

# Table of Contents

## Stream Index

**Generated:** 2025-12-13

## STREAM Specification Index

**Version:** 0.90 **Date:** 2025-11-22 **Purpose:** Complete technical specification for STREAM subsystem

---

## Document Organization

**Note:** All chapters linked below for automated document generation.

### Chapter 1: Overview

- Architecture
- Top-Level Port List
- Clocks and Reset

### Chapter 2: Functional Blocks

**Macro/Integration Level:** - Stream Core - Scheduler Group Array - Scheduler Group

**Control Path:** - Scheduler - Descriptor Engine

**Read Datapath:** - AXI Read Engine - Stream Alloc Control

**SRAM Buffering:** - SRAM Controller - SRAM Controller Unit - Stream Latency Bridge

**Write Datapath:** - Stream Drain Control - AXI Write Engine

**Configuration & Monitoring:** - APB to Descriptor Router - APB Config - Performance Profiler - MonBus AXIL Group

### Chapter 3: Interfaces

- Interface Overview

- AXI4 Interface Specification
- AXIL4 Interface Specification
- APB Programming Interface
- MonBus Interface Specification

## Chapter 4: Registers

- Register Map

## Chapter 5: Programming

- Programming Guide

## Chapter 6: Configuration Reference

- Complete Configuration Guide
- 

## Quick Reference

### Functional Unit Blocks (FUB)

Module	File	Purpose	Lines	Status
descriptor_engine	stream_fub/_descriptor_engine.sv	Descriptor fetch/parse (256-bit)	~300	[Done]
scheduler	stream_fub/_scheduler.sv	Transfer coordinator	~400	[Done] Created (corrected)
axi_read_engine	stream_fub/_axi_read_engine.sv	AXI read master	~350	[Done] Created (no FSM - streaming pipeline)
axi_write_engine	stream_fub/_axi_write_engine.sv	AXI write master	~400	[Done] Created (no FSM - streaming pipeline)
sram_controller	stream_fub/_sram_controller.sv	Per-channel buffer management	~350	[Done] Created (no FSM - pointer arithmetic + pre-allocation)

Module	File	Purpose	Lines	Status
simple_sram	stream_fub/ simple_sram.sv	Dual-port SRAM primitive	~150	[Done]
perf_profiler	stream_fub/ perf_profiler.sv	Channel performance profiling	~400	[Done] Dual-mode timestamp/elapsed tracking

## Integration Blocks (MAC)

Module	File	Purpose	Lines	Status
channel_arbiter	stream_macro/ channel_arbiter.sv	Priority arbitration	~200	[Pending] To be created
apb_config	regs/stream_regs.rdl + wrapper	Config registers	~350	[Future] PeakRDL-based
monbus_axil_group	stream_macro/ monbus_axil_group.sv	MonBus + AXIL	~800	[Done]
stream_top	stream_macro/ stream_top.sv	Top-level	~500	[Pending] To be created

## Performance Modes (AXI Engines)

STREAM AXI engines support three performance modes via compile-time parameters, offering area/performance trade-offs from tutorial simplicity to datacenter throughput.

### Holistic Overview: V1/V2/V3 Working Together

**Design Philosophy:** - **V1 (Low):** Tutorial-focused, simple architecture, minimal area - **V2 (Medium):** Command pipelining hides memory latency, 6.7x throughput improvement - **V3 (High):** Out-of-order completion maximizes memory controller efficiency, 7.0x throughput

**Key Insight:** The benefit scales with memory latency. For low-latency SRAM (2-3 cycles), V1 achieves 40% throughput. For high-latency DDR4 (70-100 cycles), V1 drops to 14% but V2/V3 maintain 94-98% throughput.

**Parameterization Strategy:** - ENABLE\_CMD\_PIPELINE = 0: V1 (default, tutorial mode) - ENABLE\_CMD\_PIPELINE = 1: V2 (command pipelined, best area efficiency) - ENABLE\_CMD\_PIPELINE = 1, ENABLE\_000\_DRAIN = 1 (write) or ENABLE\_000\_READ = 1 (read): V3 (out-of-order, maximum throughput)

### V1 - Low Performance (Single Outstanding Transaction)

**Architecture:** Single-burst-per-request, blocking on completion - **Area:** ~1,250 LUTs per engine - **Throughput:** 0.14 beats/cycle (DDR4), 0.40 beats/cycle (SRAM) - **Outstanding Txns:** 1 - **Use Case:** Tutorial examples, embedded systems, low-latency SRAM - **Key Feature:** Zero-bubble streaming pipeline (NO FSM!)

**Blocking Behavior:** - Write: Blocks on B response before accepting next request - Read: Blocks on R last before accepting next request - Simple control: 3 flags (r\_ar\_inflight, r\_ar\_valid, completion tracking)

### V2 - Medium Performance (Command Pipelined)

**Architecture:** Command queue decouples command acceptance from data completion - **Area:** ~2,000 LUTs per engine (1.6x increase) - **Throughput:** 0.94 beats/cycle (DDR4), 0.85 beats/cycle (SRAM) - **Outstanding Txns:** 4-8 (command queue depth) - **Use Case:** General-purpose FPGA, DDR3/DDR4, balanced area/performance - **Key Feature:** Hides memory latency via pipelining (6.7x improvement)

**Command Pipelining:** - Write: AW queue + W drain FSM + B scoreboard (async response tracking) - Read: AR queue + R in-order reception (simpler than write, no B channel) - Per-command SRAM pointers enable multiple bursts from same channel

**Best Area Efficiency:** 4.19x throughput per unit area (6.7x throughput / 1.6x area)

### V3 - High Performance (Out-of-Order Completion)

**Architecture:** OOO command selection maximizes memory controller efficiency - **Area:** ~3,500-4,000 LUTs per engine (2.8-3.2x increase) - **Throughput:** 0.98 beats/cycle (DDR4), 0.92 beats/cycle (SRAM) - **Outstanding Txns:** 8-16 (larger command queue) - **Use Case:** Datacenter, ASIC, HBM2, high-performance

memory controllers - **Key Feature:** Flexible drain order based on SRAM data availability (7.0x improvement)

**Out-of-Order Mechanisms:** - Write: OOO W drain (select command with SRAM data ready), AXI ID matching for B responses - Read: OOO R reception (match m\_axi\_rid to queue entry), independent SRAM write per command - Transaction ID structure: {counter, channel\_id} preserves channel routing for MonBus

**Why OOO is Naturally Supported:** 1. Per-channel SRAM partitioning (no cross-channel hazards) 2. Per-command pointer tracking (no pointer collision) 3. Transaction ID matching (channel ID in lower bits for routing)

### Performance Comparison Summary

Configuration	Area	DDR4 Throughput	SRAM Throughput	Area Efficiency	Use Case
V1	1.0x	0.14 beats/cycle (1.0x)	0.40 beats/cycle (1.0x)	1.00	Tutorial, embeded
V2	1.6x	0.94 beats/cycle (6.7x)	0.85 beats/cycle (2.1x)	4.19	General FPGA
V3	2.8x	0.98 beats/cycle (7.0x)	0.92 beats/cycle (2.3x)	2.50	Datacenter, ASIC

**Key Takeaway:** V2 offers best area efficiency (4.19x) for typical FPGA use cases. V3 provides marginal throughput improvement (7.0x vs 6.7x) at higher area cost, justified only for high-performance memory controllers that support out-of-order responses.

---

### Clock and Reset Summary

#### Clock Domains

Clock	Frequency	Usage
aclk	100-500 MHz	Primary - all STREAM logic, AXI/AXIL interfaces

Clock	Frequency	Usage
pclk	50-200 MHz	APB configuration interface (may be async to aclk)

### Reset Signals

Reset	Polarity	Type	Usage
aresetn	Active-low	Async assert, sync deassert	Primary - all STREAM logic
presetn	Active-low	Async assert, sync deassert	APB configuration interface

See: [Clocks and Reset](#) for complete timing specifications

---

## Interface Summary

### External Interfaces

Interface	Type	Width	Purpose
APB	Slave	32-bit	Configuration registers
AXI (Descriptor)	Master	256-bit	Descriptor fetch
AXI (Read)	Master	512-bit (param)	Source data read
AXI (Write)	Master	512-bit (param)	Destination data write
AXIL (Slave)	Slave	32-bit	Error/interrupt FIFO access
AXIL (Master)	Master	32-bit	MonBus packet logging to memory
IRQ	Output	1-bit	Error interrupt

## Internal Buses

Interface	Width	Purpose
MonBus	64-bit	Internal monitoring bus (channels -> monbus_axil_group)

## Area Estimates

### By Performance Mode

Configuration	Total LUTs	SRAM	Use Case
Low (Tutorial)	~9,500	64 KB	Educational, area- constrained
Medium (Typical)	~11,200	64 KB	Balanced FPGA implementatio ns
High (Performance)	~13,700	64 KB	High- throughput ASIC/FPGA

### Breakdown (Low Performance)

Component	Instances	Area/Instance	Total
Descriptor Engine	8	~300 LUTs	~2,400 LUTs
Scheduler	8	~400 LUTs	~3,200 LUTs
AXI Read Engine (Low)	1	~250 LUTs	~250 LUTs
AXI Write Engine (Low)	1	~250 LUTs	~250 LUTs
SRAM Controller	1	~1,600 LUTs	~1,600 LUTs
Simple SRAM (internal)	1-8	1024x64B total	64 KB
Channel Arbiter	3	~150 LUTs	~450 LUTs
APB Config	1	~350 LUTs	~350 LUTs

Component	Instances	Area/Instance	Total
MonBus AXIL Group	1	~1,000 LUTs	~1,000 LUTs
<b>Total</b>	-	-	<b>~9,500 LUTs + 64KB</b>

## Related Documentation

- [PRD.md](#) - Product requirements and overview
  - [ARCHITECTURAL\\_NOTES.md](#) - Critical design decisions
  - [CLAUDE.md](#) - AI development guide
  - [Register Generation](#) - PeakRDL workflow
- 

## Specification Conventions

### Signal Naming

- **Clock:** aclk, pclk
- **Reset:** aresetn, presetn (active-low)
- **Valid/Ready:** Standard AXI/custom handshake
- **Registers:** r\_ prefix (e.g., r\_state, r\_counter)
- **Wires:** w\_ prefix (e.g., w\_next\_state, w\_grant)

### Parameter Naming

- **Uppercase:** NUM\_CHANNELS, DATA\_WIDTH, ADDR\_WIDTH
- **String parameters:** PERFORMANCE (“LOW”, “MEDIUM”, “HIGH”)

### State Machine Naming

```
typedef enum logic [3:0] {
    IDLE    = 4'h0,
    ACTIVE  = 4'h1,
    // ...
} state_t;

state_t r_state, w_next_state; // Current and next state
```

---

Last Updated: 2025-10-17 Maintained By: STREAM Architecture Team

# STREAM Architecture Overview

**Component:** STREAM (Scatter-gather Transfer Rapid Engine for AXI Memory)

**Version:** 0.90 **Status:** Pre-release

---

## Introduction

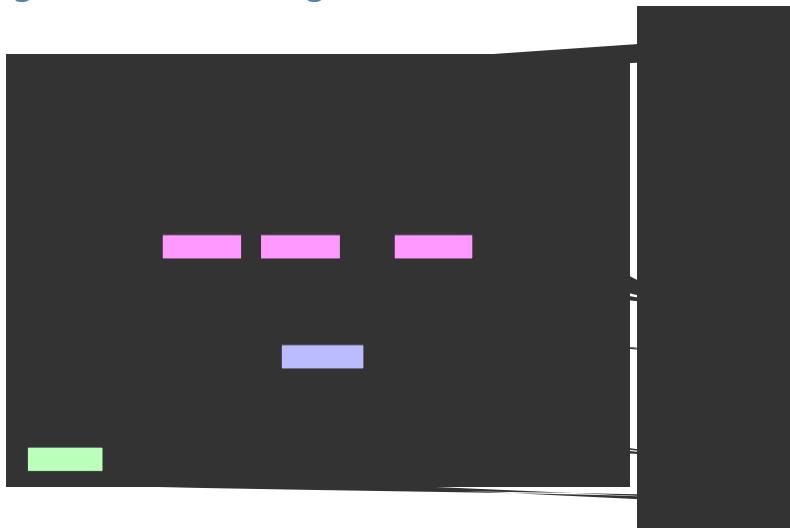
STREAM is a high-performance, multi-channel descriptor-based DMA engine designed for memory-to-memory scatter-gather transfers. It provides an educational yet production-capable architecture demonstrating key DMA concepts while maintaining professional design practices.

## Design Philosophy

1. **Tutorial Focus** - Intentional simplifications for learning
  2. **Production Quality** - Industry-standard interfaces and verification
  3. **Scalability** - 8 independent channels with shared resource arbitration
  4. **Modularity** - Clear separation of concerns across functional blocks
- 

## System Architecture

### High-Level Block Diagram



*Diagram*

APB\_CFG APB\_CFG → |enable, config| DESC\_ENG APB\_CFG → |enable, config|  
SCHED

```

DESC_ENG --> |desc_ar_valid| DESC_ARB
DESC_ARB --> |m_axi_desc_ar| DESC_MEM
DESC_MEM --> |m_axi_desc_r| DESC_ENG
DESC_ENG --> |descriptor| SCHED

SCHED --> |datard_req| RD_ARB
RD_ARB --> RD_ENG
RD_ENG --> |m_axi_rd_ar| SRC_MEM
SRC_MEM --> |m_axi_rd_r| RD_ENG
RD_ENG --> |rd_data| SRAM

SCHED --> |datawr_req| WR_ARB
WR_ARB --> WR_ENG
SRAM --> |wr_data| WR_ENG
WR_ENG --> |m_axi_wr_aw/w| DST_MEM
DST_MEM --> |m_axi_wr_b| WR_ENG

RD_ENG --> |done_strobe| SCHED
WR_ENG --> |done_strobe| SCHED

DESC_ENG --> |mon_events| MON_AGG
SCHED --> |mon_events| MON_AGG
RD_ENG --> |mon_events| MON_AGG
WR_ENG --> |mon_events| MON_AGG
MON_AGG --> MON_IF
MON_AGG --> IRQ_OUT

style DESC_ARB fill:#f9f,stroke:#333,stroke-width:3px
style RD_ARB fill:#f9f,stroke:#333,stroke-width:3px
style WR_ARB fill:#f9f,stroke:#333,stroke-width:3px
style SRAM fill:#bbf,stroke:#333,stroke-width:3px
style MON_AGG fill:#fbf,stroke:#333,stroke-width:2px

```

-->

---

## ## Key Architectural Concepts

### ### 1. Multi-Channel Design

- \*\*8 Independent Channels:\*\*
  - Each channel operates autonomously
  - Separate descriptor chains per channel
  - Independent FSM state per channel
  - Concurrent transfers across all channels

### \*\*Resource Sharing:\*\*

- Descriptor AXI master (shared via round-robin arbiter)
- Data read AXI master (shared via priority arbiter)
- Data write AXI master (shared via priority arbiter)
- SRAM buffer (per-channel FIFOs, no arbitration)

### ### 2. Descriptor-Based Operation

**Descriptor Format (256-bit):**

[255:192] Reserved [191:160] Next Descriptor Pointer [159:128] Length (in beats)  
 [127:64] Destination Address [63:0] Source Address

**Descriptor Chain:**

- Single APB write kicks off chain
- Automatic chaining via next\_descriptor\_ptr
- Explicit termination (ptr = 0 or last flag)

### ### 3. Concurrent Read/Write

**CRITICAL Design Pattern:**

STREAM uses concurrent read and write operations to handle transfers larger than the SRAM buffer:

Example: 100MB transfer with 64KB SRAM buffer

Sequential (WRONG): Read 100MB → DEADLOCK at 64KB (SRAM full)

Concurrent (CORRECT): Read fills SRAM → SRAM full → Read pauses Write drains SRAM → SRAM space freed → Read resumes Both continue until 100MB complete

**Implementation:**

- Scheduler runs read and write FSMs concurrently in XFER\_DATA state
- Independent beat counters for read vs write
- Transfer completes when BOTH counters reach zero

### ### 4. Space-Aware Flow Control

**Allocation Controller (Read Path):**

- Reserves SRAM space BEFORE issuing AXI AR
- Prevents overflow due to AR/R latency gap
- 2x space margin (accounts for in-flight allocations)

**Drain Controller (Write Path):**

- Reserves SRAM data BEFORE issuing AXI AW

- Prevents underflow due to AW/W latency gap
- Includes latency bridge occupancy in count

### ### 5. Priority-Based Arbitration

\*\*Descriptor Fetch:\*\*

- Round-robin arbitration (fair access)
- All channels equal priority

\*\*Data Read/Write:\*\*

- Priority-based arbitration
- Priority from descriptor
- Round-robin within same priority tier
- Timeout prevention for low-priority channels

---

### ## Data Flow Example

#### ### Single Channel Transfer

```
![Diagram](/mnt/data/github/rtldesignsherpa/projects/components/stream/docs/stream_spec/assets/mermaid/01_architecture_sequence.svg)
```

<!--  
Original Mermaid diagram (for editing):

```
```mermaid
sequenceDiagram
    participant SW as Software
    participant APB as APB Config
    participant DESC as Descriptor Engine
    participant SCHED as Scheduler
    participant RD as Read Engine
    participant SRAM as SRAM Controller
    participant WR as Write Engine
    participant MEM as Memory

    SW->>APB: Write CH0_CTRL = 0x1000_0000
    APB->>DESC: Kickoff descriptor fetch
    DESC->>MEM: AR: fetch descriptor @ 0x1000_0000
    MEM-->>DESC: R: descriptor data (256-bit)
    DESC->>SCHED: descriptor_packet

    SCHED->>RD: datard_valid + src_addr
    RD->>SRAM: Check space_free >= 2x burst
    SRAM-->>RD: Space OK
```

```

RD->>SRAM: Allocate 16 beats
RD->>MEM: AR: read 16 beats @ src_addr
MEM-->RD: R: 16 beats of data
RD->>SRAM: Write data (16 beats)
RD->>SCHED: done_strobe (16 beats)

par Concurrent Read/Write
    RD->>SRAM: Continue reading...
and
    SCHED->>WR: datawr_valid + dst_addr
    WR->>SRAM: Check data_avail >= burst
    SRAM-->WR: Data OK
    WR->>SRAM: Reserve 16 beats
    WR->>MEM: AW: write 16 beats @ dst_addr
    SRAM->>WR: W: 16 beats of data
    MEM-->WR: B: write response
    WR->>SCHED: done_strobe (16 beats)
end

SCHED->>SCHED: Both counters reach 0
SCHED->>DESC: Check next_descriptor_ptr
alt More descriptors
    DESC->>MEM: Fetch next descriptor
else Chain complete
    SCHED->>APB: Transfer complete IRQ
end

```

→

---

## Component Hierarchy

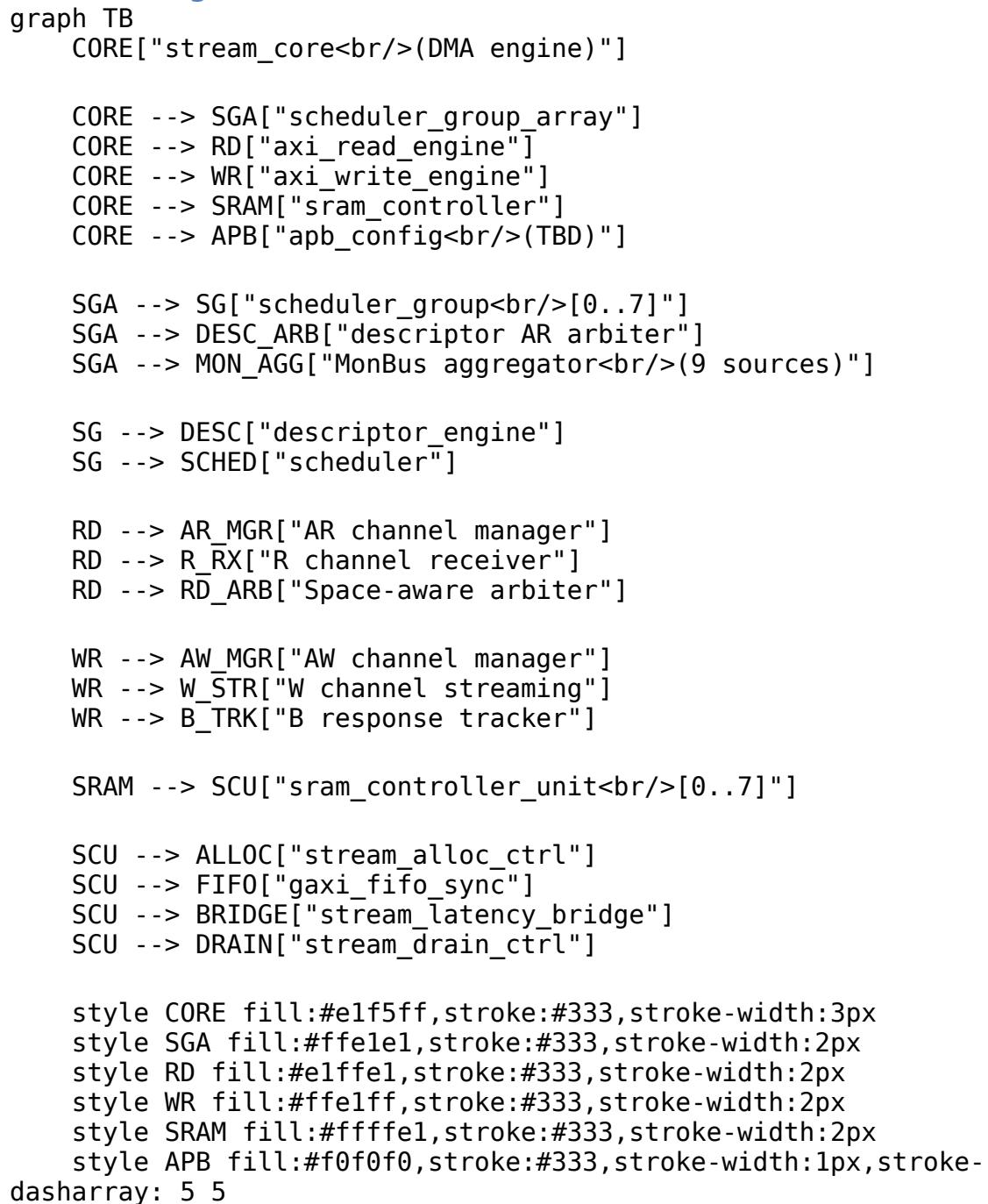
### Top-Level Integration (stream\_top\_ch8.sv)

```

stream_top_ch8 (Top-Level Wrapper)
└── APB Interface Path
    ├── apb_slave_cdc          (CDC_ENABLE=1: APB clock domain crossing)
    ├── cmdrsp_router           (Address-based routing)
    │   ├── apbtodescr          (0x000-0x03F: Channel kick-off routing)
    │   ├── perf_profiler        (0x040-0x0FF: Performance profiling)
    │   └── peakrdl_to_cmdrsp   (0x100-0x3FF: APB → CMD/RSP conversion)
    |       └── stream_regs     (PeakRDL-generated register file)
    ├── stream_config_block     (Register → config signal mapping)
    ├── stream_core             (Main DMA engine - see below)
    └── monbus_axil_group       (USE_AXI_MONITORS=1: MonBus → AXI-Lite)
        ├── Error FIFO (s_axil_err_* slave read)
        └── Master Writer (m_axil_mon_* master write)

```

## Core DMA Engine (stream\_core.sv)



## Interface Summary

### External Interfaces (stream\_top\_ch8.sv)

| Interface          | Type   | Width   | Purpose                                            |
|--------------------|--------|---------|----------------------------------------------------|
| APB                | Slave  | 32-bit  | Configuration, control, status (with optional CDC) |
| Descriptor AXI     | Master | 256-bit | Fetch descriptors from memory                      |
| Data Read AXI      | Master | 512-bit | Read source data                                   |
| Data Write AXI     | Master | 512-bit | Write destination data                             |
| Error FIFO AXIL    | Slave  | 32-bit  | Read monitor error/interrupt FIFO                  |
| Monitor Write AXIL | Master | 32-bit  | Write monitor data to memory                       |
| IRQ                | Output | 1-bit   | Interrupt (error FIFO not empty)                   |

### Internal MonBus Interface (stream\_core.sv)

| Interface | Type   | Width  | Purpose                                       |
|-----------|--------|--------|-----------------------------------------------|
| MonBus    | Output | 64-bit | Debug/trace event stream to monbus_axil_group |

### Configuration Registers (APB)

#### APB Address Map:

0x000-0x03F: Channel kick-off registers (apbtodescr routing)

0x040-0x0FF: Performance profiler interface

0x100-0x3FF: PeakRDL configuration registers

**Key Register Groups:** - 0x100: Global Control (enable, reset, version) - 0x120: Per-Channel Control (enable, reset) - 0x140: Per-Channel Status (idle, state, completion) - 0x180: Monitor FIFO Status - 0x200: Scheduler Configuration - 0x220: Descriptor Engine Configuration - 0x240-0x29F: AXI Monitor Configuration (DAXMON, RDMON, WRMON) - 0x2A0: AXI Transfer Configuration - 0x2B0: Performance Profiler Configuration

**See:** [Register Map](#) for complete documentation.

---

## Resource Utilization

### Area Breakdown (Typical 512-bit Data Width)

| Component         | Quantity | Logic (LUTs)     | Memory                                |
|-------------------|----------|------------------|---------------------------------------|
| Descriptor Engine | 8        | ~300 each        | Minimal                               |
| Scheduler         | 8        | ~400 each        | Minimal                               |
| AXI Read Engine   | 1        | ~800             | Minimal                               |
| AXI Write Engine  | 1        | ~800             | Minimal                               |
| SRAM Controller   | 1        | ~600             | $8 \times 64\text{KB} = 512\text{KB}$ |
| Arbiters          | 3        | ~150 each        | Minimal                               |
| APB Config        | 1        | ~400             | Register file                         |
| MonBus Aggregator | 1        | ~200             | Small FIFO                            |
| <b>Total</b>      | -        | <b>~10K LUTs</b> | <b>~512KB RAM</b>                     |

**Notes:** - SRAM depth configurable (typical: 512 entries  $\times$  512 bits  $\times$  8 channels = 512KB) - Logic utilization scales with NUM\_CHANNELS - Memory utilization scales with SRAM\_DEPTH

---

## Performance Characteristics

### Throughput

**Theoretical Maximum (512-bit @ 250MHz):** - Single channel: 16 GB/s (512 bits  $\times$  250 MHz) - 8 channels concurrent: Limited by memory bandwidth, not DMA

**Practical Performance:** - Dependent on: - Memory controller latency - Burst sizes (larger = better efficiency) - Transfer alignment - Channel contention

**Typical Real-World:** - 8-12 GB/s sustained (single channel) - 50-75% of theoretical maximum

## Latency

**Descriptor Fetch Latency:** - AR issue to R data: ~10-50 cycles (memory dependent) - Descriptor parsing: 1 cycle - **Total:** ~11-51 cycles per descriptor

**Transfer Initiation:** - APB write to first AR: ~5-10 cycles - AR to first R data: ~10-50 cycles - **Total:** ~15-60 cycles from kickoff to first data

**Transfer Completion:** - Last W beat to B response: ~5-20 cycles - B response to IRQ: 1 cycle - **Total:** ~6-21 cycles from last data to interrupt

---

## Design Decisions

### Why These Simplifications?

1. **Aligned Addresses Only:**
  - Eliminates complex alignment fixup logic
  - Clear data path with no byte shifting
  - Focus: Core DMA operation, not edge cases
2. **Length in Beats:**
  - Direct mapping to AXI burst length
  - No unit conversion overhead
  - Focus: AXI protocol understanding
3. **No Credit Management:**
  - Simpler resource arbitration
  - Transaction limits via configuration
  - Focus: Arbitration basics, not complex flow control
4. **No Circular Buffers:**
  - Explicit chain termination
  - Clear end-of-transfer detection
  - Focus: Descriptor chaining, not circular logic

### Production Enhancements (Future)

If extending STREAM for production use, consider:  
- [ ] Alignment fixup (byte-level granularity)  
- [ ] Length in bytes/chunks  
- [ ] Credit-based flow control  
- [ ] Circular buffer support  
- [ ] Advanced error recovery  
- [ ] Power management  
- [ ] Multiple SRAM segments

---

## Testing Strategy

### Verification Layers

- 1. Unit Tests (FUB Level):** - Descriptor engine: Fetch, parse, chain - Scheduler: FSM states, concurrent read/write - AXI engines: AR/R and AW/W/B channels - SRAM controller: Allocation, drain, FIFO - Individual tests per module
- 2. Integration Tests (Multi-Block):** - Scheduler + engines: Data flow - Descriptor + scheduler: Descriptor chaining - Full path: APB → Descriptor → Transfer → IRQ
- 3. System Tests (Full Core):** - Single channel end-to-end - Multi-channel concurrent - Error injection and recovery - Performance validation - Long-duration stress tests

### Coverage Targets

- **Code Coverage:** >95%
  - **Functional Coverage:** >90%
  - **Corner Cases:** 100% tested
  - **Error Paths:** 100% tested
- 

## Related Documentation

**Chapter 2 - Block Specifications:** - [Scheduler](#) - Core FSM controller - [Descriptor Engine](#) - Descriptor fetch/parse - [AXI Read Engine](#) - Source data read - [AXI Write Engine](#) - Destination data write - [SRAM Controller](#) - Buffering and flow control - [Scheduler Group](#) - Per-channel wrapper - [Scheduler Group Array](#) - 8-channel array

**Other Resources:** - [STREAM PRD](#) - Product requirements - [Test Plan](#) - Verification strategy

---

**Last Updated:** 2025-12-01 **Document Version:** 0.91

## STREAM Top-Level Port List

**Module:** stream\_core.sv **Location:** projects/components/stream/rtl/macro/  
**Last Updated:** 2025-12-01

---

## Overview

This document provides a complete reference for all top-level ports of the STREAM Core module, organized by interface type. All port names, directions, widths, and descriptions are extracted directly from the RTL implementation.

**Note:** This documents `stream_core.sv` which is instantiated inside `stream_top_ch8.sv`. The top-level wrapper adds:  
- APB slave interface (replaces `apb_valid/ready/addr` with standard APB protocol)  
- AXI-Lite interfaces for monitor bus (`s_axil_err_, m_axil_mon_`)  
- Optional APB CDC crossing (CDC\_ENABLE parameter)

For top-level integration, see the [Architecture Overview](#).

**Quick Navigation:** - [Clock and Reset](#) - [APB Programming Interface](#) - [Configuration Interface](#) - [Status Interface](#) - [Performance Profiler Interface](#) - [AXI4 Master](#) - [Descriptor Fetch](#) - [AXI4 Master](#) - [Data Read](#) - [AXI4 Master](#) - [Data Write](#) - [Status/Debug Outputs](#) - [Unified Monitor Bus](#)

---

## Clock and Reset

| Signal             | Direction | Width | Description                              |
|--------------------|-----------|-------|------------------------------------------|
| <code>clk</code>   | input     | 1     | System clock<br>(100-500 MHz<br>typical) |
| <code>rst_n</code> | input     | 1     | Active-low<br>asynchronous<br>reset      |

**Notes:** - All STREAM logic operates in the `clk` domain - Reset is asynchronous assert, synchronous deassert - All registers use reset macros from `reset_defs.svh`

---

## APB Programming Interface

Per-channel descriptor kick-off interface. Each channel has independent valid/ready/address signals for descriptor chain start.

| Signal                     | Direction | Width    | Description        |
|----------------------------|-----------|----------|--------------------|
| <code>apb_valid[ch]</code> | input     | NUM_CHAN | Channel descriptor |

| Signal        | Direction | Width                                 | Description                                        |
|---------------|-----------|---------------------------------------|----------------------------------------------------|
| ]             |           | NELS                                  | address valid                                      |
| apb_ready[ch] | output    | NUM_CHAN                              | Channel ready to accept                            |
| ]             |           | NELS                                  | descriptor address                                 |
| apb_addr[ch]  | input     | NUM_CHAN<br>NELS ×<br>ADDR_WIDTH<br>H | Descriptor address per<br>channel (64-bit default) |

### Usage Pattern:

```
// Kick off channel 0 with descriptor at 0x1000_0000
apb_valid[0] = 1'b1;
apb_addr[0] = 64'h0000_0000_1000_0000;
// Wait for handshake
wait (apb_ready[0]);
apb_valid[0] = 1'b0;
```

**Notes:** - Standard valid/ready handshake protocol - Descriptor address must be aligned to descriptor size (32 bytes) - Channel starts operation immediately after handshake - Default: NUM\_CHANNELS = 8, ADDR\_WIDTH = 64

---

## Configuration Interface

### Per-Channel Configuration

| Signal                 | Direction | Width            | Description                               |
|------------------------|-----------|------------------|-------------------------------------------|
| cfg_channel_enable[ch] | input     | NUM_CHAN<br>NELS | Enable channel<br>(0=disabled, 1=enabled) |
| cfg_channel_reset[ch]  | input     | NUM_CHAN<br>NELS | Soft reset channel (FSM<br>→ IDLE state)  |

**Notes:** - Channel must be enabled before accepting descriptors - Soft reset clears channel FSM but preserves config

### Global Scheduler Configuration

| Signal                   | Direction | Width | Description                         |
|--------------------------|-----------|-------|-------------------------------------|
| cfg_sched_enable         | input     | 1     | Global scheduler enable             |
| cfg_sched_timeout_cycles | input     | 16    | Timeout threshold (clock<br>cycles) |

| Signal                    | Direction | Width | Description                        |
|---------------------------|-----------|-------|------------------------------------|
| cfg_sched_ti_meout_enable | input     | 1     | Enable timeout detection           |
| cfg_sched_er_r_enable     | input     | 1     | Enable error event reporting       |
| cfg_sched_co_mpl_enable   | input     | 1     | Enable completion event reporting  |
| cfg_sched_pe_rf_enable    | input     | 1     | Enable performance event reporting |

**Notes:** - cfg\_sched\_enable is master enable for all schedulers - Timeout measured from descriptor fetch to completion - Event enables control MonBus traffic

### Descriptor Engine Configuration

| Signal                  | Direction | Width        | Description                         |
|-------------------------|-----------|--------------|-------------------------------------|
| cfg_desceng_enable      | input     | 1            | Enable descriptor engine            |
| cfg_desceng_prefetch    | input     | 1            | Enable descriptor prefetch          |
| cfg_desceng_fifo_thresh | input     | 4            | FIFO threshold for prefetch trigger |
| cfg_desceng_addr0_base  | input     | ADDR_WIDTH_H | Address range 0 base (protection)   |
| cfg_desceng_addr0_limit | input     | ADDR_WIDTH_H | Address range 0 limit (protection)  |
| cfg_desceng_addr1_base  | input     | ADDR_WIDTH_H | Address range 1 base (protection)   |
| cfg_desceng_addr1_limit | input     | ADDR_WIDTH_H | Address range 1 limit (protection)  |

**Notes:** - Descriptor engine shared across all channels - Prefetch improves latency for chained descriptors - Address ranges provide optional descriptor memory protection

### AXI Monitor Configuration

Three identical sets of monitor config signals for descriptor, read, and write AXI masters:

| Signal Prefix   | Applies To     | Description                 |
|-----------------|----------------|-----------------------------|
| cfg_desc_mon_*  | Descriptor AXI | Descriptor fetch monitoring |
| cfg_rdeng_mon_* | Read AXI       | Data read monitoring        |
| cfg_wreng_mon_* | Write AXI      | Data write monitoring       |

Each monitor has the following configuration signals:

| Signal Suffix   | Width | Description                         |
|-----------------|-------|-------------------------------------|
| _enable         | 1     | Enable monitor                      |
| _err_enable     | 1     | Enable error packet reporting       |
| _perf_enable    | 1     | Enable performance packet reporting |
| _timeout_enable | 1     | Enable timeout detection            |
| _timeout_cycles | 32    | Timeout threshold (cycles)          |
| _latency_thresh | 32    | Latency threshold for events        |
| _pkt_mask       | 16    | Packet type mask (filter events)    |
| _err_select     | 4     | Error type selector                 |
| _err_mask       | 8     | Error event mask                    |
| _timeout_mask   | 8     | Timeout event mask                  |
| _compl_mask     | 8     | Completion event mask               |
| _thresh_mask    | 8     | Threshold event mask                |
| _perf_mask      | 8     | Performance event mask              |
| _addr_mask      | 8     | Address event mask                  |
| _debug_mask     | 8     | Debug event mask                    |

### Example Signals:

```
cfg_desc_mon_enable      // Descriptor monitor enable
cfg_desc_mon_timeout_cycles // Descriptor timeout threshold
cfg_rdeng_mon_err_enable    // Read engine error reporting enable
cfg_wreng_mon_perf_enable   // Write engine perf reporting enable
```

**Notes:** - Monitors generate MonBus packets for events - Masks control which event types are reported - Timeout measured from AR/AW valid to last R/B response

### AXI Transfer Configuration

| Signal                | Direction | Width | Description                     |
|-----------------------|-----------|-------|---------------------------------|
| cfg_axi_rd_xfer_beats | input     | 8     | Read burst size (beats, 1-256)  |
| cfg_axi_wr_xfer_beats | input     | 8     | Write burst size (beats, 1-256) |

**Notes:** - Configures AXI burst length for read/write engines - Larger bursts improve bandwidth but increase latency - Typical: 16 beats for DDR4, 8 beats for low-latency SRAM

### Performance Profiler Configuration

| Signal          | Direction | Width | Description                            |
|-----------------|-----------|-------|----------------------------------------|
| cfg_perf_enable | input     | 1     | Enable profiler                        |
| cfg_perf_mode   | input     | 1     | Profiler mode (0=timestamp, 1=elapsed) |
| cfg_perf_clear  | input     | 1     | Clear profiler counters and FIFO       |

**Notes:** - Timestamp mode: Records start/end timestamps (software calculates elapsed) - Elapsed mode: Hardware calculates elapsed time directly - Clear is a single-cycle pulse

---

## Status Interface

### Per-Channel Status

| Signal                     | Direction | Width                | Description                                         |
|----------------------------|-----------|----------------------|-----------------------------------------------------|
| descriptor_engine_idle[ch] | output    | NUM_CHAN<br>NELS     | Descriptor engine idle<br>(no pending fetch)        |
| scheduler_idleness[ch]     | output    | NUM_CHAN<br>NELS     | Scheduler in IDLE state<br>(waiting for descriptor) |
| scheduler_state[ch]        | output    | NUM_CHAN<br>NELS × 7 | Scheduler FSM state<br>(ONE-HOT encoding)           |
| sched_error[ch]            | output    | NUM_CHAN<br>NELS     | Scheduler error flag<br>(sticky, cleared by reset)  |
| axi_rd_all_complete[ch]    | output    | NUM_CHAN<br>NELS     | All read transactions<br>complete for channel       |
| axi_wr_all_complete[ch]    | output    | NUM_CHAN<br>NELS     | All write transactions<br>complete for channel      |

### Scheduler State Encoding (ONE-HOT):

```
7'b00000001 // CH_IDLE - Waiting for descriptor
7'b00000010 // CH_FETCH_DESC - Fetching descriptor
7'b00001000 // CH_XFER_DATA - Transfer in progress
7'b00010000 // CH_COMPLETE - Transfer complete
7'b00100000 // CH_NEXT_DESC - Chaining to next descriptor
7'b01000000 // CH_ERROR - Error occurred
7'b10000000 // (Reserved)
```

### Transfer Complete Condition:

```
channel_complete = scheduler_idle[ch] &&
                  axi_rd_all_complete[ch] &&
                  axi_wr_all_complete[ch];
```

---

## Performance Profiler Interface

| Signal          | Direction | Width | Description              |
|-----------------|-----------|-------|--------------------------|
| perf_fifo_empty | output    | 1     | Profiler FIFO empty flag |
| perf_fifo_full  | output    | 1     | Profiler FIFO full flag  |
| perf_fifo_count | output    | 16    | Profiler FIFO occupancy  |

| Signal               | Direction | Width | Description                                 |
|----------------------|-----------|-------|---------------------------------------------|
| unt                  |           |       | (0-256)                                     |
| perf_fifo_rd         | input     | 1     | Read profiler entry (pop FIFO)              |
| perf_fifo_da_ta_low  | output    | 32    | Profiler data [31:0] (timestamp or elapsed) |
| perf_fifo_da_ta_high | output    | 32    | Profiler data [63:32] (metadata)            |

### FIFO Entry Format:

**Low Word [31:0]:** - Timestamp mode: Timestamp in clock cycles - Elapsed mode: Elapsed time in clock cycles

**High Word [31:0]:** - Bits [31:4]: Reserved (0) - Bit [3]: Event type (0=start, 1=end) - Bits [2:0]: Channel ID (0-7)

### Read Sequence:

```
// Check FIFO not empty
if (!perf_fifo_empty) begin
    // Read 36-bit entry (two registers)
    perf_fifo_rd = 1'b1; // Pulse to pop FIFO
    @(posedge clk);
    perf_fifo_rd = 1'b0;

    // Sample data on next cycle
    timestamp = perf_fifo_data_low;
    metadata = perf_fifo_data_high;
    channel_id = metadata[2:0];
    event_type = metadata[3];
end
```

---

## AXI4 Master - Descriptor Fetch (256-bit)

Fixed 256-bit width AXI4 master for descriptor fetch from memory.

### AR Channel (Read Address)

| Signal           | Direction | Width        | Description    |
|------------------|-----------|--------------|----------------|
| m_axi_desc_ar_id | output    | AXI_ID_WIDTH | Transaction ID |

| Signal        | Direction | Width      | Description                                     |
|---------------|-----------|------------|-------------------------------------------------|
| m_axi_desc_ar | output    | ADDR_WIDTH | Read address                                    |
| m_axi_desc_ar | output    | 8          | Burst length - 1<br>(AXI encoding)              |
| m_axi_desc_ar | output    | 3          | Burst size =<br>$\log_2(\text{bytes per beat})$ |
| m_axi_desc_ar | output    | 2          | Burst type<br>(01=INCR)                         |
| m_axi_desc_ar | output    | 1          | Lock type<br>(0=normal)                         |
| m_axi_desc_ar | output    | 4          | Cache<br>attributes                             |
| m_axi_desc_ar | output    | 3          | Protection<br>attributes                        |
| m_axi_desc_ar | output    | 4          | QoS value                                       |
| m_axi_desc_ar | output    | 4          | Region<br>identifier                            |
| m_axi_desc_ar | output    | CHAN_WIDTH | User signal<br>(channel ID)                     |
| m_axi_desc_ar | output    | 1          | Address valid                                   |
| m_axi_desc_ar | input     | 1          | Address ready                                   |

### R Channel (Read Data)

| Signal       | Direction | Width         | Description                                               |
|--------------|-----------|---------------|-----------------------------------------------------------|
| m_axi_desc_r | input     | AXI_ID_WID TH | Transaction ID                                            |
| m_axi_desc_r | input     | 256           | Read data (FIXED 256-bit descriptor)                      |
| m_axi_desc_r | input     | 2             | Response (00=OKAY,<br>01=EXOKAY, 10=SLVERR,<br>11=DECERR) |
| m_axi_desc_r | input     | 1             | Last beat of burst                                        |

| Signal             | Direction | Width      | Description              |
|--------------------|-----------|------------|--------------------------|
| m_axi_desc_r_user  | input     | CHAN_WIDTH | User signal (channel ID) |
| m_axi_desc_r_valid | input     | 1          | Read data valid          |
| m_axi_desc_r_ready | output    | 1          | Read data ready          |

**Notes:** - Data width is FIXED at 256 bits (descriptor size) - Typical burst: 1 beat (arlen=0) for single descriptor fetch - arsize = 3'b101 (32 bytes =  $2^5$ ) - ID encodes channel number for response routing

---

## AXI4 Master - Data Read (Parameterizable Width)

Parameterizable width AXI4 master for reading source data from memory.  
Default 512-bit.

### AR Channel (Read Address)

| Signal            | Direction | Width        | Description                       |
|-------------------|-----------|--------------|-----------------------------------|
| m_axi_rd_arid     | output    | AXI_ID_WIDTH | Transaction ID                    |
| m_axi_rd_arad_dr  | output    | ADDR_WIDTH   | Read address                      |
| m_axi_rd_arle_n   | output    | 8            | Burst length - 1                  |
| m_axi_rd_arsize   | output    | 3            | Burst size = log2(bytes per beat) |
| m_axi_rd_arburst  | output    | 2            | Burst type (01=INCR)              |
| m_axi_rd_arlock   | output    | 1            | Lock type (0=normal)              |
| m_axi_rd_arcache  | output    | 4            | Cache attributes                  |
| m_axi_rd_arprot   | output    | 3            | Protection attributes             |
| m_axi_rd_arqos    | output    | 4            | QoS value                         |
| m_axi_rd_arregion | output    | 4            | Region                            |

| Signal           | Direction | Width      | Description              |
|------------------|-----------|------------|--------------------------|
| gion             |           |            | identifier               |
| m_axi_rd_aruser  | output    | CHAN_WIDTH | User signal (channel ID) |
| m_axi_rd_arvalid | output    | 1          | Address valid            |
| m_axi_rd_arready | input     | 1          | Address ready            |

### R Channel (Read Data)

| Signal          | Direction | Width        | Description                 |
|-----------------|-----------|--------------|-----------------------------|
| m_axi_rd_rid    | input     | AXI_ID_WIDTH | Transaction ID              |
| m_axi_rd_rdata  | input     | DATA_WIDTH   | Read data (default 512-bit) |
| m_axi_rd_rresp  | input     | 2            | Response                    |
| m_axi_rd_rlast  | input     | 1            | Last beat of burst          |
| m_axi_rd_ruse   | input     | CHAN_WIDTH   | User signal (channel ID)    |
| m_axi_rd_rvalid | input     | 1            | Read data valid             |
| m_axi_rd_rready | output    | 1            | Read data ready             |

**Notes:** - Data width configurable via DATA\_WIDTH parameter (default 512) - Burst length configured via cfg\_axi\_rd\_xfer\_beats (default 16) - arsize = log2(DATA\_WIDTH/8) automatically calculated - AXI skid buffers on external interface for timing closure

---

### AXI4 Master - Data Write (Parameterizable Width)

Parameterizable width AXI4 master for writing destination data to memory. Default 512-bit.

### AW Channel (Write Address)

| Signal                        | Direction | Width        | Description                       |
|-------------------------------|-----------|--------------|-----------------------------------|
| m_axi_wr_awid                 | output    | AXI_ID_WIDTH | Transaction ID                    |
| m_axi_wr_awad <sub>dr</sub>   | output    | ADDR_WIDTH   | Write address                     |
| m_axi_wr_awle <sub>n</sub>    | output    | 8            | Burst length - 1                  |
| m_axi_wr_awsi <sub>ze</sub>   | output    | 3            | Burst size = log2(bytes per beat) |
| m_axi_wr_awbu <sub>rst</sub>  | output    | 2            | Burst type (01=INCR)              |
| m_axi_wr_awlo <sub>ck</sub>   | output    | 1            | Lock type (0=normal)              |
| m_axi_wr_awca <sub>che</sub>  | output    | 4            | Cache attributes                  |
| m_axi_wr_awpr <sub>ot</sub>   | output    | 3            | Protection attributes             |
| m_axi_wr_awqo <sub>s</sub>    | output    | 4            | QoS value                         |
| m_axi_wr_awre <sub>gion</sub> | output    | 4            | Region identifier                 |
| m_axi_wr_awus <sub>er</sub>   | output    | CHAN_WIDTH   | User signal (channel ID)          |
| m_axi_wr_awva <sub>lid</sub>  | output    | 1            | Address valid                     |
| m_axi_wr_awre <sub>ady</sub>  | input     | 1            | Address ready                     |

### W Channel (Write Data)

| Signal                     | Direction | Width                    | Description                          |
|----------------------------|-----------|--------------------------|--------------------------------------|
| m_axi_wr_wda <sub>ta</sub> | output    | DATA_WIDT <sub>H</sub>   | Write data (default 512-bit)         |
| m_axi_wr_wst <sub>rb</sub> | output    | DATA_WIDT <sub>H/8</sub> | Write strobes (byte enables, all 1s) |
| m_axi_wr_wla <sub>st</sub> | output    | 1                        | Last beat of burst                   |
| m_axi_wr_wus               | output    | CHAN_WIDTH               | User signal (channel ID)             |

| Signal           | Direction | Width | Description      |
|------------------|-----------|-------|------------------|
| er               |           | H     |                  |
| m_axi_wr_wva_lid | output    | 1     | Write data valid |
| m_axi_wr_wre_ady | input     | 1     | Write data ready |

### B Channel (Write Response)

| Signal           | Direction | Width         | Description                                         |
|------------------|-----------|---------------|-----------------------------------------------------|
| m_axi_wr_bid     | input     | AXI_ID_WID TH | Transaction ID                                      |
| m_axi_wr_bre_sp  | input     | 2             | Response (00=OKAY, 01=EXOKAY, 10=SLVERR, 11=DECERR) |
| m_axi_wr_bus_er  | input     | CHAN_WIDT H   | User signal (channel ID)                            |
| m_axi_wr_bva_lid | input     | 1             | Response valid                                      |
| m_axi_wr_bre_ady | output    | 1             | Response ready                                      |

**Notes:** - Data width configurable via DATA\_WIDTH parameter (default 512) - Burst length configured via cfg\_axi\_wr\_xfer\_beats (default 16) - awsize = log2(DATA\_WIDTH/8) automatically calculated - wstrb typically all 1s (full data width writes) - AXI skid buffers on external interface for timing closure

---

### Status/Debug Outputs

#### Descriptor AXI Monitor Status

| Signal                       | Direction | Width | Description                         |
|------------------------------|-----------|-------|-------------------------------------|
| cfg_sts_desc_mon_busy        | output    | 1     | Monitor busy processing transaction |
| cfg_sts_desc_mon_active_txns | output    | 8     | Active transaction count (0-255)    |
| cfg_sts_desc_mon_error_count | output    | 16    | Cumulative error count              |
| cfg_sts_desc                 | output    | 32    | Total transaction count             |

| Signal         | Direction | Width | Description                   |
|----------------|-----------|-------|-------------------------------|
| _mon_txn_count | output    | 1     | ID conflict detected (sticky) |
| cfg_sts_desc   | output    | 1     | ID conflict detected (sticky) |

#### Read Engine AXI Monitor Status

| Signal                            | Direction | Width | Description                   |
|-----------------------------------|-----------|-------|-------------------------------|
| cfg_sts_rden_g_skid_busy          | output    | 1     | Skid buffer busy (not empty)  |
| cfg_sts_rden_g_mon_active_txns    | output    | 8     | Active transaction count      |
| cfg_sts_rden_g_mon_error_count    | output    | 16    | Cumulative error count        |
| cfg_sts_rden_g_mon_txn_count      | output    | 32    | Total transaction count       |
| cfg_sts_rden_g_mon_conflict_error | output    | 1     | ID conflict detected (sticky) |

#### Write Engine AXI Monitor Status

| Signal                            | Direction | Width | Description                   |
|-----------------------------------|-----------|-------|-------------------------------|
| cfg_sts_wren_g_skid_busy          | output    | 1     | Skid buffer busy (not empty)  |
| cfg_sts_wren_g_mon_active_txns    | output    | 8     | Active transaction count      |
| cfg_sts_wren_g_mon_error_count    | output    | 16    | Cumulative error count        |
| cfg_sts_wren_g_mon_txn_count      | output    | 32    | Total transaction count       |
| cfg_sts_wren_g_mon_conflict_error | output    | 1     | ID conflict detected (sticky) |

**Notes:** - Status signals for debug and performance analysis  
- Transaction counts never roll over (use for profiling)  
- Error counts increment on any AXI error response  
- ID conflicts indicate internal RTL bug (should never occur)

---

## Unified Monitor Bus Interface

Single output interface for all STREAM monitoring events.

| Signal     | Direction | Width | Description                                        |
|------------|-----------|-------|----------------------------------------------------|
| mon_valid  | output    | 1     | Monitor packet valid                               |
| mon_ready  | input     | 1     | Monitor packet ready<br>(from downstream consumer) |
| mon_packet | output    | 64    | Monitor packet data (64-bit standard format)       |

### MonBus Packet Format (64-bit):

[63:56] - Packet type (event code)  
[55:48] - Agent ID (source identifier)  
[47:40] - Unit ID (1 for STREAM)  
[39:32] - Channel ID (0-7)  
[31:0] - Event-specific data

**MonBus Sources:** - Descriptor engines: 8 sources, agent IDs 16-23 (0x10-0x17) - Schedulers: 8 sources, agent IDs 48-55 (0x30-0x37) - Descriptor AXI monitor: agent ID 8 (0x08) - Read AXI monitor: configurable agent ID - Write AXI monitor: configurable agent ID

### Downstream Integration:

```
// Connect to MonBus FIFO for buffering
gaxi_fifo_sync #(
    .DATA_WIDTH(64),
    .DEPTH(256)
) u_mon_fifo (
    .i_clk      (clk),
    .i_rst_n   (rst_n),
    .i_data    (mon_packet),
    .i_valid   (mon_valid),
    .o_ready   (mon_ready),
    // ... downstream connection
);
```

**Notes:** - Standard AMBA monitor bus protocol  
 - Always buffer with FIFO to prevent backpressure to STREAM  
 - Packet format documented in `rtl/amba/includes/monitor_pkg.sv`  
 - Event codes defined in STREAM package

---

## Port Count Summary

| Interface Type   | Input Ports                     | Output Ports                | Bidirectional |
|------------------|---------------------------------|-----------------------------|---------------|
| Clock/Reset      | 2                               | 0                           | 0             |
| APB Programming  | NUM_CHANNELS × (1 + ADDR_WIDTH) | NUM_CHANNELS × LS           | 0             |
| Configuration    | ~150                            | 0                           | 0             |
| Status           | 1 (perf_fifo_rd)                | NUM_CHANNELS × LS × 10 + 30 | 0             |
| Perf Profiler    | 1                               | 4                           | 0             |
| AXI Desc Master  | 7                               | 14                          | 0             |
| AXI Read Master  | 7                               | 14                          | 0             |
| AXI Write Master | 10                              | 19                          | 0             |
| MonBus           | 1                               | 2                           | 0             |

**Approximate Total:** ~350 ports (varies with NUM\_CHANNELS and ADDR\_WIDTH/DATA\_WIDTH)

---

## Default Parameter Values

| Parameter    | Default | Description                                        |
|--------------|---------|----------------------------------------------------|
| NUM_CHANNELS | 8       | Number of DMA channels                             |
| CHAN_WIDTH   | 3       | Channel ID width ( $\log_2(\text{NUM_CHANNELS})$ ) |
| ADDR_WIDTH   | 64      | Address bus width                                  |
| DATA_WIDTH   | 512     | Data bus width                                     |

| Parameter          | Default | Description                   |
|--------------------|---------|-------------------------------|
| AXI_ID_WIDTH       | 8       | AXI transaction ID width      |
| FIFO_DEPTH         | 512     | Per-channel FIFO depth        |
| AR_MAX_OUTSTANDING | 8       | Max concurrent read requests  |
| AW_MAX_OUTSTANDING | 8       | Max concurrent write requests |

## Related Documentation

- [Stream Core Block Spec](#) - Detailed block documentation
  - [Clocks and Reset](#) - Timing specifications
  - [Architecture](#) - System architecture overview
- 

Last Updated: 2025-12-01 Maintained By: STREAM Architecture Team

## Clocks and Reset Specification

Chapter: 01 Version: 0.90 Last Updated: 2025-11-22

---

### Overview

STREAM operates in a single clock domain with a single asynchronous active-low reset. This chapter defines clock requirements, reset behavior, and timing constraints for the STREAM subsystem.

---

### Clock Domain

[Primary Clock: aclk](#)

**Specification:** - **Name:** aclk (AXI clock) - **Type:** Synchronous, single-edge (rising edge) - **Frequency:** Configurable (typical: 100 MHz - 500 MHz) - **Duty Cycle:** 50% 5% - **Jitter:** < 100 ps peak-to-peak

**Usage:** - All STREAM internal logic - All AXI master interfaces - All AXIL interfaces  
- MonBus output - SRAM

### Secondary Clock: pclk (APB Clock)

**Specification:** - **Name:** pclk (Peripheral clock) - **Type:** Synchronous, single-edge (rising edge) - **Frequency:** Configurable (typical: 50 MHz - 200 MHz) - **Relation to aclk:** May be asynchronous

**Usage:** - APB configuration interface only

**Clock Domain Crossing (CDC):** - If pclk aclk: CDC logic required in `apb_config.sv` - If pclk = aclk: Direct connection (no CDC)

**CDC Implementation:** - Use `apb_slave_cdc` wrapper (like HPET example) - `apb_slave_cdc` implements **handshake-based CDC** using `cdc_handshake` modules - One `cdc_handshake` for command interface (APB → core) - One `cdc_handshake` for response interface (core → APB) - Full req/ack handshake protocol (NOT async FIFO) - Works across all frequency ratios (slow-to-fast, fast-to-slow, any ratio)

---

## Reset

### Primary Reset: aresetn

**Specification:** - **Name:** aresetn (AXI reset, active-low) - **Type:** Asynchronous assertion, synchronous deassertion - **Polarity:** Active-low (0 = reset, 1 = normal operation) - **Assertion:** Asynchronous (can occur at any time) - **Deassertion:** Synchronous to aclk rising edge - **Duration:** Minimum 10 aclk cycles

### Reset Behavior:

