width	Direction	Required	Description
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

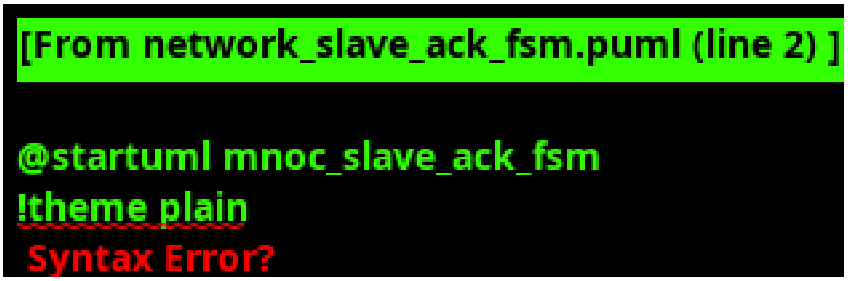
Signal Name	Type	Width	Direction	Required	Description
error_header_parity	logic	1	Output	Yes	Header parity error
error_body_parity	logic	1	Output	Yes	Body parity error
error_buffer_overflow	logic	1	Output	Yes	Buffer overflow error
error_ack_lost	logic	1	Output	Yes	ACK lost error
error_channel_id	logic	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.

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



```
[From network_slave_ack_fsm.puml (line 2) ]

@startuml mnoc_slave_ack_fsm
!theme plain
 Syntax Error?
```

Figure 13: Network Slave FSM

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 -
      ↓              ↓                    ↓                    ↓              ↓
   Parity Check   Protocol Check     FC/TS/RAW/RDA          FUB/RDA        FIFO-
based
   Format Valid   Address Valid      Type Detection       Interfaces      ACK Syste
```

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 add
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

```systemverilog
// 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:

```systemverilog
// 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

RTL Design Sherpa · Learning Hardware Design Through Practice GitHub · Documentation Index · MIT License

---

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

Parameter	Default Value	Description
`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

Signal Name	Type	Width	Direction	Required	Description
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

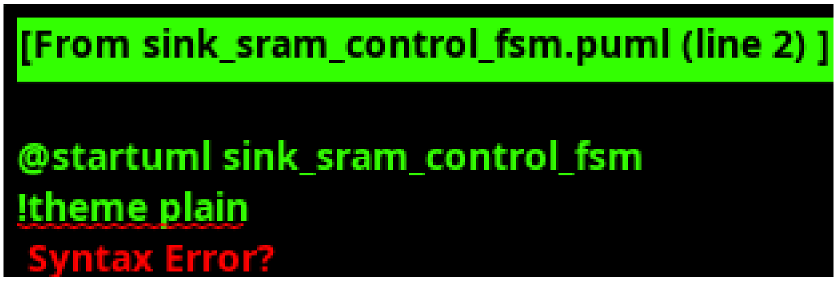
Signal Name	Type	Width	Direction	Required	Description
eos_pending	logic	CHANNELS	Output	Yes	EOS pending per channel

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.



```
[From sink_sram_control_fsm.puml (line 2) ]

@startuml sink_sram_control_fsm
!theme plain
 Syntax Error?
```

Figure 14: 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
    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 - OVERFLO
```

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

Signal Name	Type	Width	Direction	Required	Description
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

Signal Name	Type	Width	Direction	Required	Description
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

Signal Name	Type	Width	Direction	Required	Description
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 B
        ↓                      ↓                       ↓                      ↓
   Round-Robin           Zero-Cycle Issue        Perfect AXI Streaming    Backpr
   Fair Selection         No FSM Overhead        Optimal Bandwidth        Clean
```

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

RTL Design Sherpa · Learning Hardware Design Through Practice GitHub · Documentation Index · MIT License

---

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

Figure 15: 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

Signal Name	Type	Width	Direction	Required	Description
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

Signal Name	Type	Width	Direction	Required	Description
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 (back-pressure)
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

---

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

Signal Name	Type	Width	Direction	Required	Description
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

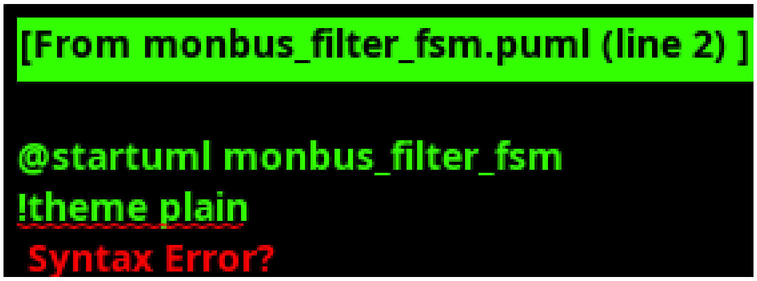
- **Monitor Bus Arbiter**: Round-robin arbitration between source and sink streams
- **Packet Filter**: Multi-level filtering based on protocol, packet type, and event codes
- **Error/Interrupt FIFO**: Stores filtered events requiring immediate attention
- **Master Write FIFO**: Stores filtered events 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.