```
// Standard reset pattern for all STREAM modules
always_ff @(posedge aclk or negedge aresetn) begin
    if (!aresetn) begin
        // Asynchronous reset assertion
        r_state <= IDLE;
        r_counter <= '0;
        r_valid <= 1'b0;
        // ... all registers to known state
    end else begin
        // Synchronous operation
        r_state <= w_next_state;
        // ... normal logic
```

```
    end  
end
```

### Secondary Reset: presetn

**Specification:** - **Name:** presetn (APB reset, active-low) - **Type:** Asynchronous assertion, synchronous deassertion - **Polarity:** Active-low (0 = reset, 1 = normal operation) - **Synchronization:** May be asynchronous to aresetn

**Usage:** - APB configuration interface only - Typically tied to aresetn if pclk = aclk

---

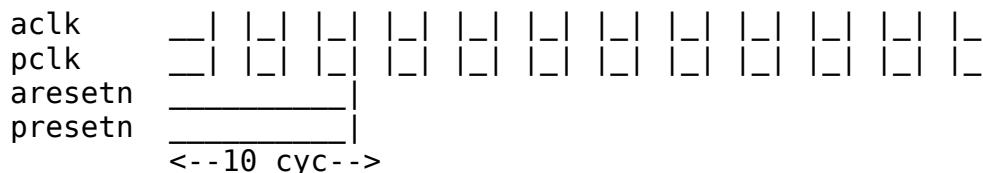
## Reset Sequencing

### Power-On Reset

#### Recommended sequence:

1. Assert aresetn (LOW)
2. Assert presetn (LOW)
3. Apply stable clocks (aclk, pclk)
4. Wait 10 aclk cycles
5. Deassert presetn (HIGH) on pclk rising edge
6. Deassert aresetn (HIGH) on aclk rising edge
7. Wait 5 aclk cycles for stabilization
8. Begin APB configuration

#### Timing diagram:



### Functional Reset

#### Software-initiated reset (per channel):

```
// Reset specific channel via APB  
write_apb(ADDR_GLOBAL_CTRL, CHANNEL_0_RESET); // Auto-clears after 1  
cycle
```

```
// Hardware response:  
// - Channel FSM returns to IDLE  
// - Channel registers cleared  
// - Outstanding transactions flushed  
// - MonBus error packet generated (if mid-transfer)
```

## Reset Recovery

After reset deassertion:

| Cycle | Event                            |
|-------|----------------------------------|
| 0     | aresetn deasserted (rising edge) |
| 1-5   | Internal state stabilization     |
| 6+    | Ready for APB configuration      |
| 10+   | Ready for descriptor transfers   |

## Clock Requirements by Module

### Functional Unit Blocks (FUB)

| Module            | Clock | Reset   | Frequency   | Notes             |
|-------------------|-------|---------|-------------|-------------------|
| descriptor_engine | aclk  | aresetn | 100-500 MHz | AXI master timing |
| scheduler         | aclk  | aresetn | 100-500 MHz | Single cycle FSM  |
| axi_read_engine   | aclk  | aresetn | 100-500 MHz | AXI master timing |
| axi_write_engine  | aclk  | aresetn | 100-500 MHz | AXI master timing |
| simple_sram       | aclk  | aresetn | 100-500 MHz | Synchronous SRAM  |

### Integration Blocks (MAC)

| Module             | Clock(s)     | Reset(s)           | Frequency        | Notes                    |
|--------------------|--------------|--------------------|------------------|--------------------------|
| channel_arbiter    | aclk         | aresetn            | 100-500 MHz      | Single cycle arbitration |
| apb_config         | pclk, (aclk) | presetn, (aresetn) | 50-200 MHz (APB) | CDC if async             |
| monbus_axi_l_group | aclk         | aresetn            | 100-500 MHz      | AXIL timing              |
| stream_top         | aclk, pclk   | aresetn, presetn   | Mixed            | Top-level                |

## Timing Constraints

### Setup and Hold Times

**Internal registers (relative to `aclk`):** - Setup time: 0.5 ns (typical) - Hold time: 0.1 ns (typical) - Clock-to-Q: 0.3 ns (typical)

**External interfaces:** - AXI/AXIL: Per ARM IHI0022E specification - APB: Per ARM IHI0024C specification

### Critical Paths

**Identified critical paths:**

1. **Arbiter -> Scheduler grant:**
  - Latency: 1 cycle
  - Path: Priority encoder -> One-hot grant
2. **AXI read -> SRAM write:**
  - Latency: 1 cycle
  - Path: R data -> SRAM write port
3. **SRAM read -> AXI write:**
  - Latency: 1 cycle
  - Path: SRAM read port -> W data

**Maximum frequency estimation:** - Typical FPGA (Xilinx 7-series): 250 MHz - High-end FPGA (UltraScale+): 400 MHz - ASIC (28nm): 500 MHz

---

## Clock Domain Crossing (CDC)

### APB Configuration CDC

**When required:** `pclk` `aclk` (asynchronous APB interface)

**CDC Implementation:**

```
// APB to STREAM domain (pclk -> aclk)
apb_slave_cdc #(
    .ADDR_WIDTH(32),
    .DATA_WIDTH(32),
    .SYNC_STAGES(2) // Dual-flop synchronizer
) u_apb_cdc (
```

```

// APB side (pclk domain)
.s_pclk(pclk),
.s_presetn(presetn),
.s_paddr(paddr),
.s_pwrite(pwrite),
.s_pwdata(pwdata),
.s_prdata(prdata),

// STREAM side (aclk domain)
.m_pclk(aclk),
.m_presetn(aresetn),
.m_paddr(paddr_sync),
.m_pwrite(pwrite_sync),
.m_pwdata(pwdata_sync),
.m_prdata(prdata_sync)
);

```

**CDC Mechanism:** - `apb_slave_cdc` uses **handshake-based CDC** via `cdc_handshake` modules (NOT async FIFOs) - **Command path** (`pclk` → `aclk`): APB write/read commands cross via req/ack handshake - **Response path** (`aclk` → `pclk`): APB read data crosses back via req/ack handshake - **Internal synchronizers**: Dual-flop synchronizers (2-3 stages) for handshake signals - **ASYNC\_REG attribute**: Applied to synchronizer stages for timing tools - **Timing constraints**: Proper constraints required in SDC/XDC for synchronizer paths

**Handshake Protocol Benefits:** - Works across **all frequency ratios** (slow-to-fast, fast-to-slow, arbitrary ratios) - Guaranteed data integrity (req/ack ensures data stability before sampling) - No FIFO depth management or gray code pointer complexity - Latency: 4-6 APB clock cycles for register access (handshake round-trip)

## No CDC Required

**Single clock domain:** If `pclk` = `aclk` and `presetn` = `aresetn`:

```

// Direct connection (no CDC wrapper)
apb_config #(
    .NUM_CHANNELS(8)
) u_apb_config (
    .pclk(aclk),           // Same clock
    .presetn(aresetn),     // Same reset
    // ... direct APB signals
);

```

---

## Reset State Initialization

### Register Reset Values

All STREAM modules must initialize to known state on reset:

```
// Descriptor Engine
if (!aresetn) begin
    r_desc_fifo_wr_ptr <= '0;
    r_desc_valid <= 1'b0;
    r_desc_error <= 1'b0;
end

// Scheduler
if (!aresetn) begin
    r_current_state <= CH_IDLE;
    r_read_beats_remaining <= '0;
    r_write_beats_remaining <= '0;
    r_timeout_counter <= '0;
end

// AXI Engines
if (!aresetn) begin
    r_burst_counter <= '0;
    m_axi_arvalid <= 1'b0;
    m_axi_awvalid <= 1'b0;
end

// Arbiter
if (!aresetn) begin
    r_last_grant_id <= '0;
    r_grant_valid <= 1'b0;
end
```

### SRAM Reset

SRAM contents: Undefined after reset (no initialization required)

### SRAM pointers:

```
if (!aresetn) begin
    wr_ptr <= '0;
    rd_ptr <= '0;
end
```

---

## Clock Gating (Optional)

For power optimization in ASIC implementations:

### Per-Channel Clock Gating

```
// Clock gate when channel idle
clock_gate_ctrl u_ch0_clk_gate (
    .clk_in(aclk),
    .enable(ch0_enable),
    .clk_out(ch0_gated_clk)
);

// Use gated clock for channel logic
scheduler #(.CHANNEL_ID(0)) u_ch0_sched (
    .aclk(ch0_gated_clk), // Gated clock
    .aresetn(aresetn),
    // ...
);
```

**Note:** Clock gating typically not used in FPGA implementations (tutorial focus).

---

## Verification Requirements

### Clock Checks

**Testbench must verify:** - [Done] Clock period consistent - [Done] Clock duty cycle 50% tolerance - [Done] No glitches on clock - [Done] Setup/hold times met

### Reset Checks

**Testbench must verify:** - [Done] All registers initialize to known state - [Done] Reset assertion clears FSMs to IDLE - [Done] Reset deassertion synchronous to clock - [Done] Minimum reset duration (10 cycles) enforced - [Done] Operations don't start until stabilization complete

### CDC Checks

**For APB CDC (if present):** - [Done] No metastability violations - [Done] Data integrity across domains - [Done] Proper flag synchronization

---

## Example Reset Testbench

```
# CocoTB testbench pattern
class StreamTB(TBBase):
```

```

async def setup_clocks_and_reset(self):
    """Complete clock and reset initialization"""
    # Start clocks
    await self.start_clock('aclk', freq=10, units='ns') # 100 MHz
    await self.start_clock('pclk', freq=20, units='ns') # 50 MHz
(async

    # Assert reset
    await self.assert_reset()

    # Hold reset for 10 aclk cycles
    await self.wait_clocks('aclk', 10)

    # Deassert reset (synchronous to aclk)
    await self.deassert_reset()

    # Stabilization period
    await self.wait_clocks('aclk', 5)

    # Ready for operation

async def assert_reset(self):
    """Assert both resets"""
    self.dut.aresetn.value = 0
    self.dut.presetn.value = 0

async def deassert_reset(self):
    """Deassert both resets synchronously"""
    # Wait for rising edge of aclk
    await RisingEdge(self.dut.aclk)
    self.dut.aresetn.value = 1

    # Wait for rising edge of pclk
    await RisingEdge(self.dut.pclk)
    self.dut.presetn.value = 1

```

---

## Related Documentation

- **Scheduler FSM:** fub\_02\_scheduler.md - Reset behavior
  - **APB Config:** mac\_02\_apb\_config.md - CDC implementation
  - **Top-Level:** mac\_04\_stream\_top.md - Clock/reset integration
-

# STREAM Core Specification

**Module:** stream\_core.sv **Location:** projects/components/stream/rtl/macro/  
**Status:** Implemented **Last Updated:** 2025-11-30

---

## Overview

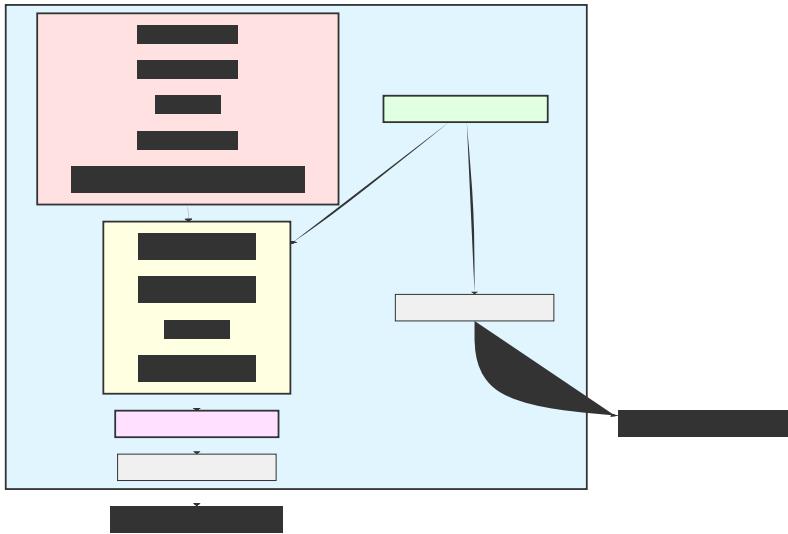
The STREAM Core is the top-level integration module that combines all STREAM components into a complete scatter-gather DMA engine. It provides 8 independent channels with descriptor-based memory-to-memory transfers.

## Key Features

- **8 Independent Channels:** Concurrent descriptor-based transfers
- **Shared AXI Masters:** Three shared AXI4 masters for efficiency
  - Descriptor fetch (256-bit fixed)
  - Data read (parameterizable, default 512-bit)
  - Data write (parameterizable, default 512-bit)
- **Per-Channel Buffering:** Independent FIFO per channel (512 entries default)
- **Performance Monitoring:** Integrated profiler with FIFO readout
- **AXI Skid Buffers:** Timing closure on all external interfaces
- **Unified MonBus:** Single monitor bus output for all events

## Block Diagram

The STREAM Core integrates the following major components:



*STREAM Core Block Diagram*

**Source:** [01\\_stream\\_core\\_block.mmd](#)

---

## Architecture

### Component Hierarchy

1. **scheduler\_group\_array** - Top scheduler layer
  - 8 × scheduler\_group instances
  - Shared descriptor engine
  - Descriptor fetch arbitration
2. **sram\_controller** - Buffering layer
  - 8 × independent FIFOs (gaxi\_fifo\_sync)
  - Per-channel allocation controllers
  - Per-channel drain controllers
3. **axi\_read\_engine** - Read datapath
  - Shared AXI master for all channels
  - ID-based routing to SRAM
  - Space allocation flow control
4. **axi\_write\_engine** - Write datapath
  - Shared AXI master for all channels
  - ID-based routing from SRAM
  - Drain reservation flow control
5. **perf\_profiler** - Performance monitoring

- Transaction counting
- Bandwidth tracking
- FIFO readout interface

## 6. AXI Skid Buffers - Timing closure

- Descriptor AXI (AR/R)
- Read AXI (AR/R)
- Write AXI (AW/W/B)

### Data Flow

#### Descriptor Fetch Flow:

```
sequenceDiagram
    participant APB as APB Write
    participant SCHED as Scheduler
    participant DESC as Descriptor Engine
    participant MEM as m_axi_desc_*
    APB->>SCHED: descriptor address
    SCHED->>DESC: fetch request
    DESC->>MEM: AXI read
    MEM-->>DESC: 256-bit descriptor
    DESC->>SCHED: parsed descriptor
```

#### Data Transfer Flow:

```
sequenceDiagram
    participant SCHED as Scheduler
    participant RD as Read Engine
    participant RD_MEM as m_axi_rd_*
    participant SRAM as SRAM Controller
    participant FIFO as FIFO[channel_id]
    participant WR as Write Engine
    participant WR_MEM as m_axi_wr_*
    SCHED->>RD: source address, length
    RD->>RD_MEM: AXI read
    RD_MEM-->>SRAM: data + channel ID
    SRAM->>FIFO: buffered data
    FIFO->>WR: drain request
    WR->>WR_MEM: AXI write
```

---

## Parameters

### Primary Configuration

| Parameter    | Type | Default               | Description                                  |
|--------------|------|-----------------------|----------------------------------------------|
| NUM_CHANNELS | int  | 8                     | Number of independent DMA channels           |
| CHAN_WIDTH   | int  | \$clog2(NUM_CHANNELS) | Channel ID width (3 for 8 channels)          |
| ADDR_WIDTH   | int  | 64                    | Address bus width                            |
| DATA_WIDTH   | int  | 512                   | Data bus width (must match memory interface) |
| AXI_ID_WIDTH | int  | 8                     | AXI transaction ID width                     |
| FIFO_DEPTH   | int  | 512                   | Per-channel FIFO depth                       |

### Monitor Control

| Parameter        | Type | Default | Description                                        |
|------------------|------|---------|----------------------------------------------------|
| USE_AXI_MONITORS | int  | 1       | Enable (1) or disable (0) AXI transaction monitors |

**Note:** When USE\_AXI\_MONITORS = 0: - All monitor configuration inputs are tied off internally - MonBus output remains present but inactive (no packets generated) - Reduces resource usage for production systems

### Outstanding Transaction Limits

| Parameter          | Default | Description                               |
|--------------------|---------|-------------------------------------------|
| AR_MAX_OUTSTANDING | 8       | Maximum concurrent read address requests  |
| AW_MAX_OUTSTANDING | 8       | Maximum concurrent write address requests |

### AXI Skid Buffer Depths

| Parameter     | Default | Purpose                   |
|---------------|---------|---------------------------|
| SKID_DEPTH_AR | 2       | AR channel timing closure |
| SKID_DEPTH_R  | 4       | R channel timing          |

| Parameter     | Default | Purpose                              |
|---------------|---------|--------------------------------------|
| SKID_DEPTH_AW | 2       | closure<br>AW channel timing closure |
| SKID_DEPTH_W  | 4       | W channel timing closure             |
| SKID_DEPTH_B  | 2       | B channel timing closure             |

**Note:** Deeper buffers on data channels (R/W) improve throughput.

### MonBus Agent IDs

| Parameter               | Default   | Description                     |
|-------------------------|-----------|---------------------------------|
| DESC_MON_BASE_AGENT_ID  | 16 (0x10) | Descriptor engines base (16-23) |
| SCHED_MON_BASE_AGENT_ID | 48 (0x30) | Schedulers base (48-55)         |
| DESC_AXI_MON_AGENT_ID   | 8 (0x08)  | Descriptor AXI master monitor   |
| MON_UNIT_ID             | 1 (0x1)   | Unit ID for all STREAM events   |

### Port List

#### Clock and Reset

| Signal | Direction | Width | Description                   |
|--------|-----------|-------|-------------------------------|
| clk    | input     | 1     | System clock                  |
| rst_n  | input     | 1     | Active-low asynchronous reset |

### APB Programming Interface

Per-channel descriptor kick-off interface:

| Signal        | Direction | Width            | Description                      |
|---------------|-----------|------------------|----------------------------------|
| apb_valid[ch] | input     | NUM_CHAN<br>NELS | Channel descriptor address valid |
| apb_ready[ch] | output    | NUM_CHAN         | Channel ready to accept          |

| Signal       | Direction | Width                                | Description                    |
|--------------|-----------|--------------------------------------|--------------------------------|
| ]            |           | NELS                                 | descriptor address             |
| apb_addr[ch] | input     | NUM_CHAN<br>NELS ×<br>ADDR_WIDT<br>H | Descriptor address per channel |

### Usage:

```
// Kick off channel 0 with descriptor at 0x1000_0000
apb_valid[0] = 1'b1;
apb_addr[0] = 64'h0000_0000_1000_0000;
// Wait for handshake
wait (apb_ready[0]);
apb_valid[0] = 1'b0;
```

## Configuration Interface

### Per-Channel Configuration:

| Signal                 | Direction | Width            | Description                     |
|------------------------|-----------|------------------|---------------------------------|
| cfg_channel_enable[ch] | input     | NUM_CHAN<br>NELS | Enable channel                  |
| cfg_channel_reset[ch]  | input     | NUM_CHAN<br>NELS | Soft reset channel (FSM → IDLE) |

### Global Scheduler Configuration:

| Signal                       | Direction | Width | Description                        |
|------------------------------|-----------|-------|------------------------------------|
| cfg_sched_enable             | input     | 1     | Global scheduler enable            |
| cfg_sched_timeout_cycles     | input     | 16    | Timeout threshold (cycles)         |
| cfg_sched_timeout_enable     | input     | 1     | Enable timeout detection           |
| cfg_sched_error_report       | input     | 1     | Enable error event reporting       |
| cfg_sched_completion_report  | input     | 1     | Enable completion event reporting  |
| cfg_sched_performance_report | input     | 1     | Enable performance event reporting |

## Descriptor Engine Configuration:

| Signal                  | Direction | Width       | Description                 |
|-------------------------|-----------|-------------|-----------------------------|
| cfg_desceng_enable      | input     | 1           | Enable descriptor engine    |
| cfg_desceng_prefetch    | input     | 1           | Enable descriptor prefetch  |
| cfg_desceng_fifo_thresh | input     | 4           | FIFO threshold for prefetch |
| cfg_desceng_addr0_base  | input     | ADDR_WIDT_H | Base address limit 0        |
| cfg_desceng_addr0_limit | input     | ADDR_WIDT_H | Limit address limit 0       |
| cfg_desceng_addr1_base  | input     | ADDR_WIDT_H | Base address limit 1        |
| cfg_desceng_addr1_limit | input     | ADDR_WIDT_H | Limit address limit 1       |

## AXI Monitor Configuration:

Three identical sets of monitor config signals (descriptor, read, write):

| Signal Prefix   | Applies To     | Description                 |
|-----------------|----------------|-----------------------------|
| cfg_desc_mon_*  | Descriptor AXI | Descriptor fetch monitoring |
| cfg_rdeng_mon_* | Read AXI       | Data read monitoring        |
| cfg_wreng_mon_* | Write AXI      | Data write monitoring       |

Each monitor has:

| Signal Suffix | Width | Description                  |
|---------------|-------|------------------------------|
| _enable       | 1     | Enable monitor               |
| _err_enable   | 1     | Enable error reporting       |
| _perf_enable  | 1     | Enable performance reporting |

| Signal Suffix   | Width | Description                  |
|-----------------|-------|------------------------------|
| _timeout_enable | 1     | Enable timeout detection     |
| _timeout_cycles | 32    | Timeout threshold            |
| _latency_thresh | 32    | Latency threshold for events |
| _pkt_mask       | 16    | Packet type mask             |
| _err_select     | 4     | Error type selector          |
| _err_mask       | 8     | Error event mask             |
| _timeout_mask   | 8     | Timeout event mask           |
| _compl_mask     | 8     | Completion event mask        |
| _thresh_mask    | 8     | Threshold event mask         |
| _perf_mask      | 8     | Performance event mask       |
| _addr_mask      | 8     | Address event mask           |
| _debug_mask     | 8     | Debug event mask             |

#### AXI Transfer Configuration:

| Signal                | Direction | Width | Description              |
|-----------------------|-----------|-------|--------------------------|
| cfg_axi_rd_xfer_beats | input     | 8     | Read burst size (beats)  |
| cfg_axi_wr_xfer_beats | input     | 8     | Write burst size (beats) |

#### Performance Profiler Configuration:

| Signal          | Direction | Width | Description                        |
|-----------------|-----------|-------|------------------------------------|
| cfg_perf_enable | input     | 1     | Enable profiler                    |
| cfg_perf_mode   | input     | 1     | Profiler mode (0=count, 1=latency) |
| cfg_perf_clear  | input     | 1     | Clear profiler counters            |

## Status Interface

### System-Level Status:

| Signal      | Direction | Width | Description                                   |
|-------------|-----------|-------|-----------------------------------------------|
| system_idle | output    | 1     | All channels idle (AND of all scheduler_idle) |

### Per-Channel Status:

| Signal                     | Direction | Width                | Description                     |
|----------------------------|-----------|----------------------|---------------------------------|
| descriptor_engine_idle[ch] | output    | NUM_CHAN<br>NELS     | Descriptor engine idle          |
| scheduler_idle[ch]         | output    | NUM_CHAN<br>NELS     | Scheduler in IDLE state         |
| scheduler_state[ch]        | output    | NUM_CHAN<br>NELS × 7 | Scheduler FSM state (ONE-HOT)   |
| sched_error[ch]            | output    | NUM_CHAN<br>NELS     | Scheduler error (sticky)        |
| axi_rd_all_complete[ch]    | output    | NUM_CHAN<br>NELS     | All read transactions complete  |
| axi_wr_all_complete[ch]    | output    | NUM_CHAN<br>NELS     | All write transactions complete |

### Performance Profiler Status:

| Signal             | Direction | Width | Description             |
|--------------------|-----------|-------|-------------------------|
| perf_fifo_empty    | output    | 1     | Profiler FIFO empty     |
| perf_fifo_full     | output    | 1     | Profiler FIFO full      |
| perf_fifo_count    | output    | 16    | Profiler FIFO occupancy |
| perf_fifo_rd       | input     | 1     | Read profiler entry     |
| perf_fifo_data_low | output    | 32    | Profiler data [31:0]    |
| perf_fifo_data     | output    | 32    | Profiler data           |

| Signal | Direction | Width   | Description |
|--------|-----------|---------|-------------|
| a_high |           | [63:32] |             |

### AXI4 Master - Descriptor Fetch (256-bit)

#### AR Channel:

| Signal               | Direction | Width        | Description              |
|----------------------|-----------|--------------|--------------------------|
| m_axi_desc_ar_id     | output    | AXI_ID_WIDTH | Transaction ID           |
| m_axi_desc_ar_addr   | output    | ADDR_WIDTH   | Address                  |
| m_axi_desc_ar_len    | output    | 8            | Burst length - 1         |
| m_axi_desc_ar_size   | output    | 3            | Burst size (log2 bytes)  |
| m_axi_desc_ar_burst  | output    | 2            | Burst type (INCR)        |
| m_axi_desc_ar_lock   | output    | 1            | Lock type                |
| m_axi_desc_ar_cache  | output    | 4            | Cache attributes         |
| m_axi_desc_ar_prot   | output    | 3            | Protection attributes    |
| m_axi_desc_ar_qos    | output    | 4            | QoS value                |
| m_axi_desc_ar_region | output    | 4            | Region identifier        |
| m_axi_desc_ar_user   | output    | CHAN_WIDTH   | User signal (channel ID) |
| m_axi_desc_ar_valid  | output    | 1            | Address valid            |
| m_axi_desc_ar_ready  | input     | 1            | Address ready            |

#### R Channel:

| Signal          | Direction | Width         | Description    |
|-----------------|-----------|---------------|----------------|
| m_axi_desc_r_id | input     | AXI_ID_WID TH | Transaction ID |

| Signal             | Direction | Width      | Description                          |
|--------------------|-----------|------------|--------------------------------------|
| m_axi_desc_r_data  | input     | 256        | Read data (FIXED 256-bit)            |
| m_axi_desc_r_resp  | input     | 2          | Response (OKAY/EXOKAY/SLVERR/DECERR) |
| m_axi_desc_r_last  | input     | 1          | Last beat of burst                   |
| m_axi_desc_r_user  | input     | CHAN_WIDTH | User signal (channel ID)             |
| m_axi_desc_r_valid | input     | 1          | Read data valid                      |
| m_axi_desc_r_ready | output    | 1          | Read data ready                      |

### AXI4 Master - Data Read (Parameterizable Width)

#### AR Channel:

| Signal            | Direction | Width        | Description             |
|-------------------|-----------|--------------|-------------------------|
| m_axi_rd_arid     | output    | AXI_ID_WIDTH | Transaction ID          |
| m_axi_rd_arad_dr  | output    | ADDR_WIDTH   | Address                 |
| m_axi_rd_arlen    | output    | 8            | Burst length - 1        |
| m_axi_rd_arze     | output    | 3            | Burst size (log2 bytes) |
| m_axi_rd_arburst  | output    | 2            | Burst type (INCR)       |
| m_axi_rd_arlock   | output    | 1            | Lock type               |
| m_axi_rd_arcache  | output    | 4            | Cache attributes        |
| m_axi_rd_arprot   | output    | 3            | Protection attributes   |
| m_axi_rd_arqos    | output    | 4            | QoS value               |
| m_axi_rd_arregion | output    | 4            | Region identifier       |

| Signal           | Direction | Width      | Description              |
|------------------|-----------|------------|--------------------------|
| m_axi_rd_aruser  | output    | CHAN_WIDTH | User signal (channel ID) |
| m_axi_rd_arvalid | output    | 1          | Address valid            |
| m_axi_rd_arready | input     | 1          | Address ready            |

### R Channel:

| Signal          | Direction | Width        | Description                 |
|-----------------|-----------|--------------|-----------------------------|
| m_axi_rd_rid    | input     | AXI_ID_WIDTH | Transaction ID              |
| m_axi_rd_rdata  | input     | DATA_WIDTH   | Read data (default 512-bit) |
| m_axi_rd_rresp  | input     | 2            | Response                    |
| m_axi_rd_rlast  | input     | 1            | Last beat of burst          |
| m_axi_rd_ruser  | input     | CHAN_WIDTH   | User signal (channel ID)    |
| m_axi_rd_rvalid | input     | 1            | Read data valid             |
| m_axi_rd_rready | output    | 1            | Read data ready             |

### AXI4 Master - Data Write (Parameterizable Width)

#### AW Channel:

| Signal           | Direction | Width        | Description             |
|------------------|-----------|--------------|-------------------------|
| m_axi_wr_awid    | output    | AXI_ID_WIDTH | Transaction ID          |
| m_axi_wr_awaddr  | output    | ADDR_WIDTH   | Address                 |
| m_axi_wr_awlen   | output    | 8            | Burst length - 1        |
| m_axi_wr_awsize  | output    | 3            | Burst size (log2 bytes) |
| m_axi_wr_awburst | output    | 2            | Burst type              |

| Signal             | Direction | Width      | Description              |
|--------------------|-----------|------------|--------------------------|
|                    |           |            | (INCR)                   |
| m_axi_wr_awlo_ck   | output    | 1          | Lock type                |
| m_axi_wr_awca_ch   | output    | 4          | Cache attributes         |
| m_axi_wr_awpr_ot   | output    | 3          | Protection attributes    |
| m_axi_wr_awqo_s    | output    | 4          | QoS value                |
| m_axi_wr_awre_gion | output    | 4          | Region identifier        |
| m_axi_wr_awus_er   | output    | CHAN_WIDTH | User signal (channel ID) |
| m_axi_wr_awva_lid  | output    | 1          | Address valid            |
| m_axi_wr_awre_ady  | input     | 1          | Address ready            |

### W Channel:

| Signal           | Direction | Width        | Description                  |
|------------------|-----------|--------------|------------------------------|
| m_axi_wr_wda_ta  | output    | DATA_WIDTH   | Write data (default 512-bit) |
| m_axi_wr_wst_rb  | output    | DATA_WIDTH/8 | Write strobes (byte enables) |
| m_axi_wr_wla_st  | output    | 1            | Last beat of burst           |
| m_axi_wr_wus_er  | output    | CHAN_WIDTH   | User signal (channel ID)     |
| m_axi_wr_wva_lid | output    | 1            | Write data valid             |
| m_axi_wr_wre_ady | input     | 1            | Write data ready             |

### B Channel:

| Signal        | Direction | Width        | Description    |
|---------------|-----------|--------------|----------------|
| m_axi_wr_bid  | input     | AXI_ID_WIDTH | Transaction ID |
| m_axi_wr_bres | input     | 2            | Response       |

| Signal              | Direction | Width      | Description                 |
|---------------------|-----------|------------|-----------------------------|
| p                   |           |            |                             |
| m_axi_wr_buse<br>r  | input     | CHAN_WIDTH | User signal<br>(channel ID) |
| m_axi_wr_bval<br>id | input     | 1          | Response valid              |
| m_axi_wr_brea<br>dy | output    | 1          | Response<br>ready           |

## Status/Debug Outputs

### Descriptor AXI Monitor:

| Signal                                  | Direction | Width | Description              |
|-----------------------------------------|-----------|-------|--------------------------|
| cfg_sts_desc<br>_mon_busy               | output    | 1     | Monitor busy             |
| cfg_sts_desc<br>_mon_active_<br>txns    | output    | 8     | Active transaction count |
| cfg_sts_desc<br>_mon_error_c<br>ount    | output    | 16    | Error count              |
| cfg_sts_desc<br>_mon_txn_cou<br>nt      | output    | 32    | Total transaction count  |
| cfg_sts_desc<br>_mon_conflic<br>t_error | output    | 1     | ID conflict detected     |

### Read Engine AXI Monitor:

| Signal                                | Direction | Width | Description              |
|---------------------------------------|-----------|-------|--------------------------|
| cfg_sts_rden<br>g_skid_busy           | output    | 1     | Skid buffer busy         |
| cfg_sts_rden<br>g_mon_active_<br>txns | output    | 8     | Active transaction count |
| cfg_sts_rden<br>g_mon_error_<br>count | output    | 16    | Error count              |
| cfg_sts_rden<br>g_mon_txn_co<br>unt   | output    | 32    | Total transaction count  |

| Signal               | Direction | Width | Description          |
|----------------------|-----------|-------|----------------------|
| cfg_sts_rden         | output    | 1     | ID conflict detected |
| g_mon_conflict_error |           |       |                      |

### Write Engine AXI Monitor:

| Signal               | Direction | Width | Description              |
|----------------------|-----------|-------|--------------------------|
| cfg_sts_wren         | output    | 1     | Skid buffer busy         |
| g_skid_busy          |           |       |                          |
| cfg_sts_wren         | output    | 8     | Active transaction count |
| g_mon_active_txns    |           |       |                          |
| cfg_sts_wren         | output    | 16    | Error count              |
| g_mon_error_count    |           |       |                          |
| cfg_sts_wren         | output    | 32    | Total transaction count  |
| g_mon_txn_count      |           |       |                          |
| cfg_sts_wren         | output    | 1     | ID conflict detected     |
| g_mon_conflict_error |           |       |                          |

### Unified Monitor Bus Interface

| Signal     | Direction | Width | Description          |
|------------|-----------|-------|----------------------|
| mon_valid  | output    | 1     | Monitor packet valid |
| mon_ready  | input     | 1     | Monitor packet ready |
| mon_packet | output    | 64    | Monitor packet data  |

**MonBus Sources:** - Descriptor engines (8 sources, agent IDs 16-23) - Schedulers (8 sources, agent IDs 48-55) - Descriptor AXI monitor (agent ID 8) - Read AXI monitor (configurable) - Write AXI monitor (configurable)

---

## Operation

### Transfer Initialization

#### Step 1: Configuration

1. Configure global settings (timeouts, monitors, transfer sizes)
2. Enable descriptor engine (cfg\_desceng\_enable = 1)
3. Enable scheduler (cfg\_sched\_enable = 1)
4. Enable target channel (cfg\_channel\_enable[ch] = 1)

### **Step 2: Descriptor Kick-off**

1. Assert apb\_valid[ch] = 1
2. Provide descriptor address on apb\_addr[ch]
3. Wait for apb\_ready[ch] handshake
4. Deassert apb\_valid[ch]

### **Step 3: Automatic Transfer**

1. Descriptor engine fetches descriptor via m\_axi\_desc\_\*
2. Scheduler receives descriptor, starts transfer
3. Read engine: memory → SRAM (via m\_axi\_rd\_\*)
4. Write engine: SRAM → memory (via m\_axi\_wr\_\*)
5. Scheduler reports completion via MonBus

### **Monitoring Transfer Progress**

#### **Check Scheduler State:**

```
// Monitor scheduler state
case (scheduler_state[ch])
  7'b0000001: // CH_IDLE - waiting for descriptor
  7'b0000010: // CH_FETCH_DESC - fetching descriptor
  7'b0000100: // CH_XFER_DATA - transfer in progress
  7'b0001000: // CH_COMPLETE - transfer complete
  7'b0010000: // CH_NEXT_DESC - chaining to next
  7'b0100000: // CH_ERROR - error occurred
endcase
```

#### **Check Completion:**

```
// All complete when:
complete = scheduler_idle[ch] &&
           axi_rd_all_complete[ch] &&
           axi_wr_all_complete[ch];
```

---

### **Testing**

**Test Location:** projects/components/stream/dv/tests/top/

#### **Key Test Scenarios:**

1. **Single channel transfer** - Basic end-to-end operation

2. **Multi-channel concurrent** - 2-8 channels simultaneously
3. **Descriptor chaining** - 2-5 descriptors linked
4. **FIFO overflow prevention** - Large transfer with small FIFO
5. **Error handling** - AXI errors, timeouts
6. **Performance profiling** - Bandwidth measurements
7. **MonBus event checking** - Verify all events reported

### Test Configuration:

```
# Basic configuration
NUM_CHANNELS = 4
DATA_WIDTH = 128
FIFO_DEPTH = 512
cfg_axi_rd_xfer_beats = 16
cfg_axi_wr_xfer_beats = 16
```

---

## Resource Utilization

### Estimated Resources (8 channels, 512-bit data, 512-deep FIFOs):

| Component         | Quantity | Est. Size                    |
|-------------------|----------|------------------------------|
| Schedulers        | 8        | 8 × ~500 FFs                 |
| Descriptor Engine | 1        | ~1000 FFs                    |
| SRAM FIFOs        | 8        | 8 × 512 × 512-bit =<br>256KB |
| AXI Engines       | 2        | 2 × ~2000 FFs                |
| Skid Buffers      | 3 sets   | ~2000 FFs                    |
| Monitors          | 3        | 3 × ~1000 FFs                |
| <b>Total</b>      |          | ~20K FFs + 256KB<br>SRAM     |

**Critical Paths:** - AXI handshake paths (improved by skid buffers) - SRAM address decode - MonBus arbiter

---

## Integration Example

```
stream_core #(
    .NUM_CHANNELS(8),
    .DATA_WIDTH(512),
```

```

    .FIFO_DEPTH(512)
) u_stream (
    .clk                      (system_clk),
    .rst_n                    (system_rst_n),
    // APB kick-off
    .apb_valid                (stream_apb_valid),
    .apb_ready                (stream_apb_ready),
    .apb_addr                 (stream_apb_addr),
    // Configuration
    .cfg_channel_enable       (stream_ch_enable),
    .cfg_sched_enable         (1'b1),
    .cfg_axi_rd_xfer_beats   (8'd16),
    .cfg_axi_wr_xfer_beats   (8'd16),
    // ... other config

    // AXI Descriptor Master
    .m_axi_desc_arid          (desc_arid),
    .m_axi_desc_araddr         (desc_araddr),
    // ... full AXI AR/R

    // AXI Read Master
    .m_axi_rd_arid            (rd_arid),
    .m_axi_rd_araddr           (rd_araddr),
    // ... full AXI AR/R

    // AXI Write Master
    .m_axi_wr_awid             (wr_awid),
    .m_axi_wr_awaddr            (wr_awaddr),
    // ... full AXI AW/W/B

    // MonBus
    .mon_valid                (stream_mon_valid),
    .mon_ready                (stream_mon_ready),
    .mon_packet               (stream_mon_packet)
);

```

---

## Related Documentation

- **Scheduler Group Array:** 02\_scheduler\_group\_array.md - Multi-channel scheduler integration
- **Scheduler Group:** 03\_scheduler\_group.md - Single channel scheduler + descriptor engine

- **Scheduler:** 04\_scheduler.md - Channel state machine and transfer coordination
  - **Descriptor Engine:** 05\_descriptor\_engine.md - Descriptor fetch and parsing
  - **AXI Read Engine:** 06\_axi\_read\_engine.md - Read datapath
  - **Stream Alloc Ctrl:** 07\_stream\_alloc\_ctrl.md - Space allocation controller
  - **SRAM Controller:** 08\_sram\_controller.md - Per-channel buffering
  - **SRAM Controller Unit:** 09\_sram\_controller\_unit.md - Single channel SRAM unit
  - **Stream Latency Bridge:** 10\_stream\_latency\_bridge.md - Timing bridge
  - **Stream Drain Ctrl:** 11\_stream\_drain\_ctrl.md - Drain flow controller
  - **AXI Write Engine:** 12\_axi\_write\_engine.md - Write datapath
  - **APB to Descriptor:** 13\_apbtodescr.md - APB configuration interface
  - **Performance Profiler:** 15\_perf\_profiler.md - Performance monitoring
  - **MonBus AXI-Lite Group:** 16\_monbus\_axil\_group.md - Monitor bus arbitration
- 

**Last Updated:** 2025-11-30 (verified against RTL implementation)

## Scheduler Group Array

**Module:** scheduler\_group\_array.sv **Location:**

projects/components/stream/rtl/macro/ **Category:** MACRO (Top-level

Integration) **Parent:** stream\_core.sv **Status:** Implemented **Last Updated:** 2025-11-30

---

### Overview

The scheduler\_group\_array module is the top-level array of 8 scheduler\_group instances that provides multi-channel DMA with shared resources. It serves as the scheduler layer of the STREAM engine, handling descriptor fetch arbitration and routing data path interfaces.

### Key Features

- **8 Independent Channels:** Each channel has its own scheduler\_group instance

- **Shared Descriptor AXI Master:** Round-robin arbitration for descriptor fetches
- **Per-Channel Data Interfaces:** Direct passthrough to AXI engines (no arbitration)
- **Unified MonBus Aggregation:** 9 sources (8 channels + descriptor AXI monitor)
- **ID-Based R Channel Routing:** Channel ID embedded in AXI ID for response demux

## Simplified from RAPIDS

This module is intentionally simplified from RAPIDS for tutorial focus: - 8 channels (vs 32 in RAPIDS) - No control read/write engines - No program engine - No network interfaces (RDA, EOS completion) - No alignment bus - Direct data path interfaces (beats-based, no chunks)

---

## Architecture

### Component Hierarchy

```

scheduler_group_array
  scheduler_group[0..7]          # 8 channel instances
    scheduler                   # Channel FSM
    descriptor_engine          # Descriptor fetch
  arbiter_round_robin           # AR channel arbitration
  axi4_master_rd_mon           # Descriptor AXI monitor
  monbus_arbiter               # 9-source MonBus aggregation

```

### Descriptor AXI Arbitration

#### AR Channel Flow:

Channel 0..7 AR Requests -> Round-Robin Arbiter -> Single AR to External AXI

|  
Embed channel\_id in AXI ID lower bits

#### R Channel Demux:

External R Response -> Extract channel\_id from RID[CHAN\_WIDTH-1:0] -> Route to correct channel

**Critical:** Channel ID is embedded in the lower bits of the AXI ID field:

```
// In AR mux
desc_axi_int_arid = {{(AXI_ID_WIDTH-CHAN_WIDTH){1'b0}}, ch[CHAN_WIDTH-
```

```

1:0]};

// In R demux
desc_r_channel_id = desc_axi_int_rid[CHAN_WIDTH-1:0];

```

## Data Path Interfaces

**IMPORTANT:** Data read and write interfaces are NOT arbitrated at this level!

- `sched_rd_*` - Per-channel arrays passed directly to `axi_read_engine`
- `sched_wr_*` - Per-channel arrays passed directly to `axi_write_engine`

The AXI engines themselves handle channel arbitration internally.

---

## Parameters

| Parameter    | Type | Default               | Description                              |
|--------------|------|-----------------------|------------------------------------------|
| NUM_CHANNELS | int  | 8                     | Number of DMA channels                   |
| CHAN_WIDTH   | int  | \$clog2(NUM_CHANNELS) | Channel ID width (3 bits for 8 channels) |
| ADDR_WIDTH   | int  | 64                    | Address bus width                        |
| DATA_WIDTH   | int  | 512                   | Data bus width                           |
| AXI_ID_WIDTH | int  | 8                     | AXI transaction ID width                 |

## Monitor Bus Agent IDs

| Parameter               | Default   | Description                              |
|-------------------------|-----------|------------------------------------------|
| DESC_MON_BASE_AGENT_ID  | 16 (0x10) | Descriptor engine base (16-23)           |
| SCHED_MON_BASE_AGENT_ID | 48 (0x30) | Scheduler base (48-55)                   |
| DESC_AXI_MON_AGENT_ID   | 8 (0x08)  | Descriptor AXI monitor                   |
| MON_UNIT_ID             | 1         | Unit ID for all STREAM events            |
| MON_MAX_TRANSACTIONS    | 16        | Max outstanding transactions for monitor |

## Port List

### Clock and Reset

| Signal | Direction | Width | Description                   |
|--------|-----------|-------|-------------------------------|
| clk    | input     | 1     | System clock                  |
| rst_n  | input     | 1     | Active-low asynchronous reset |

### APB Programming Interface (Per-Channel)

| Signal        | Direction | Width                                | Description                |
|---------------|-----------|--------------------------------------|----------------------------|
| apb_valid[ch] | input     | NUM_CHAN<br>NELS                     | Descriptor address valid   |
| apb_ready[ch] | output    | NUM_CHAN<br>NELS                     | Ready to accept descriptor |
| apb_addr[ch]  | input     | NUM_CHAN<br>NELS x<br>ADDR_WIDT<br>H | Descriptor address         |

### Configuration Interface (Per-Channel)

| Signal                 | Direction | Width        | Description        |
|------------------------|-----------|--------------|--------------------|
| cfg_channel_enable[ch] | input     | NUM_CHANNELS | Enable channel     |
| cfg_channel_reset[ch]  | input     | NUM_CHANNELS | Soft reset channel |

### Global Scheduler Configuration

| Signal                   | Direction | Width | Description              |
|--------------------------|-----------|-------|--------------------------|
| cfg_sched_enable         | input     | 1     | Master scheduler enable  |
| cfg_sched_timeout_cycles | input     | 16    | Timeout threshold        |
| cfg_sched_timeout_enable | input     | 1     | Enable timeout detection |
| cfg_sched_error_enable   | input     | 1     | Enable error             |

| Signal                 | Direction | Width | Description                   |
|------------------------|-----------|-------|-------------------------------|
| cfg_sched_compl_enable | input     | 1     | Enable completion reporting   |
| cfg_sched_perf_enable  | input     | 1     | Enable performance monitoring |

### Descriptor Engine Configuration

| Signal                  | Direction | Width       | Description                 |
|-------------------------|-----------|-------------|-----------------------------|
| cfg_desceng_enable      | input     | 1           | Enable descriptor engine    |
| cfg_desceng_prefetch    | input     | 1           | Enable descriptor prefetch  |
| cfg_desceng_fifo_thresh | input     | 4           | FIFO threshold for prefetch |
| cfg_desceng_addr0_base  | input     | ADDR_WIDT_H | Address range 0 base        |
| cfg_desceng_addr0_limit | input     | ADDR_WIDT_H | Address range 0 limit       |
| cfg_desceng_addr1_base  | input     | ADDR_WIDT_H | Address range 1 base        |
| cfg_desceng_addr1_limit | input     | ADDR_WIDT_H | Address range 1 limit       |

### Status Interface (Per-Channel)

| Signal                     | Direction | Width             | Description                   |
|----------------------------|-----------|-------------------|-------------------------------|
| descriptor_engine_idle[ch] | output    | NUM_CHAN_NELS     | Descriptor engine idle        |
| scheduler_idle[ch]         | output    | NUM_CHAN_NELS     | Scheduler idle                |
| scheduler_state[ch]        | output    | NUM_CHAN_NELS x 7 | Scheduler FSM state (ONE-HOT) |
| sched_error[ch]            | output    | NUM_CHAN_NELS     | Scheduler error (sticky)      |

## Shared Descriptor AXI4 Master Read Interface

### AR Channel:

| Signal             | Direction | Width        | Description             |
|--------------------|-----------|--------------|-------------------------|
| desc_axi_arva_lid  | output    | 1            | Address valid           |
| desc_axi_arre_ady  | input     | 1            | Address ready           |
| desc_axi_arad_dr   | output    | ADDR_WIDTH   | Address                 |
| desc_axi_arle_n    | output    | 8            | Burst length - 1        |
| desc_axi_arsi_ze   | output    | 3            | Burst size (log2 bytes) |
| desc_axi_arbu_rst  | output    | 2            | Burst type (INCR)       |
| desc_axi_arid      | output    | AXI_ID_WIDTH | Transaction ID          |
| desc_axi_arlo_ck   | output    | 1            | Lock type               |
| desc_axi_arca_che  | output    | 4            | Cache attributes        |
| desc_axi_arpr_ot   | output    | 3            | Protection attributes   |
| desc_axi_arqs      | output    | 4            | QoS value               |
| desc_axi_arre_gion | output    | 4            | Region identifier       |

### R Channel (FIXED 256-bit):

| Signal           | Direction | Width | Description                          |
|------------------|-----------|-------|--------------------------------------|
| desc_axi_rva_lid | input     | 1     | Read data valid                      |
| desc_axi_rre_ady | output    | 1     | Read data ready                      |
| desc_axi_rda_ta  | input     | 256   | Read data (FIXED 256-bit descriptor) |
| desc_axi_rre_sp  | input     | 2     | Response                             |
| desc_axi_rla     | input     | 1     | Last beat of burst                   |

| Signal       | Direction | Width            | Description    |
|--------------|-----------|------------------|----------------|
| st           |           |                  |                |
| desc_axi_rid | input     | AXI_ID_WID<br>TH | Transaction ID |

#### Shared Data Read Interface (Per-Channel Arrays)

| Signal                        | Direction | Width                                | Description            |
|-------------------------------|-----------|--------------------------------------|------------------------|
| sched_rd_val_id[ch]           | output    | NUM_CHAN<br>NELS                     | Read request valid     |
| sched_rd_addr[r[ch]]          | output    | NUM_CHAN<br>NELS x<br>ADDR_WIDT<br>H | Source address         |
| sched_rd_beats[ch]            | output    | NUM_CHAN<br>NELS x 32                | Beats to read          |
| sched_rd_done_strobe[ch]      | input     | NUM_CHAN<br>NELS                     | Read completion strobe |
| sched_rd_beats_done[ch]       | input     | NUM_CHAN<br>NELS x 32                | Beats completed        |
| sched_rd_error_strobe[or[ch]] | input     | NUM_CHAN<br>NELS                     | Read error             |

#### Shared Data Write Interface (Per-Channel Arrays)

| Signal                   | Direction | Width                                | Description             |
|--------------------------|-----------|--------------------------------------|-------------------------|
| sched_wr_val_id[ch]      | output    | NUM_CHAN<br>NELS                     | Write request valid     |
| sched_wr_ready_dy[ch]    | input     | NUM_CHAN<br>NELS                     | Write ready             |
| sched_wr_addr[r[ch]]     | output    | NUM_CHAN<br>NELS x<br>ADDR_WIDT<br>H | Destination address     |
| sched_wr_beats[ch]       | output    | NUM_CHAN<br>NELS x 32                | Beats to write          |
| sched_wr_done_strobe[ch] | input     | NUM_CHAN<br>NELS                     | Write completion strobe |

| Signal                   | Direction | Width                 | Description     |
|--------------------------|-----------|-----------------------|-----------------|
| sched_wr_bea_ts_done[ch] | input     | NUM_CHAN<br>NELS x 32 | Beats completed |
| sched_wr_err or[ch]      | input     | NUM_CHAN<br>NELS      | Write error     |

## Unified Monitor Bus Interface

| Signal     | Direction | Width | Description          |
|------------|-----------|-------|----------------------|
| mon_valid  | output    | 1     | Monitor packet valid |
| mon_ready  | input     | 1     | Monitor packet ready |
| mon_packet | output    | 64    | Monitor packet data  |

## Operation

### Descriptor Fetch Arbitration

1. **Request Phase:** Channels assert desc\_ar\_valid[ch] when needing descriptor fetch
2. **Grant Phase:** Round-robin arbiter selects one channel
3. **AR Issue:** Granted channel's AR signals muxed to external interface
4. **ID Embedding:** Channel ID placed in lower bits of arid
5. **Grant ACK:** Arbiter acknowledged when arready received
6. **R Routing:** Response routed to correct channel based on rid[CHAN\_WIDTH-1:0]

### Data Path Flow

Data interfaces pass directly through without arbitration:

```

scheduler_group[ch].sched_rd_* -> sched_rd_*[ch] -> axi_read_engine
(handles arbitration)
scheduler_group[ch].sched_wr_* -> sched_wr_*[ch] -> axi_write_engine
(handles arbitration)

```

### MonBus Aggregation

9 sources aggregated via monbus\_arbiter: - Sources 0-7: Per-channel mon\_\* from scheduler\_groups - Source 8: Descriptor AXI master monitor

---

## Integration Example

```
scheduler_group_array #(
    .NUM_CHANNELS          (8),
    .ADDR_WIDTH             (64),
    .DATA_WIDTH              (512),
    .AXI_ID_WIDTH            (8)
) u_scheduler_group_array (
    .clk                    (clk),
    .rst_n                  (rst_n),

    // APB kick-off
    .apb_valid               (apb_valid),
    .apb_ready                (apb_ready),
    .apb_addr                 (apb_addr),

    // Configuration
    .cfg_channel_enable       (cfg_channel_enable),
    .cfg_channel_reset        (cfg_channel_reset),
    .cfg_sched_enable         (cfg_sched_enable),
    // ... other config

    // Descriptor AXI master
    .desc_axi_arvalid        (desc_axi_arvalid),
    .desc_axi_arready         (desc_axi_arready),
    // ... full AR/R interface

    // Data read interface (to axi_read_engine)
    .sched_rd_valid           (sched_rd_valid),
    .sched_rd_addr             (sched_rd_addr),
    .sched_rd_beats            (sched_rd_beats),
    .sched_rd_done_strobe      (rd_done_strobe),
    .sched_rd_beats_done       (rd_beats_done),

    // Data write interface (to axi_write_engine)
    .sched_wr_valid           (sched_wr_valid),
    .sched_wr_ready            (sched_wr_ready),
    .sched_wr_addr             (sched_wr_addr),
    .sched_wr_beats            (sched_wr_beats),
    .sched_wr_done_strobe      (wr_done_strobe),
    .sched_wr_beats_done       (wr_beats_done),

    // Monitor bus
    .mon_valid                (sga_mon_valid),
    .mon_ready                 (sga_mon_ready),
    .mon_packet                (sga_mon_packet)
);
```

---

## Related Documentation

- **Parent:** 01\_stream\_core.md - Top-level integration
  - **Child:** 03\_scheduler\_group.md - Single channel scheduler + descriptor engine
  - **Scheduler:** 04\_scheduler.md - Channel state machine
  - **Descriptor Engine:** 05\_descriptor\_engine.md - Descriptor fetch and parsing
  - **AXI Read Engine:** 06\_axi\_read\_engine.md - Read datapath (receives sched\_rd\_\*)
  - **AXI Write Engine:** 12\_axi\_write\_engine.md - Write datapath (receives sched\_wr\_\*)
  - **MonBus Arbiter:** 16\_monbus\_axil\_group.md - Monitor bus aggregation
- 

**Last Updated:** 2025-11-30 (verified against RTL implementation)

## Scheduler Group

**Module:** scheduler\_group.sv **Location:**

projects/components/stream/rtl/macro/ **Category:** MACRO (Channel Wrapper)

**Parent:** scheduler\_group\_array.sv **Status:** Implemented **Last Updated:** 2025-11-30

---

## Overview

The scheduler\_group module is a wrapper combining the scheduler and descriptor\_engine for a single STREAM channel. It provides a simplified interface compared to RAPIDS by removing unused features while maintaining the core descriptor-based DMA functionality.

## Key Features

- **Single Channel Wrapper:** Combines scheduler + descriptor\_engine
- **Simplified from RAPIDS:** No program engine, no control engines, no alignment bus
- **MonBus Aggregation:** Combines 2 internal sources (scheduler, descriptor\_engine)

- **Direct Data Interfaces:** Per-channel read/write interfaces to shared engines

## Simplified from RAPIDS

Removed from RAPIDS scheduler\_group: - Program engine - Control read/write engines - Alignment bus - Network interfaces (RDA, EOS completion) - Credit management

---

## Architecture

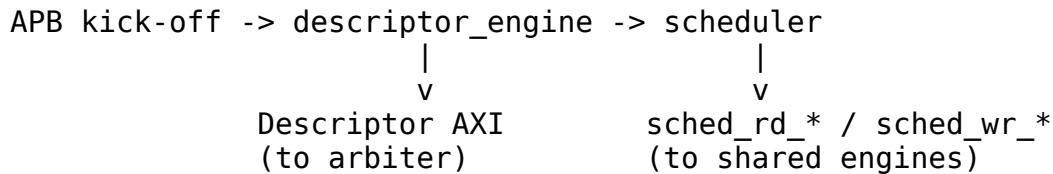
### Component Hierarchy

```

scheduler_group
  scheduler          # Channel state machine
  descriptor_engine # Descriptor fetch and parsing
  arbiter_round_robin_monbus # 2-source MonBus arbiter

```

### Data Flow



**Descriptor Flow:** 1. APB kick-off provides descriptor address 2. Descriptor engine fetches via shared AXI (arbitrated externally) 3. Parsed descriptor delivered to scheduler 4. Scheduler coordinates read/write phases

**Data Path Flow:** 1. Scheduler asserts sched\_rd\_valid with source address/beats 2. External read engine issues AXI reads, returns completion strobe 3. Scheduler asserts sched\_wr\_valid with destination address/beats 4. External write engine issues AXI writes, returns completion strobe 5. Scheduler advances to next descriptor or completes

---

## Parameters

| Parameter    | Type | Default | Description              |
|--------------|------|---------|--------------------------|
| CHANNEL_ID   | int  | 0       | Channel identifier (0-7) |
| NUM_CHANNELS | int  | 8       | Total number of channels |

| Parameter    | Type | Default               | Description              |
|--------------|------|-----------------------|--------------------------|
| CHAN_WIDTH   | int  | \$clog2(NUM_CHANNELS) | Channel ID width         |
| ADDR_WIDTH   | int  | 64                    | Address bus width        |
| DATA_WIDTH   | int  | 512                   | Data bus width           |
| AXI_ID_WIDTH | int  | 8                     | AXI transaction ID width |

## Monitor Bus Agent IDs

| Parameter          | Type  | Default | Description                |
|--------------------|-------|---------|----------------------------|
| DESC_MON_AGEN_T_ID | 8-bit | 0x10+ch | Descriptor engine agent ID |
| SCHED_MON_AGENT_ID | 8-bit | 0x30+ch | Scheduler agent ID         |
| MON_UNIT_ID        | 4-bit | 0x1     | Unit ID                    |
| MON_CHANNEL_ID     | 6-bit | ch      | Channel ID in packets      |

## Port List

### Clock and Reset

| Signal | Direction | Width | Description                   |
|--------|-----------|-------|-------------------------------|
| clk    | input     | 1     | System clock                  |
| rst_n  | input     | 1     | Active-low asynchronous reset |

### APB Programming Interface

| Signal    | Direction | Width | Description              |
|-----------|-----------|-------|--------------------------|
| apb_valid | input     | 1     | Descriptor address valid |
| apb_ready | output    | 1     | Ready to accept          |

| Signal   | Direction | Width      | Description        |
|----------|-----------|------------|--------------------|
|          |           |            | descriptor         |
| apb_addr | input     | ADDR_WIDTH | Descriptor address |

### Configuration Interface

| Signal                 | Direction | Width | Description                |
|------------------------|-----------|-------|----------------------------|
| cfg_channel_e<br>nable | input     | 1     | Enable this<br>channel     |
| cfg_channel_r<br>eset  | input     | 1     | Soft reset this<br>channel |

### Scheduler Configuration

| Signal                       | Direction | Width | Description                         |
|------------------------------|-----------|-------|-------------------------------------|
| cfg_sched_tim<br>eout_cycles | input     | 16    | Timeout<br>threshold                |
| cfg_sched_tim<br>eout_enable | input     | 1     | Enable timeout<br>detection         |
| cfg_sched_err<br>_enable     | input     | 1     | Enable error<br>reporting           |
| cfg_sched_com<br>pl_enable   | input     | 1     | Enable<br>completion<br>reporting   |
| cfg_sched_per<br>f_enable    | input     | 1     | Enable<br>performance<br>monitoring |

### Descriptor Engine Configuration

| Signal                      | Direction | Width          | Description                    |
|-----------------------------|-----------|----------------|--------------------------------|
| cfg_desceng_<br>prefetch    | input     | 1              | Enable descriptor<br>prefetch  |
| cfg_desceng_<br>fifo_thresh | input     | 4              | FIFO threshold for<br>prefetch |
| cfg_desceng_<br>addr0_base  | input     | ADDR_WIDT<br>H | Address range 0 base           |
| cfg_desceng_<br>addr0_limit | input     | ADDR_WIDT      | Address range 0 limit          |

| Signal                  | Direction | Width     | Description           |
|-------------------------|-----------|-----------|-----------------------|
|                         |           | H         |                       |
| cfg_desceng_addr1_base  | input     | ADDR_WIDT | Address range 1 base  |
| cfg_desceng_addr1_limit | input     | ADDR_WIDT | Address range 1 limit |
|                         |           | H         |                       |

### Status Interface

| Signal                 | Direction | Width | Description                   |
|------------------------|-----------|-------|-------------------------------|
| descriptor_engine_idle | output    | 1     | Descriptor engine idle        |
| scheduler_idle         | output    | 1     | Scheduler in IDLE state       |
| scheduler_state        | output    | 7     | Scheduler FSM state (ONE-HOT) |
| sched_error            | output    | 1     | Scheduler error (sticky)      |

### Descriptor AXI Interface (to external arbiter)

#### AR Channel:

| Signal        | Direction | Width        | Description           |
|---------------|-----------|--------------|-----------------------|
| desc_ar_valid | output    | 1            | Address valid         |
| desc_ar_ready | input     | 1            | Address ready         |
| desc_ar_addr  | output    | ADDR_WIDTH   | Address               |
| desc_ar_len   | output    | 8            | Burst length - 1      |
| desc_ar_size  | output    | 3            | Burst size            |
| desc_ar_burst | output    | 2            | Burst type            |
| desc_ar_id    | output    | AXI_ID_WIDTH | Transaction ID        |
| desc_ar_lock  | output    | 1            | Lock type             |
| desc_ar_cache | output    | 4            | Cache attributes      |
| desc_ar_prot  | output    | 3            | Protection attributes |

| Signal         | Direction | Width | Description       |
|----------------|-----------|-------|-------------------|
| desc_ar_qos    | output    | 4     | QoS value         |
| desc_ar_region | output    | 4     | Region identifier |

### R Channel (256-bit fixed):

| Signal       | Direction | Width        | Description                    |
|--------------|-----------|--------------|--------------------------------|
| desc_r_valid | input     | 1            | Read data valid                |
| desc_r_ready | output    | 1            | Read data ready                |
| desc_r_data  | input     | 256          | Read data (256-bit descriptor) |
| desc_r_resp  | input     | 2            | Response                       |
| desc_r_last  | input     | 1            | Last beat                      |
| desc_r_id    | input     | AXI_ID_WIDTH | Transaction ID                 |

### Data Read Interface (to shared read engine)

| Signal               | Direction | Width      | Description            |
|----------------------|-----------|------------|------------------------|
| sched_rd_valid       | output    | 1          | Read request valid     |
| sched_rd_addr        | output    | ADDR_WIDTH | Source address         |
| sched_rd_beats       | output    | 32         | Beats to read          |
| sched_rd_done_strobe | input     | 1          | Read completion strobe |
| sched_rd_beats_done  | input     | 32         | Beats completed        |
| sched_rd_error       | input     | 1          | Read error             |

### Data Write Interface (to shared write engine)

| Signal         | Direction | Width | Description         |
|----------------|-----------|-------|---------------------|
| sched_wr_valid | output    | 1     | Write request valid |

| Signal               | Direction | Width      | Description             |
|----------------------|-----------|------------|-------------------------|
| sched_wr_ready       | input     | 1          | Write ready             |
| sched_wr_addr        | output    | ADDR_WIDTH | Destination address     |
| sched_wr_beats       | output    | 32         | Beats to write          |
| sched_wr_done_strobe | input     | 1          | Write completion strobe |
| sched_wr_beats_done  | input     | 32         | Beats completed         |
| sched_wr_error       | input     | 1          | Write error             |

## Monitor Bus Interface

| Signal     | Direction | Width | Description          |
|------------|-----------|-------|----------------------|
| mon_valid  | output    | 1     | Monitor packet valid |
| mon_ready  | input     | 1     | Monitor packet ready |
| mon_packet | output    | 64    | Monitor packet data  |

## Operation

### Transfer Sequence

1. **APB Kick-off:** apb\_valid with descriptor address
2. **Descriptor Fetch:** descriptor\_engine fetches via AXI
3. **Descriptor Parse:** 256-bit descriptor parsed into fields
4. **Read Phase:** scheduler asserts sched\_rd\_valid
5. **Read Completion:** engine returns sched\_rd\_done\_strobe
6. **Write Phase:** scheduler asserts sched\_wr\_valid
7. **Write Completion:** engine returns sched\_wr\_done\_strobe
8. **Chain Check:** if next\_descriptor\_ptr != 0, fetch next
9. **Complete:** scheduler returns to IDLE

## MonBus Sources

The scheduler\_group aggregates 2 MonBus sources: 1. **Descriptor Engine:** Fetch events, errors 2. **Scheduler:** State transitions, completions, errors

---

## Integration Example

```
scheduler_group #(
    .CHANNEL_ID          (ch),
    .NUM_CHANNELS        (8),
    .ADDR_WIDTH          (64),
    .DATA_WIDTH          (512),
    .AXI_ID_WIDTH        (8),
    .DESC_MON_AGENT_ID   (8'h10 + ch),
    .SCHED_MON_AGENT_ID  (8'h30 + ch)
) u_scheduler_group (
    .clk                 (clk),
    .rst_n               (rst_n),
    // APB interface
    .apb_valid           (apb_valid[ch]),
    .apb_ready           (apb_ready[ch]),
    .apb_addr            (apb_addr[ch]),
    // Configuration
    .cfg_channel_enable  (cfg_channel_enable[ch]),
    .cfg_channel_reset   (cfg_channel_reset[ch]),
    // ... other config
    // Descriptor AXI (to arbiter)
    .desc_ar_valid       (desc_ar_valid[ch]),
    .desc_ar_ready       (desc_ar_ready[ch]),
    // ... full AR/R interface
    // Data interfaces (direct passthrough)
    .sched_rd_valid      (sched_rd_valid[ch]),
    .sched_rd_addr       (sched_rd_addr[ch]),
    .sched_rd_beats      (sched_rd_beats[ch]),
    .sched_rd_done_strobe (sched_rd_done_strobe[ch]),
    .sched_rd_beats_done (sched_rd_beats_done[ch]),
    .sched_wr_valid      (sched_wr_valid[ch]),
    .sched_wr_ready      (sched_wr_ready[ch]),
    .sched_wr_addr       (sched_wr_addr[ch]),
    .sched_wr_beats      (sched_wr_beats[ch]),
    .sched_wr_done_strobe (sched_wr_done_strobe[ch]),
    .sched_wr_beats_done (sched_wr_beats_done[ch]),
```

```
// Monitor bus
    .mon_valid          (mon_valid_ch[ch]),
    .mon_ready          (mon_ready_ch[ch]),
    .mon_packet         (mon_packet_ch[ch])
);
```

---

## Related Documentation

- **Parent:** 02\_scheduler\_group\_array.md - Multi-channel array
  - **Scheduler:** 04\_scheduler.md - Channel state machine details
  - **Descriptor Engine:** 05\_descriptor\_engine.md - Descriptor fetch logic
  - **AXI Read Engine:** 06\_axi\_read\_engine.md - Shared read engine
  - **AXI Write Engine:** 12\_axi\_write\_engine.md - Shared write engine
- 

**Last Updated:** 2025-11-30 (verified against RTL implementation)

## Scheduler Specification

**Module:** scheduler.sv **Location:** projects/components/stream/rtl/fub/  
**Status:** Implemented **Last Updated:** 2025-11-30

---

## Overview

The Scheduler coordinates descriptor-based memory-to-memory DMA transfers for a single channel. It receives descriptors, manages concurrent read/write operations, and handles descriptor chaining.

## Key Features

- **Concurrent read/write:** Read and write engines run simultaneously (prevents deadlock)
- **Beat-based tracking:** Length in data width units (STREAM simplification)
- **Aligned addresses:** No alignment fixup logic (must be pre-aligned)
- **Descriptor chaining:** Follows next\_descriptor\_ptr for multi-buffer transfers
- **Interrupt generation:** MonBus IRQ event when gen\_irq flag set
- **Error handling:** Timeout detection, error aggregation from engines

- **MonBus integration:** State transition and IRQ event reporting

## Block Diagram



*Diagram*

**Source:** [02\\_scheduler\\_block.mmd](#)

---

## CRITICAL: Concurrent Read/Write Design

### Why Concurrent Operation is Essential:

The scheduler runs read and write engines **CONCURRENTLY** in CH\_XFER\_DATA state. This prevents deadlock when transfer size exceeds SRAM buffer capacity:

Example: 100MB transfer with 2KB SRAM buffer

Sequential operation (WRONG):

1. Read 100MB → DEADLOCK at 2KB (SRAM full, can't complete read)

Concurrent operation (CORRECT):

1. Read starts filling SRAM → SRAM becomes full (2KB)
2. Read pauses (natural backpressure)
3. Write drains SRAM → SRAM has free space
4. Read resumes → Both continue until 100MB complete

**Implementation:** - Both sched\_rd\_valid and sched\_wr\_valid asserted in CH\_XFER\_DATA - Independent beat counters: r\_read\_beats\_remaining, r\_write\_beats\_remaining - Exit when **BOTH** counters reach zero

---

## Parameters

```
parameter int CHANNEL_ID = 0;          // Channel identifier
parameter int NUM_CHANNELS = 8;         // Total channels in
system
parameter int CHAN_WIDTH = $clog2(NUM_CHANNELS); // Channel ID width
parameter int ADDR_WIDTH = 64;          // Address bus width
parameter int DATA_WIDTH = 512;          // Data bus width
(beats)

// Monitor Bus Parameters
parameter logic [7:0] MON_AGENT_ID = 8'h40;      // STREAM Scheduler
Agent ID
parameter logic [3:0] MON_UNIT_ID = 4'h1;           // Unit identifier
parameter logic [5:0] MON_CHANNEL_ID = 6'h0;        // Base channel ID

// Descriptor Width (FIXED at 256-bit for STREAM)
parameter int DESC_WIDTH = 256;
```

## Validation:

```
// Scheduler only supports 256-bit STREAM descriptors
if (DESC_WIDTH != 256)
    $fatal("DESC_WIDTH must be 256 for STREAM scheduler");
```

---

## Port List

### Clock and Reset

| Signal | Direction | Width | Description                         |
|--------|-----------|-------|-------------------------------------|
| clk    | input     | 1     | System clock                        |
| rst_n  | input     | 1     | Active-low<br>asynchronous<br>reset |

### Configuration Interface

| Signal                   | Direction | Width | Description                                              |
|--------------------------|-----------|-------|----------------------------------------------------------|
| cfg_channel_enable       | input     | 1     | Enable this channel                                      |
| cfg_channel_reset        | input     | 1     | Channel soft reset (FSM<br>→ IDLE)                       |
| cfg_sched_timeout_cycles | input     | 16    | Timeout threshold in<br>clock cycles (runtime<br>config) |

| Signal                   | Direction | Width | Description              |
|--------------------------|-----------|-------|--------------------------|
| cfg_sched_timeout_enable | input     | 1     | Enable timeout detection |

### Status Interface

| Signal          | Direction | Width | Description                          |
|-----------------|-----------|-------|--------------------------------------|
| scheduler_idle  | output    | 1     | Scheduler idle flag                  |
| scheduler_state | output    | 7     | Current FSM state (one-hot encoding) |

### Descriptor Engine Interface

| Signal            | Direction | Width | Description                                  |
|-------------------|-----------|-------|----------------------------------------------|
| descriptor_valid  | input     | 1     | Descriptor valid from descriptor engine      |
| descriptor_ready  | output    | 1     | Scheduler ready to accept descriptor         |
| descriptor_packet | input     | 256   | 256-bit STREAM descriptor (see format below) |
| descriptor_error  | input     | 1     | Error signal from descriptor engine          |

### Data Read Interface

#### To AXI Read Engine:

| Signal          | Direction | Width      | Description                                   |
|-----------------|-----------|------------|-----------------------------------------------|
| sched_rd_val_id | output    | 1          | Channel requests read                         |
| sched_rd_addr   | output    | ADDR_WIDTH | Source address (aligned, static during burst) |
| sched_rd_beats  | output    | 32         | Beats remaining to read                       |

#### Completion from Read Engine:

| Signal               | Direction | Width | Description                          |
|----------------------|-----------|-------|--------------------------------------|
| sched_rd_done_strobe | input     | 1     | Read burst completed (1-cycle pulse) |

| Signal               | Direction | Width | Description                        |
|----------------------|-----------|-------|------------------------------------|
| sched_rd_bea_ts_done | input     | 32    | Number of beats completed in burst |
| sched_rd_err or      | input     | 1     | Read engine error (sticky)         |

## Data Write Interface

### To AXI Write Engine:

| Signal          | Direction | Width      | Description                                        |
|-----------------|-----------|------------|----------------------------------------------------|
| sched_wr_val_id | output    | 1          | Channel requests write                             |
| sched_wr_ready  | input     | 1          | Engine ready for channel (completion handshake)    |
| sched_wr_addr   | output    | ADDR_WIDTH | Destination address (aligned, static during burst) |
| sched_wr_beats  | output    | 32         | Beats remaining to write                           |

### Completion from Write Engine:

| Signal               | Direction | Width | Description                           |
|----------------------|-----------|-------|---------------------------------------|
| sched_wr_done_strobe | input     | 1     | Write burst completed (1-cycle pulse) |
| sched_wr_beats_done  | input     | 32    | Number of beats completed in burst    |
| sched_wr_error or    | input     | 1     | Write engine error (sticky)           |

## Error Signals

| Signal      | Direction | Width | Description                                              |
|-------------|-----------|-------|----------------------------------------------------------|
| sched_error | output    | 1     | Scheduler error output (aggregates rd/wr errors, sticky) |

## Monitor Bus Interface

| Signal     | Direction | Width | Description               |
|------------|-----------|-------|---------------------------|
| mon_valid  | output    | 1     | Monitor packet valid      |
| mon_ready  | input     | 1     | Monitor bus ready         |
| mon_packet | output    | 64    | 64-bit monitor bus packet |

## Interface

### Clock and Reset

```
input logic clk;
input logic rst_n; // Active-low
asynchronous reset
```

### Configuration Interface

```
input logic cfg_channel_enable; // 
Enable this channel
input logic cfg_channel_reset; // 
Channel reset (soft reset)
input logic [15:0] cfg_sched_timeout_cycles; //
Timeout threshold (runtime config)
input logic cfg_sched_timeout_enable; // 
Enable timeout detection
```

**Channel Reset Behavior:** - cfg\_channel\_reset forces FSM to CH\_IDLE  
immediately - Clears descriptor\_loaded flag - Resets beat counters - Independent  
of global rst\_n

**Timeout Configuration:** - Runtime configurable via cfg\_sched\_timeout\_cycles  
(replaces compile-time TIMEOUT\_CYCLES parameter) - Can be disabled/enabled  
dynamically via cfg\_sched\_timeout\_enable

### Status Interface

```
output logic scheduler_idle; // 
Scheduler in CH_IDLE
output logic [6:0] scheduler_state; // Current
state (ONE-HOT)
```

**State Encoding (ONE-HOT):** - [0] = CH\_IDLE - [1] = CH\_FETCH\_DESC - [2] =  
CH\_XFER\_DATA - [3] = CH\_COMPLETE - [4] = CH\_NEXT\_DESC - [5] = CH\_ERROR -  
[6] = Reserved

## Descriptor Engine Interface

```
input logic descriptor_valid;
output logic descriptor_ready;
input logic [DESC_WIDTH-1:0] descriptor_packet; // 256-bit
STREAM descriptor
input logic descriptor_error; // Error
from descriptor engine
```

**Descriptor Handshake:** - descriptor\_ready asserted in CH\_IDLE or CH\_NEXT\_DESC - Descriptor captured when valid && ready - Supports descriptor chaining (next\_descriptor\_ptr)

## Data Read Interface (to AXI Read Engine)

### Request:

```
output logic sched_rd_valid; // Request
read access
output logic [ADDR_WIDTH-1:0] sched_rd_addr; // Source
address (static base)
output logic [31:0] sched_rd_beats; // Total
beats to read
```

### Completion:

```
input logic sched_rd_done_strobe; // Read
engine completed beats
input logic [31:0] sched_rd_beats_done; // Number
of beats completed
```

### Error:

```
input logic sched_rd_error; // Read
engine error
```

**Address Management:** - Scheduler provides **static base address** in sched\_rd\_addr - Read engine handles address increment internally - Scheduler does NOT update sched\_rd\_addr after each burst

**Interface Timing Notes (November 2025 Updates):** - Scheduler holds sched\_rd\_valid high while sched\_rd\_beats > 0 - Engine reports completion via sched\_rd\_done\_strobe (pulsed) - Scheduler decrements r\_read\_beats\_remaining by sched\_rd\_beats\_done - Process repeats until r\_read\_beats\_remaining == 0

## Data Write Interface (to AXI Write Engine)

### Request:

```

output logic           sched_wr_valid;          // Request
write access          sched_wr_ready;         // Engine
input logic            sched_wr_addr;          //
grants access          sched_wr_beats;         // Total
output logic [ADDR_WIDTH-1:0] Destination address (static base)
Destination address (static base)
output logic [31:0]      sched_wr_beats;
beats to write

```

### Completion:

```

input logic           sched_wr_done_strobe; // Write
engine completed beats
input logic [31:0]      sched_wr_beats_done;   // Number
of beats completed

```

### Error:

```

input logic           sched_wr_error;        // Write
engine error

```

### Interface Timing Notes (November 2025 Updates):

**sched\_wr\_ready Timing:** - REGISTERED OUTPUT from write engine - Not combinatorial from completion signals - Asserts 1 cycle after channel becomes ready to accept new request - Cleared 1 cycle after handshake (valid && ready) - **Do not expect immediate deassertion on handshake** - takes 1 cycle

### Example Timing:

| Cycle | sched_wr_valid | sched_wr_ready | sched_wr_done_strobe | Notes                             |
|-------|----------------|----------------|----------------------|-----------------------------------|
| 0     | 0              | 0              | 0                    | Idle                              |
| 1     | 1              | 0              | 0                    | Request<br>(not ready yet)        |
| 2     | 1              | 1              | 0                    | Handshake!                        |
| 3     | 1              | 0              | 0                    | Ready<br>cleared (registered)     |
| ...   |                |                |                      | (W-phase executes)                |
| 50    | 1              | 0              | 1                    | B<br>response, done strobe        |
| 51    | 1              | 1              | 0                    | Ready re-<br>asserts (next cycle) |

### Monitor Bus Interface

```

output logic           mon_valid;
input logic            mon_ready;
output logic [63:0]      mon_packet;

```

**MonBus Events Generated:** - State transitions (IDLE → FETCH\_DESC, etc.) - IRQ event (when descriptor.gen\_irq set) - Error events

---

## Descriptor Format

### STREAM Descriptor (256-bit)

```
typedef struct packed {
    logic [47:0] reserved;           // [255:208] Reserved
    logic [7:0] desc_priority;       // [207:200] Transfer
priority
    logic [3:0] channel_id;          // [199:196] Channel ID
(informational)
    logic         error;             // [195] Error flag
    logic         last;              // [194] Last in chain flag
    logic         gen_irq;            // [193] Generate interrupt
on completion
    logic         valid;              // [192] Valid descriptor
    logic [31:0] next_descriptor_ptr; // [191:160] Next descriptor
address (0 = last)
    logic [31:0] length;             // [159:128] Length in BEATS
    logic [63:0] dst_addr;            // [127:64] Destination
address (aligned)
    logic [63:0] src_addr;            // [63:0] Source address
(aligned)
} descriptor_t;
```

**Field Constraints:** - src\_addr / dst\_addr: Must be aligned to DATA\_WIDTH (e.g., 64-byte aligned for 512-bit data) - length: Transfer size in BEATS (not bytes or chunks) - next\_descriptor\_ptr: 0 or address of next descriptor - valid: Must be 1 for descriptor to be accepted - last: Terminates chain (overrides next\_descriptor\_ptr) - gen\_irq: Generates STREAM\_EVENT\_IRQ via MonBus when transfer completes

### Descriptor Bit Positions:

|                 |           |
|-----------------|-----------|
| DESC_SRC_ADDR:  | [63:0]    |
| DESC_DST_ADDR:  | [127:64]  |
| DESC_LENGTH:    | [159:128] |
| DESC_NEXT_PTR:  | [191:160] |
| DESC_VALID_BIT: | [192]     |
| DESC_GEN_IRQ:   | [193]     |
| DESC_LAST:      | [194]     |

---

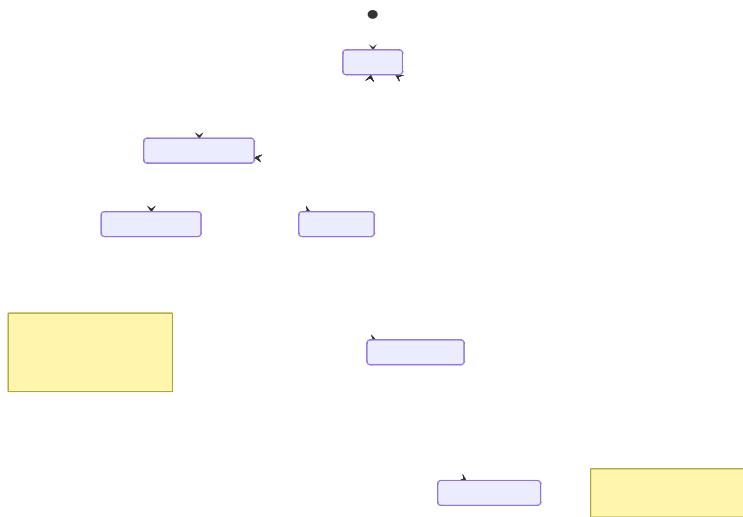
# FSM Operation

## State Machine

### States (ONE-HOT encoded):

|               |                                  |
|---------------|----------------------------------|
| CH_IDLE       | - Waiting for descriptor         |
| CH_FETCH_DESC | - Latch and validate descriptor  |
| CH_XFER_DATA  | - Concurrent read/write transfer |
| CH_COMPLETE   | - Transfer done, check chaining  |
| CH_NEXT_DESC  | - Fetch next chained descriptor  |
| CH_ERROR      | - Error condition                |

### FSM Flow:



### Diagram

Source: [02\\_scheduler\\_block.mmd](#)

## State Transitions

**CH\_IDLE:** - Wait for: descriptor\_valid && cfg\_channel\_enable - Action: Assert descriptor\_ready - Next: CH\_FETCH\_DESC (when handshake occurs)

**CH\_FETCH\_DESC:** - Action: - Latch descriptor fields into r\_descriptor - Initialize working registers (r\_src\_addr, r\_dst\_addr, r\_\*\_beats\_remaining) - Validate descriptor.valid bit - Next: - CH\_XFER\_DATA (if valid) - CH\_ERROR (if invalid)

**CH\_XFER\_DATA:** - Action: - Assert BOTH sched\_rd\_valid and sched\_wr\_valid (concurrent operation!) - Decrement r\_read\_beats\_remaining on sched\_rd\_done\_strobe - Decrement r\_write\_beats\_remaining on sched\_wr\_done\_strobe - Monitor timeout counter - Exit When:

```
r_read_beats_remaining == 0 && r_write_beats_remaining == 0 - Next:  
CH_COMPLETE
```

**CH\_COMPLETE:** - **Action:** - Generate MonBus IRQ event (if gen\_irq set) - Check next\_descriptor\_ptr and last flag - Clear descriptor\_loaded flag - **Next:** - CH\_NEXT\_DESC (if chaining) - CH\_IDLE (if last or no chain)

**CH\_NEXT\_DESC:** - **Wait for:** descriptor\_valid (descriptor engine fetches next) - **Action:** Assert descriptor\_ready - **Next:** CH\_FETCH\_DESC

**CH\_ERROR:** - **STICKY STATE:** Once in error, stays here until channel reset - **Action:** Report error via MonBus - **Recovery:** Requires cfg\_channel\_reset assertion (or global reset) - **Note:** scheduler\_idle asserts in this state (allows external monitoring)

---

## Beat Tracking

### Independent Counters

#### Initialization (CH\_FETCH\_DESC):

```
r_read_beats_remaining <= r_descriptor.length;  
r_write_beats_remaining <= r_descriptor.length;
```

#### Decrement (CH\_XFER\_DATA):

```
// Read progress (independent)  
if (sched_rd_done_strobe) begin  
    r_read_beats_remaining <= (r_read_beats_remaining >=  
    sched_rd_beats_done) ?  
        (r_read_beats_remaining -  
        sched_rd_beats_done) : 32'h0;  
end  
  
// Write progress (independent)  
if (sched_wr_done_strobe) begin  
    r_write_beats_remaining <= (r_write_beats_remaining >=  
    sched_wr_beats_done) ?  
        (r_write_beats_remaining -  
        sched_wr_beats_done) : 32'h0;  
end
```

**Saturation:** - Counters saturate at 0 (prevent underflow) - Safety check for engine misbehavior

## Completion Detection

### Combinational Flags:

```
w_read_complete      = (r_read_beats_remaining == 0);  
w_write_complete    = (r_write_beats_remaining == 0);  
w_transfer_complete = w_read_complete && w_write_complete;
```

### State Exit:

```
// In CH_XFER_DATA:  
if (w_transfer_complete) begin  
    w_next_state = CH_COMPLETE;  
end
```

## Multiple Requests per Descriptor

For large transfers, scheduler issues multiple requests as engines complete work:

Descriptor: length = 256 beats (16KB @ 512-bit data)  
Engine burst size: 16 beats (configured via cfg\_axi\_wr\_xfer\_beats)

Request sequence:

Request 1: sched\_wr\_valid=1, sched\_wr\_beats=256, ready → handshake  
Engine executes 16-beat burst  
sched\_wr\_done\_strobe=1, sched\_wr\_beats\_done=16  
Scheduler updates: 256 - 16 = 240 beats remaining

Request 2: sched\_wr\_valid=1, sched\_wr\_beats=240, ready → handshake  
Engine executes 16-beat burst  
sched\_wr\_done\_strobe=1, sched\_wr\_beats\_done=16  
Scheduler updates: 240 - 16 = 224 beats remaining

... (continues for 16 requests total)

Request 16: sched\_wr\_valid=1, sched\_wr\_beats=16, ready → handshake  
Engine executes 16-beat burst  
sched\_wr\_done\_strobe=1, sched\_wr\_beats\_done=16  
Scheduler updates: 16 - 16 = 0 beats remaining →

COMPLETE

**Key Point:** Scheduler holds sched\_wr\_valid high and keeps reissuing requests as long as sched\_wr\_beats > 0. Engines handle each request independently.

---

## Address Management

### Static Base Address

#### Scheduler Provides:

```
assign sched_rd_addr = r_src_addr; // Static base, set in  
CH_FETCH_DESC  
assign sched_wr_addr = r_dst_addr; // Static base, set in  
CH_FETCH_DESC
```

**Scheduler Does NOT:** - Increment addresses after each burst - Calculate byte offsets - Handle alignment

**Engine Responsibility:** - Read engine: m\_axi\_araddr = sched\_rd\_addr +  
(beats\_issued << AXSIZE) - Write engine: m\_axi\_awaddr = sched\_wr\_addr +  
(beats\_issued << AXSIZE)

---

## Timeout Detection

### Timeout Counter

**Configuration:** - Runtime configurable via cfg\_sched\_timeout\_cycles (16-bit) - Can be enabled/disabled via cfg\_sched\_timeout\_enable - Replaces compile-time TIMEOUT\_CYCLES parameter

#### Increment:

```
// In CH_XFER_DATA when waiting for engines  
if (cfg_sched_timeout_enable && (!sched_rd_ready || !sched_wr_ready))  
begin  
    r_timeout_counter <= r_timeout_counter + 1;  
end
```

#### Timeout Flag:

```
assign w_timeout_expired = (r_timeout_counter >=  
cfg_sched_timeout_cycles);
```

#### Reset:

```
// Clear when state changes or engines respond  
if (state_change || (sched_rd_ready && sched_wr_ready)) begin  
    r_timeout_counter <= 0;  
end
```

**Action on Timeout:** - FSM transitions to CH\_ERROR - MonBus timeout event generated - Channel must be reset to recover

---

## Error Handling

### Error Sources

**External:** - descriptor\_error - Descriptor engine reports error - sched\_rd\_error - Read engine error (AXI RRESP != OKAY, etc.) - sched\_wr\_error - Write engine error (AXI BRESP != OKAY, etc.)

**Internal:** - w\_timeout\_expired - Timeout counter exceeded threshold - !r\_descriptor.valid - Invalid descriptor in CH\_FETCH\_DESC

### Sticky Error Flags

```
logic r_read_error_sticky; // Set on sched_rd_error, cleared in  
CH_IDLE  
logic r_write_error_sticky; // Set on sched_wr_error, cleared in  
CH_IDLE  
logic r_descriptor_error; // Set on descriptor_error or validation  
failure
```

### Set Condition:

```
if (sched_rd_error)  
    r_read_error_sticky <= 1'b1;  
  
if (sched_wr_error)  
    r_write_error_sticky <= 1'b1;
```

### Clear Condition:

```
if (r_current_state == CH_IDLE)  
    r_*_error_sticky <= 1'b0;
```

## Error Recovery

### Error Transition:

```
// Any state with error condition  
if (descriptor_error || sched_rd_error || sched_wr_error ||  
    r_read_error_sticky || r_write_error_sticky || w_timeout_expired)  
begin  
    w_next_state = CH_ERROR;  
end
```

### CH\_ERROR is STICKY:

```

CH_ERROR: begin
    // Error state - STICKY, stay here until reset
    // Once in error, only way out is through reset
    w_next_state = CH_ERROR;
end

```

**Recovery:** - CH\_ERROR is a **sticky state** - does NOT auto-recover - Software **must** assert cfg\_channel\_reset (or global rst\_n) - On channel reset: FSM → CH\_IDLE, sticky flags cleared

---

## Interrupt Generation

### IRQ via MonBus

**Trigger:** - Descriptor completes (CH\_COMPLETE state) - r\_descriptor.gen\_irq flag set

#### MonBus Event:

```

// In CH_COMPLETE state with gen_irq set
mon_packet = {
    MON_AGENT_ID,           // [63:56] Agent ID (0x40 = STREAM
Scheduler)
    MON_UNIT_ID,            // [55:52] Unit ID
    MON_CHANNEL_ID,          // [51:46] Channel ID
    STREAM_EVENT_IRQ,        // [45:40] Event code (IRQ)
    descriptor_fields        // [39:0] Descriptor info
};

```

**No Separate IRQ Signal:** - IRQ communicated via MonBus only - Software monitors MonBus for IRQ events - Event includes channel ID for routing

---

## Descriptor Chaining

### Chain Detection

#### In CH\_COMPLETE:

```

if (r_descriptor.next_descriptor_ptr != 32'h0 && !r_descriptor.last)
begin
    w_next_state = CH_NEXT_DESC; // Chain to next descriptor
end else begin
    w_next_state = CH_IDLE;      // Complete (last or no chain)
end

```

## Chain Termination

**Explicit Termination:** - next\_descriptor\_ptr == 0 → Stop - last == 1 → Stop  
(overrides next\_descriptor\_ptr)

### Example Chain:

Descriptor 0 @ 0x1000:  
src\_addr = 0x2000, dst\_addr = 0x3000, length = 64  
next\_descriptor\_ptr = 0x1040, last = 0  
→ Chains to next

Descriptor 1 @ 0x1040:  
src\_addr = 0x2100, dst\_addr = 0x3100, length = 32  
next\_descriptor\_ptr = 0x0000, last = 1  
→ Last in chain

---

## MonBus Integration

### Event Types

**State Transitions:** - IDLE → FETCH\_DESC: Descriptor fetch start - FETCH\_DESC → XFER\_DATA: Transfer start - XFER\_DATA → COMPLETE: Transfer complete - COMPLETE → NEXT\_DESC: Chain fetch - Any → ERROR: Error occurred

**Special Events:** - STREAM\_EVENT\_IRQ: Interrupt generation (gen\_irq flag) - STREAM\_EVENT\_TIMEOUT: Timeout expired - STREAM\_EVENT\_ERROR: Error condition

### MonBus Packet Format

#### Generic Event:

[63:56] - MON\_AGENT\_ID (0x40)  
[55:52] - MON\_UNIT\_ID  
[51:46] - MON\_CHANNEL\_ID + CHANNEL\_ID  
[45:40] - Event code  
[39:0] - Event-specific data

---

## Timing Diagrams

### Normal Transfer (No Chaining)

*Scheduler Normal Transfer Timing*

**Source:** [04\\_scheduler\\_normal\\_transfer.mmd](#)

**Notes:** - Both read and write run concurrently in XFER state - Independent done strobes decrement separate counters

## Descriptor Chaining

*Scheduler Descriptor Chaining Timing*

Source: [04\\_scheduler\\_chaining.mmd](#)

### Single-Descriptor Transfer (Detailed)

| Cycle                                 | State                                  | sched_wr_valid | sched_wr_ready | sched_wr_beats | Notes           |
|---------------------------------------|----------------------------------------|----------------|----------------|----------------|-----------------|
| 0                                     | IDLE                                   | 0              | 0              | 0              |                 |
| Waiting                               |                                        |                |                |                |                 |
| 1                                     | FETCH_DESC                             | 0              | 0              | 0              |                 |
| Request descriptor                    |                                        |                |                |                |                 |
| 2                                     | FETCH_DESC                             | 0              | 0              | 0              | (fetch latency) |
| 3                                     | XFER_DATA                              | 0              | 0              | 0              |                 |
| Descriptor received                   |                                        |                |                |                |                 |
| ...                                   | (read phase)                           |                |                |                |                 |
| 50                                    | XFER_DATA                              | 1              | 0              | 256            |                 |
| Assert write request                  |                                        |                |                |                |                 |
| 51                                    | XFER_DATA                              | 1              | 1              | 256            |                 |
| Engine ready, handshake!              |                                        |                |                |                |                 |
| 52                                    | XFER_DATA                              | 1              | 0              | 256            |                 |
| Ready cleared (registered)            |                                        |                |                |                |                 |
| ...                                   | (engine executes burst)                |                |                |                |                 |
| 100                                   | XFER_DATA                              | 1              | 0              | 256            |                 |
| done_strobe=1, beats_done=16          |                                        |                |                |                |                 |
| 101                                   | XFER_DATA                              | 1              | 1              | 240            |                 |
| Ready re-asserts, new beats_remaining |                                        |                |                |                |                 |
| 102                                   | XFER_DATA                              | 1              | 0              | 240            |                 |
| Handshake again                       |                                        |                |                |                |                 |
| ...                                   | (continues until beats_remaining == 0) |                |                |                |                 |

### Chained Descriptors (Detailed)

Descriptor chain:

```
Desc 0: length=128, next_ptr=0x1000_0100
Desc 1: length=64, next_ptr=0
```

| Cycle | State                 | Descriptor  | sched_wr_beats | Notes                  |
|-------|-----------------------|-------------|----------------|------------------------|
| 0     | IDLE                  | -           | 0              | Start                  |
| 1     | FETCH_DESC            | Req @0x1000 | 0              | Fetch first            |
| 3     | XFER_DATA             | Desc 0      | 128            | Transfer desc 0        |
| ...   | (8 bursts × 16 beats) |             |                |                        |
| 200   | COMPLETE              | Desc 0      | 0              | Transfer done          |
| 201   | NEXT_DESC             | Check ptr   | 0              | next_ptr = 0x1000_0100 |
| 202   | FETCH_DESC            | Req @0x1100 | 0              | Fetch second           |
| 204   | XFER_DATA             | Desc 1      | 64             | Transfer desc 1        |
| ...   | (4 bursts × 16 beats) |             |                |                        |
| 400   | COMPLETE              | Desc 1      | 0              | Transfer done          |
| 401   | NEXT_DESC             | Check ptr   | 0              | next_ptr = 0 →         |

|           |      |   |   |                |
|-----------|------|---|---|----------------|
| terminate |      |   |   |                |
| 402       | IDLE | - | 0 | Chain complete |

---

## Testing

**Test Location:** projects/components/stream/dv/tests/fub\_tests/scheduler/

### Key Test Scenarios:

1. **Single descriptor transfer** - Basic operation
  2. **Descriptor chaining** - 2-4 descriptors linked
  3. **Concurrent read/write** - Verify no deadlock with small SRAM
  4. **Large transfer (> SRAM)** - 100MB transfer with 2KB SRAM
  5. **IRQ generation** - gen\_irq flag set
  6. **Error handling** - Descriptor, read, write errors
  7. **Timeout detection** - Engine stall scenarios
  8. **Channel reset** - cfg\_channel\_reset during transfer
  9. **Runtime timeout config** - Change cfg\_sched\_timeout\_cycles dynamically
  10. **sched\_rd\_ready / sched\_wr\_ready timing validation** - Verify registered ready behavior
- 

## Performance Considerations

### Concurrent Operation Benefit

**Without Concurrency (Sequential):** - Max transfer size = SRAM buffer size - Deadlock when transfer > buffer - Throughput = min(read\_bw, write\_bw)

**With Concurrency:** - No transfer size limit - Natural flow control via SRAM full/empty - Throughput = max(read\_bw, write\_bw) (pipeline overlap)

### Example Performance

**Configuration:** - DATA\_WIDTH = 512 bits (64 bytes/beat) - SRAM = 2KB (32 beats) - Transfer = 100MB (1,562,500 beats)

**Sequential (hypothetical):** - DEADLOCK at 2KB (can't complete read)

**Concurrent:** - Read fills SRAM (32 beats) - Write drains SRAM concurrently - Both engines sustain ~0.9 beats/cycle - Total time: ~1.7M cycles

---

## Related Documentation

- **Descriptor Engine:** 05\_descriptor\_engine.md - Descriptor fetch
  - **AXI Read Engine:** 06\_axi\_read\_engine.md - Source data read
  - **AXI Write Engine:** 12\_axi\_write\_engine.md - Destination data write
  - **SRAM Controller:** 08\_sram\_controller.md - Buffer management
  - **Scheduler Group:** 03\_scheduler\_group.md - Single-channel integration
  - **Scheduler Group Array:** 02\_scheduler\_group\_array.md - Multi-channel integration
- 

## Revision History

| Date       | Version | Changes                                                                                                                                                                                                                                                                                                                                                                                                                                                                          |
|------------|---------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 2025-10-17 | 1.0     | Initial documentation with old signal names (datard_, datawr_)                                                                                                                                                                                                                                                                                                                                                                                                                   |
| 2025-11-16 | 1.5     | Enhanced documentation with detailed sections                                                                                                                                                                                                                                                                                                                                                                                                                                    |
| 2025-11-21 | 2.0     | <b>Merged documentation:-</b><br>Updated all signal names (sched_rd_, sched_wr_)-<br>Added runtime timeout configuration<br>(cfg_sched_timeout_cycles/enable)- Registered ready signal timing clarification-<br>Added multiple requests per descriptor section-<br>Enhanced beat tracking and error handling details-<br>Updated all code examples and timing diagrams- Added timing examples for chained descriptors- Combined best content from multiple documentation sources |
| 2025-11-30 | 2.1     | <b>RTL Sync Update:-</b>                                                                                                                                                                                                                                                                                                                                                                                                                                                         |

| Date | Version | Changes                                                                                                                                |
|------|---------|----------------------------------------------------------------------------------------------------------------------------------------|
|      |         | CH_ERROR is now STICKY (requires reset to recover)- scheduler_idle asserts in CH_ERROR state- Updated related documentation references |

**Last Updated:** 2025-11-30 (matched to current RTL implementation)

## Descriptor Engine

**Module:** descriptor\_engine.sv **Location:**

projects/components/stream/rtl/fub/ **Category:** FUB (Functional Unit Block)

**Parent:** scheduler\_group.sv **Status:** Implemented **Last Updated:** 2025-11-30

---

### Overview

The descriptor\_engine module is an autonomous descriptor fetch engine with chaining support. It fetches 256-bit STREAM descriptors from memory via AXI4 master interface and provides two operating modes for kick-off.

### Key Features

- **Two Operating Modes:**
    - APB-initiated: Software writes descriptor address, engine fetches
    - Autonomous chaining: Engine automatically fetches next\_descriptor\_ptr
  - **Two-FIFO Architecture:**
    - Descriptor address FIFO: Holds addresses to fetch
    - Descriptor data FIFO: Buffers fetched descriptors for scheduler
  - **Address Range Validation:** Two configurable ranges for security
  - **MonBus Event Reporting:** Fetch complete and error events
-

## Architecture

### Operating Flow

#### APB Mode:

APB write -> skid buffer -> desc addr FIFO -> AXI fetch -> desc data FIFO -> scheduler

#### Chaining Mode:

Descriptor complete -> extract next\_ptr -> validate -> desc addr FIFO  
->  
AXI fetch -> desc data FIFO -> scheduler (repeat until last=1 or  
next\_ptr=0)

#### FSM States

```
typedef enum logic [2:0] {
    RD_IDLE      = 3'b000, // Waiting for descriptor address
    RD_ISSUE_ADDR = 3'b001, // Issue AXI AR transaction
    RD_WAIT_DATA  = 3'b010, // Wait for AXI R response
    RD_COMPLETE   = 3'b011, // Descriptor fetched, push to FIFO
    RD_ERROR      = 3'b100  // Error occurred
} read_engine_state_t;
```

### Descriptor Format (256-bit STREAM)

| Bits      | Field               | Description                                          |
|-----------|---------------------|------------------------------------------------------|
| [63:0]    | src_addr            | Source memory address (64-bit, must be aligned)      |
| [127:64]  | dst_addr            | Destination memory address (64-bit, must be aligned) |
| [159:128] | length              | Transfer length in BEATS (not bytes!)                |
| [191:160] | next_descriptor_ptr | Address of next descriptor (32-bit, 0 = last)        |
| [192]     | valid               | Descriptor valid flag                                |
| [193]     | gen_irq             | Generate interrupt on completion                     |
| [194]     | last                | Last descriptor in chain (explicit termination)      |
| [195]     | error               | Descriptor error flag                                |
| [199:196] | channel_id          | Channel identifier                                   |
| [207:200] | priority            | Descriptor priority                                  |
| [255:208] | reserved            | Reserved for future use                              |

## Parameters

| Parameter            | Type | Default               | Description                   |
|----------------------|------|-----------------------|-------------------------------|
| CHANNEL_ID           | int  | 0                     | Channel identifier            |
| NUM_CHANNELS         | int  | 32                    | Total number of channels      |
| CHAN_WIDTH           | int  | \$clog2(NUM_CHANNELS) | Channel ID width              |
| ADDR_WIDTH           | int  | 64                    | Address bus width             |
| AXI_ID_WIDTH         | int  | 8                     | AXI transaction ID width      |
| FIFO_DEPTH           | int  | 8                     | Descriptor data FIFO depth    |
| DESC_ADDR_FIFO_DEPTH | int  | 2                     | Descriptor address FIFO depth |
| TIMEOUT_CYCLE_S      | int  | 1000                  | AXI timeout threshold         |

## Monitor Bus Parameters

| Parameter      | Type  | Default | Description                |
|----------------|-------|---------|----------------------------|
| MON_AGENT_ID   | 8-bit | 0x10    | Descriptor Engine Agent ID |
| MON_UNIT_ID    | 4-bit | 0x1     | Unit identifier            |
| MON_CHANNEL_ID | 6-bit | 0x0     | Base channel ID            |

## Port List

### Clock and Reset

| Signal | Direction | Width | Description                   |
|--------|-----------|-------|-------------------------------|
| clk    | input     | 1     | System clock                  |
| rst_n  | input     | 1     | Active-low asynchronous reset |

### APB Programming Interface

| Signal    | Direction | Width      | Description              |
|-----------|-----------|------------|--------------------------|
| apb_valid | input     | 1          | Descriptor address valid |
| apb_ready | output    | 1          | Ready to accept address  |
| apb_addr  | input     | ADDR_WIDTH | Descriptor address       |

### Scheduler Interface

| Signal            | Direction | Width | Description                  |
|-------------------|-----------|-------|------------------------------|
| channel_idle      | input     | 1     | Scheduler idle (enables APB) |
| descriptor_val_id | output    | 1     | Descriptor available         |
| descriptor_ready  | input     | 1     | Scheduler ready to accept    |
| descriptor_packet | output    | 256   | FIXED 256-bit descriptor     |
| descriptor_error  | output    | 1     | Fetch error flag             |

### Enhanced Control Outputs

| Signal         | Direction | Width | Description                |
|----------------|-----------|-------|----------------------------|
| descriptor_eos | output    | 1     | End of Stream (future use) |
| descriptor_eol | output    | 1     | End of Line (future use)   |

| Signal          | Direction | Width | Description                 |
|-----------------|-----------|-------|-----------------------------|
| descriptor_eo_d | output    | 1     | End of Data<br>(future use) |
| descriptor_type | output    | 2     | Packet type<br>(future use) |

#### AXI4 AR Channel

| Signal    | Direction | Width        | Description             |
|-----------|-----------|--------------|-------------------------|
| ar_valid  | output    | 1            | Address valid           |
| ar_ready  | input     | 1            | Address ready           |
| ar_addr   | output    | ADDR_WIDTH   | Address                 |
| ar_len    | output    | 8            | Burst length - 1        |
| ar_size   | output    | 3            | Burst size (log2 bytes) |
| ar_burst  | output    | 2            | Burst type (INCR)       |
| ar_id     | output    | AXI_ID_WIDTH | Transaction ID          |
| ar_lock   | output    | 1            | Lock type               |
| ar_cache  | output    | 4            | Cache attributes        |
| ar_prot   | output    | 3            | Protection attributes   |
| ar_qos    | output    | 4            | QoS value               |
| ar_region | output    | 4            | Region identifier       |

#### AXI4 R Channel (FIXED 256-bit)

| Signal  | Direction | Width | Description               |
|---------|-----------|-------|---------------------------|
| r_valid | input     | 1     | Read data valid           |
| r_ready | output    | 1     | Read data ready           |
| r_data  | input     | 256   | Read data (FIXED 256-bit) |
| r_resp  | input     | 2     | Response                  |

| Signal | Direction | Width        | Description    |
|--------|-----------|--------------|----------------|
| r_last | input     | 1            | Last beat      |
| r_id   | input     | AXI_ID_WIDTH | Transaction ID |

### Configuration Interface

| Signal              | Direction | Width      | Description                 |
|---------------------|-----------|------------|-----------------------------|
| cfg_prefetch_enable | input     | 1          | Enable descriptor prefetch  |
| cfg_fifo_thre shold | input     | 4          | FIFO threshold for prefetch |
| cfg_addr0_bas e     | input     | ADDR_WIDTH | Address range 0 base        |
| cfg_addr0_lim it    | input     | ADDR_WIDTH | Address range 0 limit       |
| cfg_addr1_bas e     | input     | ADDR_WIDTH | Address range 1 base        |
| cfg_addr1_lim it    | input     | ADDR_WIDTH | Address range 1 limit       |
| cfg_channel_re set  | input     | 1          | Channel reset               |

### Status Interface

| Signal                 | Direction | Width | Description |
|------------------------|-----------|-------|-------------|
| descriptor_engine_idle | output    | 1     | Engine idle |

### Monitor Bus Interface

| Signal     | Direction | Width | Description          |
|------------|-----------|-------|----------------------|
| mon_valid  | output    | 1     | Monitor packet valid |
| mon_ready  | input     | 1     | Monitor packet ready |
| mon_packet | output    | 64    | Monitor packet data  |

## Operation

### APB Acceptance Policy

APB writes are accepted ONLY when ALL conditions met: 1. !

r\_channel\_reset\_active - Not in channel reset  
2. w\_desc\_addr\_fifo\_empty - No pending descriptor fetches  
3. channel\_idle - Scheduler completed all prior descriptors  
4. !r\_apb\_ip - No APB transaction currently in progress  
5. apb\_addr != 0 - Address 0 is invalid

### Autonomous Chaining Decision Tree

**Level 1: Basic Eligibility (w\_chain\_condition)** - next\_descriptor\_ptr != 0 (non-null pointer) - descriptor.last == 0 (not explicitly marked as last) - descriptor.valid == 1 (valid descriptor)

**Level 2: Address Validation (w\_next\_addr\_valid)** - next\_addr within cfg\_addr0\_base..cfg\_addr0\_limit OR - next\_addr within cfg\_addr1\_base..cfg\_addr1\_limit

**Level 3: Final Decision (w\_should\_chain)** - Level 1 + Level 2 conditions met - ! r\_descriptor\_error (no fetch error) - w\_desc\_fifo\_wr\_ready (descriptor FIFO has space)

### AXI Transaction

```
ar_len = 8'h00;           // Single beat transfer
ar_size = 3'b110;         // 64 bytes (512-bit for 256-bit descriptor)
ar_burst = 2'b01;         // INCR burst type
ar_id = {{padding}, CHANNEL_ID}; // Channel ID in lower bits
ar_cache = 4'b0010;        // Normal non-cacheable bufferable
ar_prot = 3'b000;          // Data, secure, unprivileged
```

---

### Integration Example

```
descriptor_engine #(
    .CHANNEL_ID          (ch),
    .NUM_CHANNELS         (8),
    .ADDR_WIDTH           (64),
    .AXI_ID_WIDTH         (8),
    .FIFO_DEPTH           (8),
    .MON_AGENT_ID         (8'h10 + ch)
) u_descriptor_engine (
    .clk                  (clk),
    .rst_n                (rst_n),
```

```

// APB interface
.apb_valid      (apb_valid),
.apb_ready      (apb_ready),
.apb_addr       (apb_addr),

// Scheduler interface
.channel_idle   (scheduler_idle),
.descriptor_valid (desc_valid),
.descriptor_ready (desc_ready),
.descriptor_packet (desc_packet),
.descriptor_error (desc_error),

// AXI interface
.ar_valid        (ar_valid),
.ar_ready        (ar_ready),
.ar_addr         (ar_addr),
// ... full AR/R interface

// Configuration
.cfg_prefetch_enable (cfg_desceng_prefetch),
.cfg_addr0_base    (cfg_desceng_addr0_base),
.cfg_addr0_limit   (cfg_desceng_addr0_limit),
.cfg_addr1_base    (cfg_desceng_addr1_base),
.cfg_addr1_limit   (cfg_desceng_addr1_limit),
.cfg_channel_reset (cfg_channel_reset),