```
[From monbus_filter_fsm.puml (line 2) ]

@startuml monbus_filter_fsm
!theme plain
 Syntax Error?
```

Figure 16: 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

RTL Design Sherpa · Learning Hardware Design Through Practice GitHub · Documentation Index · MIT License

---

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

FSM Name	Module File	States	PlantUML Status	Purpose
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

FSM Name	Module File	States	PlantUML Status	Purpose
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.

RTL Design Sherpa · Learning Hardware Design Through Practice GitHub · Documentation Index · MIT License

---

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

Signal	IO	Description
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

Signal	IO	Description
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

Signal	IO	Description
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

Signal	IO	Description
`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

Signal	IO	Description
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

Custom Network v2.1	AXIS4 v3.0	Notes
`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

RTL Design Sherpa · Learning Hardware Design Through Practice GitHub · Documentation Index · MIT License

---

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

Signal	Width	Direction	Required Values	Description
`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

Signal	Width	Direction	Required Values	Description
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

Response Type	Usage	Description
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

Benefit Area	Simplification	Impact
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

Protocol Rule	Requirement	Description
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

Validation Area	Requirements
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

Parameter	Value	Description
Write Data	0xDEADBEEF-CAFEBABE	64-bit data value
Required Settings	aw_addr=0x1000, aw_prot=3'b000, w_data=0xDEAD-BEEF-CAFEBABE, 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

Step	Action	Signal States
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

Consideration	Impact	Description
Efficiency	64-bit transfers maximize data efficiency	Modern system optimization

RTL Design Sherpa · Learning Hardware Design Through Practice GitHub · Documentation Index · MIT License

---

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**

Interface Group	Data Width	Address Width	ID Width	User Width	Transfer Mode
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

Step	Address	Size	AxSIZE	Length	Bytes Transferred	Notes
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

Transfer Size	Address Offset	Chunk Pattern	Description
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

Parameter	Formula	Description
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	EX-OKAY	Exclusive access success	Bus-width aligned exclusive	4-byte aligned exclusive
2’b10	SLVERR	Slave error	Slave-specific error	Slave-specific error
2’b11	DE-CERR	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

Interface Type	Recommended Mode	Rationale
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)

Aspect	Requirement
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 (Delta Network Integration)

TUSER[1:0] - Delta Network Packet Type Encoding:

RAPIDS enforces strict packet type filtering based on TUSER[1:0] encoding as defined by the Delta Network specification:

TUSER[1:0]	Packet Type	Full Name	Accepted By	Sent By	Purpose
2'b00	**PKT_DATA**	Data Packet	RAPIDS, Compute Tiles	RAPIDS, Compute Tiles	Compute data transfers
2'b01	**CDA**	Compute Direct Access	RAPIDS only	HIVE-C only	Descriptor injection
2'b10	**PKT_CONFIG**	Configuration Packet	Tile Routers	HIVE-C, SERV	Configuration
2'b11	**PKT_STATUS**	Status Packet	HIVE-C	SERV monitors	Monitoring/status

RAPIDS Interface Packet Type Rules:

RAPIDS Interface	Direction	Accepts	Sends	Rejects
AXIS CDA Input	Slave	CDA (2’b01)	-	PKT_DATA, PKT_CONFIG, PKT_STATUS
AXIS Data Output (MM2S)	Master	-	PKT_DATA (2’b00)	CDA, PKT_CONFIG, PKT_STATUS
AXIS Data Input (S2MM)	Slave	PKT_DATA (2’b00)	-	CDA, PKT_CONFIG, PKT_STATUS

Packet Type Validation: - **Invalid packet rejection:** RAPIDS asserts TREADY=0 when invalid TUSER value detected - **Error reporting:** Sets error flag in status register - **Interrupt generation:** Generates interrupt (if enabled) on packet type violation - **Hardware enforcement:** RAPIDS cannot transmit invalid packet types (design constraint)

Additional TUSER Fields (RAPIDS-Specific):

For PKT_DATA and CDA packets, RAPIDS may use additional TUSER bits beyond [1:0]:

Bits	Field	Description	Usage
`[1:0]`	**Packet Type**	PKT_DATA or CDA	**Mandatory** - Delta Network routing
`[5:2]`	**Priority**	0-15 priority level	Optional - Descriptor scheduling
`[7:6]`	**Reserved**	Future use	-

Usage Pattern: - TUSER valid on **every beat** of packet (Delta Network requirement) - TUSER[1:0] must be consistent across all beats of same packet - Downstream logic validates TUSER[1:0] on every beat for protocol compliance - Priority field (TUSER[5:2]) used for CDA packet scheduling in descriptor engine

---

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

Benefit Area	Advantage	Impact
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

Timing Parameter	Requirement	Description
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 (Delta Network Integration)

Interface Assignments

RAPIDS Interface	AXIS Role	Direction	Width	Packet Types	Delta Network Connection
AXIS CDA Input	Slave (RX)	Input	128-bit	CDA (2’b01) only	From HIVE-C via Delta Network
AXIS Data Output (MM2S)	Master (TX)	Output	128-bit	PKT_DATA (2’b00) only	To compute tiles via Delta Network
AXIS Data Input (S2MM)	Slave (RX)	Input	128-bit	PKT_DATA (2’b00) only	From compute tiles via Delta Network
Internal Source Stream	Master (TX)	Internal	512-bit	N/A - Internal datapath	SRAM to MM2S engine

RAPIDS Interface	AXIS Role	Direction	Width	Packet Types	Delta Network Connection
Internal Sink Stream	Slave (RX)	Internal	512-bit	N/A - Internal datapath	S2MM engine to SRAM

Key Points: - **Delta Network interfaces:** 128-bit TDATA width (standard Delta Network data width) - **Internal datapaths:** 512-bit TDATA width (high-bandwidth SRAM access) - **TUSER enforcement:** Hardware validates packet types on all Delta Network interfaces - **Virtual Tile 16:** RAPIDS addressed as tile 16 in Delta Network topology

Packet Processing Flow

CDA Descriptor Path (HIVE-C -> RAPIDS):