// Status
.descriptor_engine_idle (desc_engine_idle),

// Monitor bus
.mon_valid        (desceng_mon_valid),
.mon_ready        (desceng_mon_ready),
.mon_packet       (desceng_mon_packet)
);

```

---

## Common Issues

### Issue 1: APB Write Not Accepted

**Symptom:** apb\_ready never asserts

**Root Causes:** 1. Channel not idle (channel\_idle = 0) 2. Previous APB transaction still in progress (r\_apb\_ip = 1) 3. Descriptor address FIFO not empty 4. Channel reset active

**Solution:** Wait for channel to complete current work before next kick-off.

## Issue 2: Descriptor Chain Stops Early

**Symptom:** Chaining stops before expected

**Root Causes:** 1. `next_descriptor_ptr` is 0 or out of valid range 2. `last` flag set in descriptor 3. AXI error during fetch

**Debug:** Check descriptor memory contents and address range configuration.

---

## Related Documentation

- **Parent:** `03_scheduler_group.md` - Channel wrapper
  - **Consumer:** `04_scheduler.md` - Scheduler that receives descriptors
  - **Arbiter:** `02_scheduler_group_array.md` - Descriptor AXI arbitration
  - **APB Interface:** `13_apbtodescr.md` - APB to descriptor kick-off
- 

**Last Updated:** 2025-11-30 (verified against RTL implementation)

## AXI Read Engine

**Module:** `axi_read_engine.sv` **Location:** `projects/components/stream/rtl/fub/`

**Category:** FUB (Functional Unit Block) **Parent:** `stream_core.sv` **Status:**

Implemented **Last Updated:** 2025-11-30

---

## Overview

The `axi_read_engine` module is a high-performance multi-channel AXI4 read engine with space-aware arbitration. It serves all 8 STREAM channels through a single AXI master interface with intelligent flow control.

## Key Features

- **Round-Robin Arbitration:** Fair scheduling across channels
- **Space-Aware Masking:** Only arbitrate channels with sufficient SRAM space
- **Pre-Allocation Handshake:** Reserve SRAM space before data arrives
- **Streaming Data Path:** Direct passthrough to SRAM controller
- **Channel ID in AXI ID:** Enables per-channel response routing

- **Pipelined/Non-Pipelined Modes:** Configurable outstanding transaction depth
- 

## Architecture

### Operation Flow

1. Scheduler Interface: Each channel can request read bursts
2. Space Checking: Mask channels without sufficient SRAM space
3. Arbitration: Round-robin arbiter selects next channel to service
4. AXI AR Issue: Issue read command to AXI, assert rd\_alloc to SRAM controller
5. AXI R Response: Stream read data directly to SRAM controller

### Key Design Decisions

**Combinational AR Outputs:** AR outputs are driven combinationally from the arbiter to avoid 1-cycle delay. When axi\_rd\_alloc\_space\_free goes to 0, arvalid drops in the same cycle.

**No Internal Buffering:** The engine is a streaming pipeline with no internal data storage. Data flows directly from AXI R channel to SRAM controller.

### System Idle Behavior

When ALL channels complete their work (`w_arb_request == 0`), the engine becomes idle until new requests arrive. This is NOT a bubble - it's legitimate system idle.

**Why This Matters for Testing:** - With few active channels and short bursts, there are brief periods when all channels are in their WRITE phase - During these periods, `m_axi_arvalid = 0` (no R channel activity) - This is EXPECTED behavior, not a performance bug - The `dbg_arb_request` signal exposes `w_arb_request` to help testbenches distinguish:  
- **TRUE BUBBLE:** `arvalid=0` while `arb_request!=0` (channels waiting, arbiter stalled)  
- **SYSTEM IDLE:** `arvalid=0` while `arb_request==0` (no channels need service)

---

## Parameters

| Parameter    | Type | Default | Description        |
|--------------|------|---------|--------------------|
| NUM_CHANNELS | int  | 8       | Number of channels |
| ADDR_WIDTH   | int  | 64      | AXI address width  |

| Parameter          | Type | Default | Description                                  |
|--------------------|------|---------|----------------------------------------------|
| DATA_WIDTH         | int  | 512     | AXI data width                               |
| ID_WIDTH           | int  | 8       | AXI ID width                                 |
| SEG_COUNT_WIDTH    | int  | 8       | Width of space/count signals                 |
| PIPELINE           | int  | 0       | 0: non-pipelined, 1: pipelined               |
| AR_MAX_OUTSTANDING | int  | 8       | Maximum outstanding AR requests (PIPELINE=1) |
| STROBE_EVERY_BEAT  | int  | 0       | 0: strobe on last beat, 1: strobe every beat |

## Derived Parameters

| Parameter | Derivation      | Description                  |
|-----------|-----------------|------------------------------|
| NC        | NUM_CHANNELS    | Short alias                  |
| AW        | ADDR_WIDTH      | Short alias                  |
| DW        | DATA_WIDTH      | Short alias                  |
| IW        | ID_WIDTH        | Short alias                  |
| SCW       | SEG_COUNT_WIDTH | Segment count width          |
| CIW       | \$clog2(NC)     | Channel ID width (min 1 bit) |

## Port List

### Clock and Reset

| Signal | Direction | Width | Description                   |
|--------|-----------|-------|-------------------------------|
| clk    | input     | 1     | System clock                  |
| rst_n  | input     | 1     | Active-low asynchronous reset |

### Configuration Interface

| Signal                | Direction | Width | Description                  |
|-----------------------|-----------|-------|------------------------------|
| cfg_axi_rd_xfer_beats | input     | 8     | Transfer size in beats (all) |

| Signal              | Direction | Width | Description |
|---------------------|-----------|-------|-------------|
| Number of channels) |           |       |             |

### Scheduler Interface (Per-Channel)

| Signal              | Direction | Width   | Description             |
|---------------------|-----------|---------|-------------------------|
| sched_rd_vali_d[ch] | input     | NC      | Channel requests read   |
| sched_rd_addr [ch]  | input     | NC x AW | Source addresses        |
| sched_rd_beat s[ch] | input     | NC x 32 | Beats remaining to read |

### Completion Interface (Per-Channel)

| Signal                   | Direction | Width   | Description                     |
|--------------------------|-----------|---------|---------------------------------|
| sched_rd_done_strobe[ch] | output    | NC      | Burst completed (1 cycle pulse) |
| sched_rd_beats_done[ch]  | output    | NC x 32 | Number of beats completed       |

### SRAM Allocation Interface

| Signal                      | Direction | Width    | Description              |
|-----------------------------|-----------|----------|--------------------------|
| axi_rd_alloc_req            | output    | 1        | Request space allocation |
| axi_rd_alloc_size           | output    | 8        | Beats to reserve         |
| axi_rd_alloc_id             | output    | IW       | Transaction ID (channel) |
| axi_rd_alloc_space_free[ch] | input     | NC x SCW | Free space per channel   |

### SRAM Write Interface

| Signal             | Direction | Width | Description          |
|--------------------|-----------|-------|----------------------|
| axi_rd_sram_v alid | output    | 1     | Read data valid      |
| axi_rd_sram_r eady | input     | 1     | Ready to accept data |
| axi_rd_sram_i      | output    | IW    | Transaction ID       |

| Signal            | Direction | Width | Description       |
|-------------------|-----------|-------|-------------------|
| d                 |           |       | (channel)         |
| axi_rd_sram_d ata | output    | DW    | Read data payload |

### AXI4 AR Channel

| Signal        | Direction | Width | Description             |
|---------------|-----------|-------|-------------------------|
| m_axi_arvalid | output    | 1     | Address valid           |
| m_axi_arready | input     | 1     | Address ready           |
| m_axi_arid    | output    | IW    | Transaction ID          |
| m_axi_araddr  | output    | AW    | Address                 |
| m_axi_arlen   | output    | 8     | Burst length - 1        |
| m_axi_arsize  | output    | 3     | Burst size (log2 bytes) |
| m_axi_arburst | output    | 2     | Burst type (INCR)       |

### AXI4 R Channel

| Signal       | Direction | Width | Description        |
|--------------|-----------|-------|--------------------|
| m_axi_rvalid | input     | 1     | Read data valid    |
| m_axi_rready | output    | 1     | Read data ready    |
| m_axi_rid    | input     | IW    | Transaction ID     |
| m_axi_rdata  | input     | DW    | Read data          |
| m_axi_rresp  | input     | 2     | Response           |
| m_axi_rlast  | input     | 1     | Last beat of burst |

### Error Interface

| Signal             | Direction | Width | Description                   |
|--------------------|-----------|-------|-------------------------------|
| sched_rd_error[ch] | output    | NC    | Sticky error flag per channel |

## Debug Interface

| Signal                  | Direction | Width | Description            |
|-------------------------|-----------|-------|------------------------|
| dbg_rd_all_complete[ch] | output    | NC    | All reads complete     |
| dbg_r_beats_rcvd        | output    | 32    | Total R beats received |
| dbg_sram_writes         | output    | 32    | Total writes to SRAM   |
| dbg_arb_request[ch]     | output    | NC    | Arbiter request vector |

## Operation

### Space-Aware Request Masking

```
// Only request arbitration if:  
// 1. Scheduler is requesting (sched_rd_valid)  
// 2. Sufficient SRAM space available (w_space_ok)  
// 3. Below outstanding limit (w_below_outstanding_limit)  
w_arb_request[i] = sched_rd_valid[i] && w_space_ok[i] &&  
w_below_outstanding_limit[i];
```

### Transfer Size Calculation

```
// Calculate actual transfer size for this channel  
// Min of remaining beats or configured max  
w_transfer_size[i] = (sched_rd_beats[i] <= cfg_axi_rd_xfer_beats +  
1) ?  
    (sched_rd_beats[i] - 1) : cfg_axi_rd_xfer_beats;
```

### Outstanding Transaction Tracking

**PIPELINE=0 (Non-Pipelined):** - Binary flag per channel (0 or 1 outstanding) - Set when AR issues for channel - Clear when R last arrives for channel

**PIPELINE=1 (Pipelined):** - Counter per channel (0 to AR\_MAX\_OUTSTANDING) - Increment on AR handshake - Decrement on R last handshake

### Completion Strobe

Strobes fire when AR transaction is accepted (not when R completes):

```
// Pulse when AR transaction is accepted (arvalid && arready)  
// This tells scheduler that request was issued to AXI bus  
if (m_axi_arvalid && m_axi_arready) begin  
    r_done_strobe[w_arb_grant_id] <= 1'b1;
```

```

    r_beats_done[w_arb_grant_id] <= w_transfer_size[w_arb_grant_id] +
1;
end

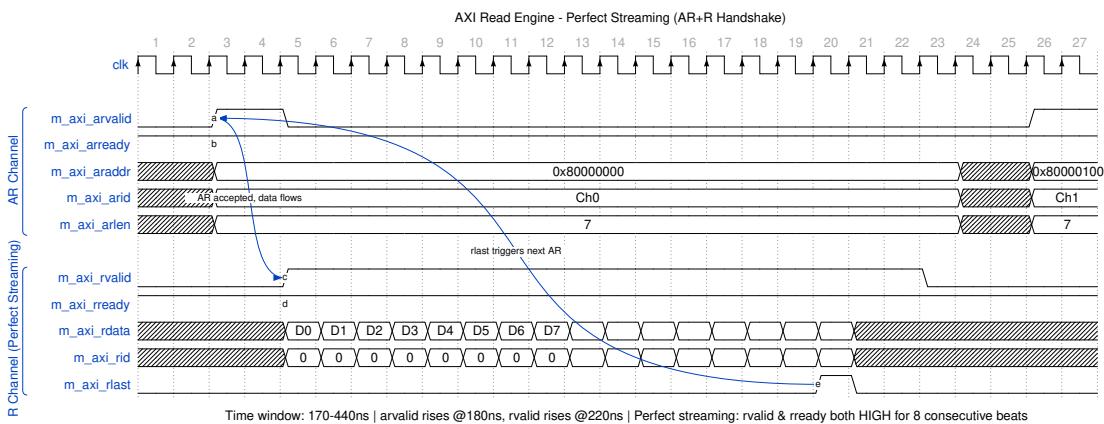
```

---

## Timing Diagrams

### Perfect Streaming - AXI Read Transaction

The following timing diagram shows the AXI read engine operating at maximum throughput with **perfect streaming** - where both `rvalid` and `rready` remain HIGH for consecutive clock cycles, achieving one data beat per cycle.



### AXI Read Engine - Perfect Streaming

#### Transaction Flow:

- 1. AR Channel Handshake (Address Phase)**
  - `m_axi_arvalid` rises to initiate a read request
  - `m_axi_arready` is already HIGH (slave ready)
  - Address (`araddr`), ID (`arid`), and length (`arlen=7` for 8 beats) are captured
  - Handshake completes in a single cycle when both valid and ready are HIGH
- 2. R Channel Streaming (Data Phase)**
  - After AR acceptance, the AXI slave begins returning data
  - `m_axi_rvalid` rises and **stays HIGH** for all 8 beats
  - `m_axi_rready` remains HIGH throughout (no backpressure from SRAM controller)
  - Perfect streaming:** One data beat transferred every clock cycle

- `m_axi_rlast` rises on the final beat (beat 7) to mark burst completion

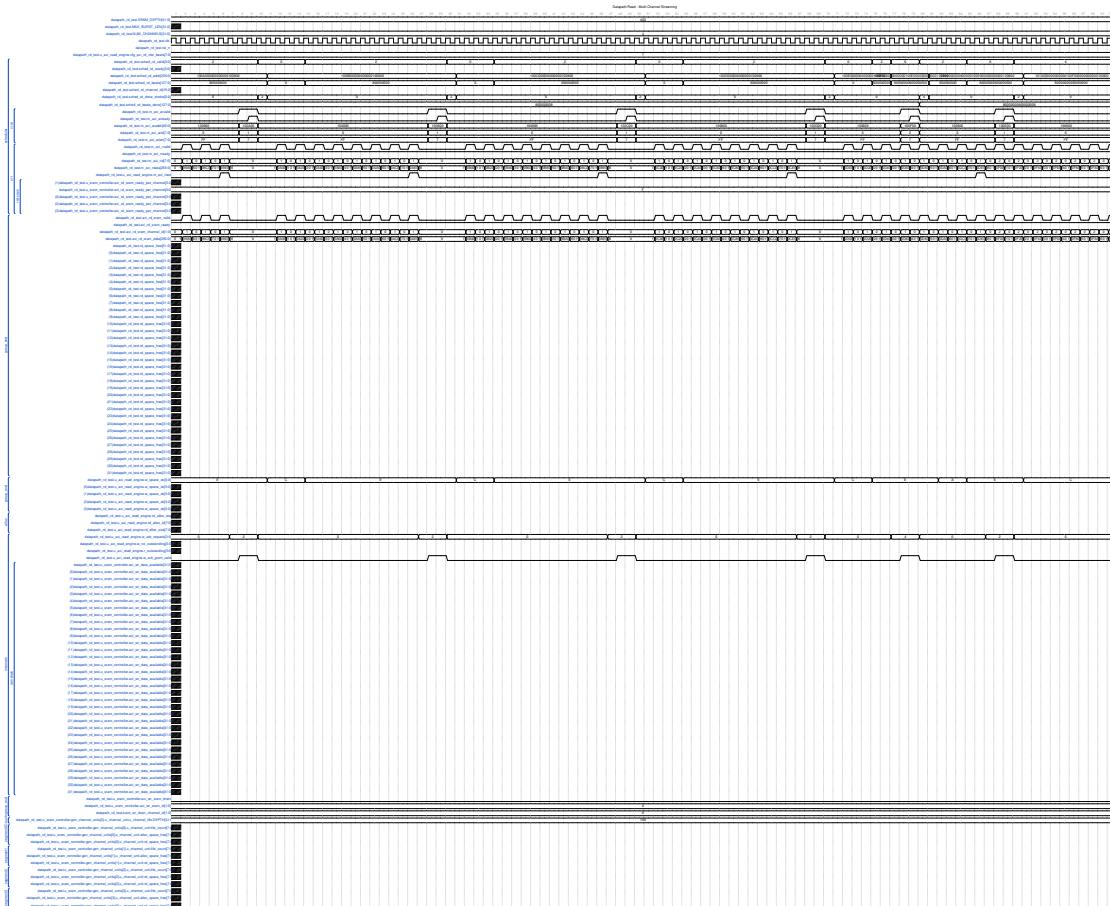
### 3. Next Transaction

- After `rlast`, the engine can immediately issue the next AR request
- The cycle repeats for the next channel or next burst

**Key Performance Indicators:** - **No bubbles:** `rvalid` and `rready` both HIGH during data phase - **Full bandwidth:** Data width (256/512 bits) x clock frequency - **Zero-wait states:** SRAM controller accepts data at line rate

## Multi-Channel Streaming

For multi-channel operation showing channel switching while maintaining streaming performance, see:



## Datapath Read - Multi-Channel

This diagram shows how the engine arbitrates between channels while maintaining high throughput.

---

## Integration Example

```
axi_read_engine #(
    .NUM_CHANNELS      (8),
    .ADDR_WIDTH        (64),
    .DATA_WIDTH        (512),
    .ID_WIDTH          (8),
    .SEG_COUNT_WIDTH  (10),
    .PIPELINE          (0),
    .AR_MAX_OUTSTANDING (8)
) u_axi_read_engine (
    .clk                (clk),
    .rst_n              (rst_n),

    // Configuration
    .cfg_axi_rd_xfer_beats (cfg_axi_rd_xfer_beats),

    // Scheduler interface
    .sched_rd_valid      (sched_rd_valid),
    .sched_rd_addr        (sched_rd_addr),
    .sched_rd_beats       (sched_rd_beats),

    // Completion interface
    .sched_rd_done_strobe (sched_rd_done_strobe),
    .sched_rd_beats_done (sched_rd_beats_done),

    // SRAM allocation interface
    .axi_rd_alloc_req     (axi_rd_alloc_req),
    .axi_rd_alloc_size    (axi_rd_alloc_size),
    .axi_rd_alloc_id      (axi_rd_alloc_id),
    .axi_rd_alloc_space_free(axi_rd_alloc_space_free),

    // SRAM write interface
    .axi_rd_sram_valid    (axi_rd_sram_valid),
    .axi_rd_sram_ready    (axi_rd_sram_ready),
    .axi_rd_sram_id       (axi_rd_sram_id),
    .axi_rd_sram_data     (axi_rd_sram_data),

    // AXI master
    .m_axi_arvalid        (m_axi_rd_arvalid),
    .m_axi_arready        (m_axi_rd_arready),
    .m_axi_arid           (m_axi_rd_arid),
    .m_axi_araddr         (m_axi_rd_araddr),
    .m_axi_arlen          (m_axi_rd_arlen),
    .m_axi_arsize          (m_axi_rd_arsize),
    .m_axi_arburst         (m_axi_rd_arburst),

    .m_axi_rvalid          (m_axi_rd_rvalid),
    .m_axi_rready          (m_axi_rd_rready),
```

```
.m_axi_rid          (m_axi_rd_rid),
.m_axi_rdata        (m_axi_rd_rdata),
.m_axi_rresp        (m_axi_rd_rresp),
.m_axi_rlast        (m_axi_rd_rlast),

// Error and debug
.sched_rd_error     (sched_rd_error),
.dbg_rd_all_complete (dbg_rd_all_complete),
.dbg_arb_request   (dbg_arb_request)
);
```

---

## Related Documentation

- **Parent:** 01\_stream\_core.md - Top-level integration
  - **Scheduler Array:** 02\_scheduler\_group\_array.md - Provides sched\_rd\_\* signals
  - **Allocation Controller:** 07\_stream\_alloc\_ctrl.md - Space tracking for read engine
  - **SRAM Controller:** 08\_sram\_controller.md - Receives read data
  - **Write Engine:** 12\_axi\_write\_engine.md - Complementary write datapath
- 

**Last Updated:** 2025-12-13 (added timing diagrams)

## Stream Allocation Controller

**Module:** stream\_alloc\_ctrl.sv **Location:**

projects/components/stream/rtl/fub/ **Category:** FUB (Functional Unit Block)

**Parent:** sram\_controller\_unit.sv **Status:** Implemented **Last Updated:** 2025-11-30

---

## Overview

The stream\_alloc\_ctrl module is a **virtual FIFO without data storage** that tracks space allocation and availability using FIFO pointer logic. It provides pre-allocation support for AXI read engines to prevent race conditions between burst requests and data arrival.

## What Makes This a "Virtual FIFO"

**Virtual FIFO:** - Has write pointer (allocation pointer) - Has read pointer (actual write pointer) - Calculates full/empty/space\_free - **NO data storage** - only pointer arithmetic

**Use Case:** Reserve FIFO space BEFORE AXI read data arrives

## The Allocation Problem

### Without Allocation Controller

**Problem:** Race condition between space check and data arrival

Cycle 0: Read engine checks FIFO space  
space\_free = 100 beats → OK to issue 16-beat burst

Cycle 5: AR handshake completes, AXI read starts

Cycle 10: Meanwhile, write engine drains 90 beats from FIFO  
space\_free = 10 beats (NOT ENOUGH!)

Cycle 15: AXI read data starts arriving (16 beats)  
→ OVERFLOW! Only 10 beats of space available

### With Allocation Controller

**Solution:** Reserve space when issuing AR, not when data arrives

Cycle 0: Read engine checks space\_free = 100 beats  
Allocate 16 beats:  
rd\_alloc\_req = 1, rd\_alloc\_size = 16  
→ space\_free = 84 beats (reserved!)

Cycle 5: AR handshake completes

Cycle 10: Write engine checks space\_free = 84 beats  
→ Sees reduced space, won't overdrain

Cycle 15: AXI read data arrives, enters FIFO  
→ Space was reserved, guaranteed to fit

## Architecture

### Two-Pointer System



*Diagram*

→

**Write Pointer (Allocation Pointer):** - Advances when space is **allocated** (burst request) - Variable-size increment (rd\_alloc\_size) - Represents “reserved space”

**Read Pointer (Actual Write Pointer):** - Advances when data **exits the controller** (output handshake) - Single-beat increment - Represents “released space”

**Space Calculation:**

space\_free = DEPTH - (wr\_ptr - rd\_ptr)

### CRITICAL: Confusing Naming Convention

**WARNING:** Allocation controller uses OPPOSITE naming from normal FIFO!

**Normal FIFO:** | Signal | Meaning | —— | —— | | wr\_\* | Write data →  
Consumes space → space\_free decreases | | rd\_\* | Read data → Frees space →  
space\_free increases |

**Allocation Controller:** | Signal | Meaning | —— | —— | | wr\_\* | **ALLOCATE**  
space → Reserves space → space\_free **decreases** | | rd\_\* | **RELEASE** space →  
Data exits controller → space\_free **increases** |

**Why This Matters:**

```

// "Write" side = ALLOCATE (reserve space for upcoming burst)
// Read engine says "I need 16 beats"
.wr_valid (rd_alloc_req),
.wr_size (rd_alloc_size), // Variable size (16 in this example)

// "Read" side = RELEASE (data exits controller, free space)
// Must monitor OUTPUT handshake (after latency bridge!)
.rd_valid (axi_wr_sram_valid && axi_wr_sram_ready), // Output side!

```

**Key Insight:** The “read” side of allocation controller connects to the OUTPUT of the FIFO + latency bridge, NOT the FIFO input!

## Parameters

| Parameter         | Type | Default | Description                                   |
|-------------------|------|---------|-----------------------------------------------|
| DEPTH             | int  | 512     | Virtual FIFO depth (must match physical FIFO) |
| ALMOST_WR_MAR_GIN | int  | 1       | Almost full threshold                         |
| ALMOST_RD_MAR_GIN | int  | 1       | Almost empty threshold                        |
| REGISTERED        | int  | 1       | Register outputs for timing                   |

## Port List

### Clock and Reset

| Signal      | Direction | Width | Description                   |
|-------------|-----------|-------|-------------------------------|
| axi_aclk    | input     | 1     | System clock                  |
| axi_aresetn | input     | 1     | Active-low asynchronous reset |

### Write Interface (Allocation Requests)

| Signal   | Direction | Width | Description            |
|----------|-----------|-------|------------------------|
| wr_valid | input     | 1     | Allocate space request |

| Signal   | Direction | Width | Description                   |
|----------|-----------|-------|-------------------------------|
| wr_size  | input     | 8     | Number of entries to allocate |
| wr_ready | output    | 1     | Space available (! wr_full)   |

### Read Interface (Actual Data Written)

| Signal   | Direction | Width | Description                           |
|----------|-----------|-------|---------------------------------------|
| rd_valid | input     | 1     | Data exits controller (release space) |
| rd_ready | output    | 1     | Not empty (! rd_empty)                |

### Status Outputs

| Signal          | Direction | Width | Description                    |
|-----------------|-----------|-------|--------------------------------|
| space_free      | output    | AW+1  | Available space (beats)        |
| wr_full         | output    | 1     | Full flag (no space available) |
| wr_almost_full  | output    | 1     | Almost full flag               |
| rd_empty        | output    | 1     | Empty flag (no allocations)    |
| rd_almost_empty | output    | 1     | Almost empty flag              |

## Interfaces

### Write Interface (Allocation Requests)

| Signal   | Direction | Width | Description          |
|----------|-----------|-------|----------------------|
| wr_valid | Input     | 1     | Allocate space       |
| wr_size  | Input     | 8     | Number of entries to |

| Signal   | Direction | Width | Description                |
|----------|-----------|-------|----------------------------|
|          |           |       | allocate                   |
| wr_ready | Output    | 1     | Space available (!wr_full) |

### Usage:

```
// Read engine allocates space before issuing AR
rd_alloc_req = (space_check_ok && !ar_pending);
rd_alloc_size = cfg_axi_rd_xfer_beats;
```

### Read Interface (Actual Data Written)

| Signal   | Direction | Width | Description           |
|----------|-----------|-------|-----------------------|
| rd_valid | Input     | 1     | Data exits controller |
| rd_ready | Output    | 1     | Not empty (!rd_empty) |

### Usage:

```
// Connect to OUTPUT handshake (after latency bridge)
assign rd_valid = (axi_wr_sram_valid && axi_wr_sram_ready);
```

### Status Outputs

| Signal          | Direction | Width | Description                |
|-----------------|-----------|-------|----------------------------|
| space_free      | Output    | AW+1  | Available unreserved space |
| wr_full         | Output    | 1     | No space available         |
| wr_almost_full  | Output    | 1     | Almost full                |
| rd_empty        | Output    | 1     | No allocations pending     |
| rd_almost_empty | Output    | 1     | Almost empty               |

**Note:** space\_free is the most important output - used by read engine for space checking.

# Operation

## Allocation Flow

### Step 1: Check Space

```
// Read engine checks space availability
if (space_free >= (cfg_axi_rd_xfer_beats << 1)) begin // 2x margin
    // Space OK, proceed to allocation
end
```

### Step 2: Allocate Space

```
rd_alloc_req = 1'b1;
rd_alloc_size = 8'd16; // Reserve 16 beats
```

```
// Next cycle: wr_ptr advances
r_wr_ptr_bin <= r_wr_ptr_bin + 16;
space_free decreases by 16
```

### Step 3: AXI Read Executes

```
// Some cycles later, AXI AR handshake completes
m_axi_arvalid = 1, m_axi_arready = 1
// AXI read starts
```

### Step 4: Data Arrives at FIFO

```
// Many cycles later, AXI read data arrives
axi_rd_sram_valid = 1, axi_rd_sram_ready = 1
// Data enters FIFO (allocation controller doesn't see this directly)
```

### Step 5: Data Exits Controller (After Latency Bridge)

```
// Eventually, data traverses FIFO + latency bridge
axi_wr_sram_valid = 1, axi_wr_sram_ready = 1
```

```
// THIS triggers allocation controller release!
rd_valid = 1
r_rd_ptr_bin <= r_rd_ptr_bin + 1
space_free increases by 1
```

## Pointer Arithmetic

### Write Pointer (Allocation):