```
HIVE-C (VexRiscv RISC-V)
    ↓ CDA packet (TUSER=2'b01)
Delta Network (routes to tile 16)
    ↓ 128-bit AXIS
RAPIDS AXIS CDA Input (packet type validation)
    ↓ Descriptor extraction
Descriptor FIFO (256-bit descriptors)
    ↓ Parsed descriptors
Scheduler (executes MM2S or S2MM operations)
```

S2MM Path (Compute Tiles -> Memory):

```
Compute Tiles (0-15)
    ↓ PKT_DATA packet (TUSER=2'b00, TID=source tile)
Delta Network (routes to tile 16)
    ↓ 128-bit AXIS
RAPIDS AXIS Data Input (S2MM)
    ↓ Packet type validation (TUSER=2'b00)
    ↓ TID -> Channel routing (TID[3:0] = channel ID)
S2MM Channel FIFO (per-channel buffering)
    ↓ Internal 512-bit datapath
Sink SRAM Control (buffering)
    ↓ AXI4 Write
DDR Memory
```

MM2S Path (Memory -> Compute Tiles):

```
DDR Memory
    ↓ AXI4 Read
Source SRAM Control (buffering)
```

```
    ↓ Internal 512-bit datapath
MM2S Data FIFO
    ↓ Packet formation
RAPIDS AXIS Data Output (MM2S)
    ↓ PKT_DATA packet (TUSER=2'b00, TDEST=target tile)
Delta Network (routes to tile 0-15)
    ↓ 128-bit AXIS
Compute Tiles (destination tile)
```

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

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

Aspect	Requirement
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

Value	Name	Purpose	Applicable Protocols
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

Code	Event Name	Description
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

Code	Event Name	Description
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

Code	Event Name	Description
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

Code	Event Name	Description
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

Code	Event Name	Description
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

Code	Event Name	Description
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

Field	Width	Description	Protocol Usage
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

Field	Width	Description	Protocol
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

ACK Type	Value	Description
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]

Decoding Step	Method
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

Function	Parameter	Description
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

Function	Parameter	Description
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

Signal	Width	Purpose
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.

RTL Design Sherpa · Learning Hardware Design Through Practice GitHub · Documentation Index · MIT License

Programming

TBD