```
// Variable-size increment
if (w_write && !r_wr_full) begin
    r_wr_ptr_bin <= r_wr_ptr_bin + (AW+1)'(wr_size);
end
```

## Read Pointer (Release):

```
// Single-beat increment (uses counter_bin utility)
counter_bin #(
    .WIDTH (AW + 1),
    .MAX   (D)
) read_pointer_inst (
    .clk      (axi_aclk),
    .rst_n    (axi_aresetn),
    .enable   (w_read && !r_rd_empty),
    .counter_bin_curr (r_rd_ptr_bin),
    .counter_bin_next (w_rd_ptr_bin_next)
);
```

## Space Calculation:

```
// Occupancy = wr_ptr - rd_ptr
w_count = w_wr_ptr_bin_next - w_rd_ptr_bin_next;

// Space free = total depth - occupancy
space_free = (AW+1)'(D) - w_count;
```

## Timing Behavior

### Allocation Latency

#### Allocation is IMMEDIATE (combinational + 1 cycle):

```
Cycle N: rd_alloc_req = 1, rd_alloc_size = 16
Cycle N+1: r_wr_ptr_bin = old_value + 16
            space_free = old_value - 16
```

Read engine sees updated space\_free on next cycle.

### Release Latency

#### Release is IMMEDIATE (combinational + 1 cycle):

```
Cycle N: axi_wr_sram_valid = 1, axi_wr_sram_ready = 1
          → rd_valid = 1
Cycle N+1: r_rd_ptr_bin = old_value + 1
            space_free = old_value + 1
```

Write engine sees updated space\_free on next cycle.

## Integration Example

### In sram\_controller\_unit.sv

```
stream_alloc_ctrl #(
    .DEPTH(SD),
```

```

    .REGISTERED(1)
) u_alloc_ctrl (
    .axi_aclk          (clk),
    .axi_aresetn       (rst_n),
    // ALLOCATE (reserve space for upcoming burst)
    .wr_valid          (rd_alloc_req),      // From read engine
    .wr_size           (rd_alloc_size),     // Burst size
    .wr_ready          (),                  // Unused (space check
uses space_free)

    // RELEASE (data exits controller - OUTPUT handshake!)
    .rd_valid          (axi_wr_sram_valid && axi_wr_sram_ready), // After latency bridge!
    .rd_ready          (),                  // Unused

    // Space tracking
    .space_free        (alloc_space_free), // To read engine (via
register)

    // Unused status
    .wr_full           (),
    .wr_almost_full   (),
    .rd_empty          (),
    .rd_almost_empty  ()
);

// Register output to break long paths
`ALWAYS_FF_RST(clk, rst_n,
    if (~RST_ASSERTED(rst_n)) begin
        rd_space_free <= SCW'(SD); // Full space on reset
    end else begin
        rd_space_free <= alloc_space_free;
    end
)

```

## Debug Support

### Display Statements

#### Allocation:

```
$display("ALLOC @ %t: allocated %0d beats, wr_ptr: %0d -> %0d,
space_free will be %0d",
        $time, wr_size, r_wr_ptr_bin, r_wr_ptr_bin + wr_size,
        D - (r_wr_ptr_bin + wr_size - r_rd_ptr_bin));
```

#### Release (Drain):

```

$display("DRAIN @ %t: drained 1 beat, rd_ptr: %0d -> %0d, space_free
will be %0d",
        $time, r_rd_ptr_bin, w_rd_ptr_bin_next,
        D - (r_wr_ptr_bin - w_rd_ptr_bin_next));

```

## Waveform Analysis

**Key Signals to Monitor:** - r\_wr\_ptr\_bin - Allocation pointer - r\_rd\_ptr\_bin - Release pointer - space\_free - Available space - rd\_alloc\_req, rd\_alloc\_size - Allocation requests - axi\_wr\_sram\_valid, axi\_wr\_sram\_ready - Release handshake

## Common Issues

### Issue 1: Space Not Released

**Symptom:** space\_free decreases but never increases

**Root Cause:** rd\_valid not connected to output handshake

**Wrong:**

```
.rd_valid (axi_rd_sram_valid && axi_rd_sram_ready) // FIFO input -
WRONG!
```

**Correct:**

```
.rd_valid (axi_wr_sram_valid && axi_wr_sram_ready) // After latency
bridge - CORRECT!
```

### Issue 2: Overflow Despite Allocation

**Symptom:** FIFO overflows even with allocation controller

**Root Cause:** Read engine uses wrong space value

**Wrong:**

```
if (space_free >= cfg_axi_rd_xfer_beats) begin // Exact match -
WRONG!
    allocate();
end
```

**Correct:**

```
if (space_free >= (cfg_axi_rd_xfer_beats << 1)) begin // 2x margin -
CORRECT!
    allocate();
end
```

**Why 2x:** Accounts for in-flight allocations and pipeline delays.

### Issue 3: Allocation Pointer Overflow

**Symptom:** r\_wr\_ptr\_bin wraps at unexpected value

**Root Cause:** Pointer width insufficient

**Check:**

```
parameter int AW = $clog2(D); // Address width
logic [AW:0] r_wr_ptr_bin; // AW+1 bits (for full detection)
```

**Extra bit allows distinguishing full ( $\text{wr\_ptr} = \text{rd\_ptr} + \text{DEPTH}$ ) from empty ( $\text{wr\_ptr} = \text{rd\_ptr}$ ).**

### Comparison with Drain Controller

| Aspect            | Allocation Controller           | Drain Controller                |
|-------------------|---------------------------------|---------------------------------|
| <b>Purpose</b>    | Reserve FIFO space              | Reserve FIFO data               |
| <b>User</b>       | AXI read engine                 | AXI write engine                |
| <b>Write side</b> | Allocation request (reserve)    | Data enters FIFO<br>(increment) |
| <b>Read side</b>  | Data exits controller (release) | Drain request (reserve)         |
| <b>Naming</b>     | OPPOSITE of normal FIFO         | SAME as normal FIFO             |
| <b>Output</b>     | space_free                      | data_available                  |

### Resource Utilization

**Per Instance:** -  $2 \times (\text{AW}+1)$ -bit counters ( $\text{wr\_ptr}$ ,  $\text{rd\_ptr}$ ) -  $1 \times (\text{AW}+1)$ -bit space\_free calculation - FIFO control block (full/empty logic) - ~50-100 flip-flops total

**Example (DEPTH=512, AW=9):** -  $2 \times 10$ -bit counters = 20 FFs - Control logic = ~30 FFs - **Total:** ~50 FFs

**Very lightweight - pointer logic only, no data storage.**

### Related Modules

- **Counterpart:** stream\_drain\_ctrl.sv - Drain-side flow control
- **Parent:** sram\_controller\_unit.sv - Instantiates allocation controller
- **User:** axi\_read\_engine.sv - Checks rd\_space\_free before issuing AR

## Related Documentation

- **Drain Controller:** 11\_stream\_drain\_ctrl.md
  - **SRAM Controller Unit:** 09\_sram\_controller\_unit.md
  - **SRAM Controller:** 08\_sram\_controller.md
  - **AXI Read Engine:** 06\_axi\_read\_engine.md
- 

**Last Updated:** 2025-11-30 (verified against RTL implementation)

## SRAM Controller Specification

**Module:** sram\_controller.sv **Location:** projects/components/stream/rtl/fub/

**Status:** Implemented **Last Updated:** 2025-11-30

---

## Overview

The SRAM Controller provides per-channel buffering between AXI read and write engines using independent FIFO structures. Each channel has its own FIFO with dedicated allocation controller (write side) and drain controller (read side).

## Key Features

- **Per-channel FIFOs:** Independent gaxi\_fifo\_sync per channel (no segmentation complexity)
- **ID-based routing:** Transaction ID selects channel for write/read operations
- **Allocation controller:** Reserves space before AXI read data arrives
- **Drain controller:** Manages data availability for AXI write engine
- **Latency bridge:** Aligns FIFO read latency (registered output)
- **Saturating counters:** 8-bit space/count reporting per channel

## Design Rationale

### Why Per-Channel FIFOs Instead of Segmented SRAM?

The implementation uses **independent FIFOs** rather than a monolithic SRAM divided into segments:

**Advantages of Per-Channel FIFOs:** - **Simpler logic** - FIFOs are standard, well-tested components - **Better isolation** - Channel failures don't affect others -

**Easier timing** - No cross-channel paths or arbitration - **Modular** - Easy to add/remove channels - **No pointer arithmetic** - FIFO handles internally

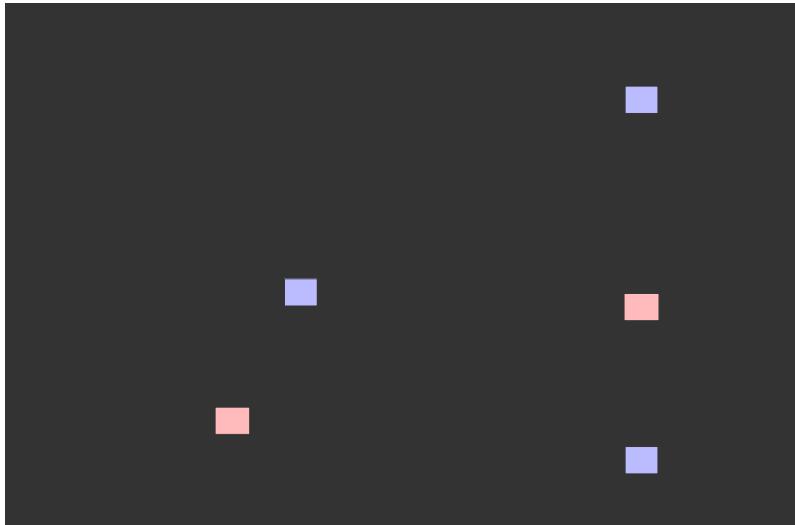
**Trade-offs:** - More SRAM resources ( $NC \times DEPTH$  vs. shared pool) - Potential waste if channels idle (unused FIFO space)

**Decision:** Simplicity and isolation outweigh SRAM efficiency for STREAM's tutorial focus.

---

## Architecture

### Block Diagram



*Diagram*

**Source:** [05\\_sram\\_controller\\_block.mmd](#)

### Per-Channel Architecture

Each channel contains three components (in `sram_controller_unit`):

1. **Allocation Controller (`stream_alloc_ctrl`):**
  - Receives `rd_alloc_req` from read engine
  - Tracks reserved vs. committed space
  - Provides `rd_space_free` to read engine
2. **FIFO (`gaxi_fifo_sync`):**
  - Stores data between read and write engines
  - Depth = `SRAM_DEPTH` parameter

- Standard valid/ready handshaking
3. **Drain Controller + Latency Bridge (`stream_drain_ctrl`):**
- Receives `wr_drain_req` from write engine
  - Provides `wr_drain_data_avail` to write engine
  - Latency bridge aligns FIFO read latency
- 

## Parameters

```

parameter int NUM_CHANNELS = 8;                                // Number of
independent channels
parameter int DATA_WIDTH = 512;                                 // Data width in bits
parameter int SRAM_DEPTH = 512;                                // Depth per channel
FIFO
parameter int SEG_COUNT_WIDTH = $clog2(SRAM_DEPTH) + 1;    // Width of
count signals

// Short aliases (internal use)
parameter int NC = NUM_CHANNELS;
parameter int DW = DATA_WIDTH;
parameter int SD = SRAM_DEPTH;
parameter int SCW = SEG_COUNT_WIDTH;                           // FIFO depth counter
width
parameter int CIW = (NC > 1) ? $clog2(NC) : 1;      // Channel ID width
(min 1 bit)

```

**Note:** “SEG\_COUNT\_WIDTH” refers to the FIFO depth counter width (historical name from segmented design consideration).

---

## Port List

### Clock and Reset

| Signal             | Direction | Width | Description                         |
|--------------------|-----------|-------|-------------------------------------|
| <code>clk</code>   | input     | 1     | System clock                        |
| <code>rst_n</code> | input     | 1     | Active-low<br>asynchronous<br>reset |

### Allocation Interface

#### AXI Read Engine Flow Control (Space Reservation):

| Signal                      | Direction | Width                                     | Description                                 |
|-----------------------------|-----------|-------------------------------------------|---------------------------------------------|
| axi_rd_alloc_req            | input     | 1                                         | Allocation request (single request with ID) |
| axi_rd_alloc_size           | input     | 8                                         | Number of beats to allocate                 |
| axi_rd_alloc_id             | input     | CIW                                       | Channel ID for allocation                   |
| axi_rd_alloc_space_free[ch] | output    | NUM_CHAN<br>NELS ×<br>SEG_COUNT<br>_WIDTH | Free space per channel FIFO                 |

## Write Interface

### AXI Read Engine → FIFO (ID-based routing):

| Signal            | Direction | Width      | Description                             |
|-------------------|-----------|------------|-----------------------------------------|
| axi_rd_sram_valid | input     | 1          | Write data valid (single valid with ID) |
| axi_rd_sram_ready | output    | 1          | Ready (muxed from selected channel)     |
| axi_rd_sram_id    | input     | CIW        | Channel ID select for write             |
| axi_rd_sram_data  | input     | DATA_WIDTH | Write data (common bus)                 |

## Drain Interface

### Write Engine Flow Control (Data Availability):

| Signal                      | Direction | Width                                     | Description                                   |
|-----------------------------|-----------|-------------------------------------------|-----------------------------------------------|
| axi_wr_drain_data_avail[ch] | output    | NUM_CHAN<br>NELS ×<br>SEG_COUNT<br>_WIDTH | Available data after reservations per channel |
| axi_wr_drain_req[ch]        | input     | NUM_CHAN<br>NELS                          | Per-channel drain request                     |
| axi_wr_drain_size[ch]       | input     | NUM_CHAN<br>NELS × 8                      | Per-channel drain size (beats to reserve)     |

## Read Interface

FIFO → AXI Write Engine (ID-based routing):

| Signal                | Direction | Width            | Description                             |
|-----------------------|-----------|------------------|-----------------------------------------|
| axi_wr_sram_valid[ch] | output    | NUM_CHAN<br>NELS | Per-channel valid (data available)      |
| axi_wr_sram_drain     | input     | 1                | Drain request (consumer ready)          |
| axi_wr_sram_id        | input     | CIW              | Channel ID select for read              |
| axi_wr_sram_data      | output    | DATA_WIDTH       | Read data from selected channel (muxed) |

## Debug Interface

| Signal                   | Direction | Width            | Description                             |
|--------------------------|-----------|------------------|-----------------------------------------|
| dbg_bridge_pending[ch]   | output    | NUM_CHAN<br>NELS | Latency bridge pending per channel      |
| dbg_bridge_out_valid[ch] | output    | NUM_CHAN<br>NELS | Latency bridge output valid per channel |

## Interface

### Clock and Reset

```
input logic clk;
input logic rst_n; // Active-low asynchronous reset
```

### Allocation Interface (AXI Read Engine Flow Control)

#### Space Reservation:

```
input logic axi_rd_alloc_req; // Single allocation request
input logic [7:0] axi_rd_alloc_size; // Beats to allocate
input logic [CIW-1:0] axi_rd_alloc_id; // Channel ID for allocation
output logic [NC-1:0][SCW-1:0] axi_rd_alloc_space_free; // Free space per channel
```

**Allocation Protocol:** 1. Read engine issues AR transaction 2. **Before R data arrives:** Engine asserts axi\_rd\_alloc\_req with size and ID 3. Allocation controller reserves space in selected channel 4. Engine tracks reserved space, prevents over-issuing AR commands 5. Space commits when actual data arrives (FIFO write)

### Why Allocation is Critical:

Without pre-allocation, multiple AR commands could be issued before data arrives, causing FIFO overflow:

Problem without allocation:

```
Cycle 0: Issue AR (16 beats) - FIFO has 32 free
Cycle 1: Issue AR (16 beats) - FIFO still shows 32 free (data hasn't arrived!)
Cycle 2: Issue AR (16 beats) - FIFO still shows 32 free
Cycle 10: R data starts arriving (48 beats total)
→ OVERFLOW! Only 32 beats of space
```

Solution with allocation:

```
Cycle 0: Issue AR (16 beats), allocate 16 - FIFO shows 16 free (reserved)
Cycle 1: Issue AR (16 beats), allocate 16 - FIFO shows 0 free (reserved)
Cycle 2: Cannot issue AR - no space available
Cycle 10: R data arrives - space is guaranteed
```

## Write Interface (AXI Read Engine → FIFO)

### ID-Based Write:

|                                      |                    |                    |
|--------------------------------------|--------------------|--------------------|
| <b>input logic</b>                   | axi_rd_sram_valid; | // Single          |
| <i>valid for all channels</i>        |                    |                    |
| <b>input logic [CIW-1:0]</b>         | axi_rd_sram_id;    | // Channel ID      |
| <i>select</i>                        |                    |                    |
| <b>output logic</b>                  | axi_rd_sram_ready; | // Ready           |
| <i>(muxed from selected channel)</i> |                    |                    |
| <b>input logic [DW-1:0]</b>          | axi_rd_sram_data;  | // Shared data bus |

**Write Protocol:** 1. Read engine asserts axi\_rd\_sram\_valid with data 2. axi\_rd\_sram\_id selects which channel FIFO receives data 3. Controller decodes ID to per-channel valid\_decoded[id] 4. Selected channel's FIFO ready muxed to axi\_rd\_sram\_ready

## Drain Interface (AXI Write Engine Flow Control)

### Data Availability:

```



```

**Drain Protocol:** 1. Write engine checks axi\_wr\_drain\_data\_avail[ch] for available data 2. Engine asserts axi\_wr\_drain\_req[ch] to reserve data for W burst 3. Drain controller updates available count (subtracts reserved) 4. Data commits when actually read from FIFO

### Why Drain Reservation is Critical:

Similar to allocation, drain prevents under-reporting data availability:

Problem without drain reservation:

```

Cycle 0: FIFO has 32 beats available
Cycle 1: Issue AW (16 beats) - FIFO still shows 32 available
Cycle 2: Issue AW (16 beats) - FIFO still shows 32 available
Cycle 3: Issue AW (16 beats) - FIFO still shows 32 available
Cycle 10: W bursts start draining (48 beats expected)
→ UNDERFLOW! Only 32 beats available

```

Solution with drain reservation:

```

Cycle 0: FIFO has 32 beats available
Cycle 1: Issue AW (16 beats), drain 16 - shows 16 available
(reserved)
Cycle 2: Issue AW (16 beats), drain 16 - shows 0 available
(reserved)
Cycle 3: Cannot issue AW - no data available
Cycle 10: W bursts drain - data is guaranteed

```

## Read Interface (FIFO → AXI Write Engine)

### ID-Based Read:

```

output logic [NC-1:0]           axi_wr_sram_valid;      // Per-channel
valid (data available)


```

**Read Protocol:** 1. Write engine checks axi\_wr\_sram\_valid[ch] (per-channel) 2. Engine asserts axi\_wr\_sram\_drain with axi\_wr\_sram\_id 3. Controller decodes ID

to per-channel drain\_decoded[id] 4. Selected channel's data muxed to axi\_wr\_sram\_data

### Debug Interface

```
output logic [NC-1:0]          dbg_bridge_pending;    // Latency  
bridge pending per channel  
output logic [NC-1:0]          dbg_bridge_out_valid; // Latency  
bridge output valid per channel
```

**Purpose:** Monitor latency bridge state to catch bugs in read timing.

---

## ID Decode Logic

### Write Valid Decode

**Decode axi\_rd\_sram\_id to per-channel valid:**

```
always_comb begin  
    axi_rd_sram_valid_decoded = '0;  
    if (axi_rd_sram_valid && axi_rd_sram_id < NC) begin  
        axi_rd_sram_valid_decoded[axi_rd_sram_id] = 1'b1;  
    end  
end
```

**Mux ready from selected channel:**

```
always_comb begin  
    if (axi_rd_sram_id < NC) begin  
        axi_rd_sram_ready =  
            axi_rd_sram_ready_per_channel[axi_rd_sram_id];  
    end else begin  
        axi_rd_sram_ready = 1'b0; // Invalid ID → not ready  
    end  
end
```

### Read/Drain Decode

**Decode axi\_wr\_sram\_id to per-channel drain:**

```
always_comb begin  
    axi_wr_sram_drain_decoded = '0;  
    if (axi_wr_sram_drain && axi_wr_sram_id < NC) begin  
        axi_wr_sram_drain_decoded[axi_wr_sram_id] = 1'b1;  
    end  
end
```

**Mux data from selected channel:**

```

always_comb begin
    if (axi_wr_sram_id < NC) begin
        axi_wr_sram_data =
            axi_wr_sram_data_per_channel[axi_wr_sram_id];
    end else begin
        axi_wr_sram_data = '0; // Invalid ID → zero data
    end
end

```

## Allocation Decode

Decode axi\_rd\_alloc\_id to per-channel allocation:

```

always_comb begin
    axi_rd_alloc_req_decoded = '0;
    if (axi_rd_alloc_req && axi_rd_alloc_id < NC) begin
        axi_rd_alloc_req_decoded[axi_rd_alloc_id] = 1'b1;
    end
end

```

---

## Per-Channel Unit

Each channel instantiates sram\_controller\_unit (separate module):

```

sram_controller_unit #(
    .DATA_WIDTH(DW),
    .SRAM_DEPTH(SRAM_DEPTH),
    .SEG_COUNT_WIDTH(SEG_COUNT_WIDTH)
) u_channel_unit (
    .clk                (clk),
    .rst_n              (rst_n),

    // Write interface (decoded valid from ID)
    .axi_rd_sram_valid (axi_rd_sram_valid_decoded[i]),
    .axi_rd_sram_ready (axi_rd_sram_ready_per_channel[i]),
    .axi_rd_sram_data  (axi_rd_sram_data), // SHARED

    // Read interface (decoded drain)
    .axi_wr_sram_valid (axi_wr_sram_valid[i]),
    .axi_wr_sram_ready (axi_wr_sram_drain_decoded[i]),
    .axi_wr_sram_data  (axi_wr_sram_data_per_channel[i]),

    // Allocation interface (decoded req from ID)
    .rd_alloc_req      (axi_rd_alloc_req_decoded[i]),
    .rd_alloc_size     (axi_rd_alloc_size), // SHARED
    .rd_space_free     (axi_rd_alloc_space_free[i]),

```

```

// Drain interface (per-channel)
.wr_drain_req      (axi_wr_drain_req[i]),
.wr_drain_size     (axi_wr_drain_size[i]),
.wr_drain_data_avail(axi_wr_drain_data_avail[i]),

// Debug
.dbg_bridge_pending   (dbg_bridge_pending[i]),
.dbg_bridge_out_valid (dbg_bridge_out_valid[i])
);

```

**Key Points:** - axi\_rd\_sram\_data is **shared** - all units see same data bus - Only unit with valid\_decoded[i] = 1 writes to its FIFO - rd\_alloc\_size is **shared** - all see same size value - Only unit with alloc\_req\_decoded[i] = 1 reserves space

---

## Operation Flows

### Write Flow (AXI Read Data → FIFO)

#### Allocation Phase (before data arrives):

1. Read engine issues AR command (araddr, arlen, arid)
2. Engine extracts channel ID from arid[CIW-1:0]
3. Engine asserts:
 

```

        axi_rd_alloc_req = 1
        axi_rd_alloc_id = channel_id
        axi_rd_alloc_size = arlen + 1 (burst size in beats)
      
```
4. Allocation controller (in sram\_controller\_unit):
  - Checks rd\_space\_free[channel\_id] >= alloc\_size
  - Reserves space (doesn't commit yet)
5. Engine proceeds with AR transaction

#### Data Arrival Phase:

1. AXI R data arrives:
 

```

        m_axi_rvalid = 1
        m_axi_rid = transaction_id (contains channel_id in lower bits)
        m_axi_rdata = data
      
```
2. Read engine forwards to SRAM controller:
 

```

        axi_rd_sram_valid = 1
        axi_rd_sram_id = rid[CIW-1:0] (extract channel ID)
        axi_rd_sram_data = rdata
      
```
3. SRAM controller decodes:
 

```

        axi_rd_sram_valid_decoded[channel_id] = 1
      
```
4. Selected channel FIFO:

- Writes data if ready
- Commits 1 beat of reserved space
- Decrement reserved counter

5. Ready mux:

```
axi_rd_sram_ready = fifo_ready[channel_id]
```

### Read Flow (FIFO → AXI Write Data)

#### Drain Reservation Phase (before W data drains):

1. Write engine checks drain\_data\_avail[channel\_id]
2. If sufficient data available:
  - $axi\_wr\_drain\_req[channel\_id] = 1$
  - $axi\_wr\_drain\_size[channel\_id] = burst\_size$
3. Drain controller:
  - Reserves data (decrements available count)
  - Prevents other channels from seeing this data

#### Data Drain Phase:

1. Write engine ready to consume data:
 

```
axi_wr_sram_drain = 1
      axi_wr_sram_id = channel_id
```
  2. SRAM controller decodes:
 

```
axi_wr_sram_drain_decoded[channel_id] = 1
```
  3. Selected channel FIFO:
    - Reads data (1-cycle latency through bridge)
    - Commits 1 beat of reserved data
    - Decrements drain reservation
  4. Data mux:
 

```
axi_wr_sram_data = fifo_data[channel_id]
```
  5. Valid output:
 

```
axi_wr_sram_valid[channel_id] = bridge_valid
```
- 

### Allocation vs. Drain Controllers

#### Why Separate Controllers?

**Allocation Controller (Write Side):** - Problem: AXI read AR issued before R data arrives - Solution: Pre-allocate space when AR issues, commit when R arrives - Prevents: Over-issuing AR commands when FIFO full

**Drain Controller (Read Side):** - Problem: AXI write AW issued before W data drains from FIFO - Solution: Reserve data when AW issues, commit when W drains - Prevents: Under-reporting available data when burst in progress

## Flow Control Comparison

**Read Engine (uses allocation):**

```
// Check space BEFORE issuing AR
if (rd_space_free[ch] >= burst_size) begin
    issue_ar_command();
    assert_rd_alloc_req(); // Reserve space
end
// Later: R data arrives → commits reservation
```

**Write Engine (uses drain):**

```
// Check data availability BEFORE issuing AW
if (wr_drain_data_avail[ch] >= burst_size) begin
    issue_aw_command();
    assert_wr_drain_req(); // Reserve data
end
// Later: W beats drain → commits reservation
```

---

## Timing Diagrams

### Write Path (R Data → FIFO)

|         |                          |             |   |   |   |   |
|---------|--------------------------|-------------|---|---|---|---|
| Cycle:  | 0                        | 1           | 2 | 3 | 4 | 5 |
| Alloc:  | [REQ]                    | ---grant--- |   |   |   |   |
|         | (ch2, size=4)            |             |   |   |   |   |
| R Data: | [V][V][V][V]             |             |   |   |   |   |
| R ID:   | [2][2][2][2]             |             |   |   |   |   |
| Decode: | [1][1][1][1] (channel 2) |             |   |   |   |   |
| FIFO:   | [W][W][W][W]             |             |   |   |   |   |
| Ready:  | [1][1][1][1][1][1]...    |             |   |   |   |   |

Notes:

- Allocation at cycle 0 reserves 4 beats
- R data arrives cycles 3-6
- Each beat commits 1 reserved entry

### Read Path (FIFO → W Data)

|        |   |   |   |   |   |   |   |
|--------|---|---|---|---|---|---|---|
| Cycle: | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|        |   |   |   |   |   |   |   |

```

Drain: [REQ][---reserve---]
      (ch2, size=2)

Drain:          [D][D]
Drain ID:       [2][2]
|           |
Decode:        [1][1] (channel 2)
Bridge:         [-][V][V] (1-cycle latency)
W Data:        [D0][D1]

```

**Notes:**

- Drain request at cycle 0 reserves 2 beats
  - Actual drain at cycles 3-4
  - Latency bridge adds 1 cycle (data at 4-5)
- 

## Error Conditions

### Invalid Channel ID

**Condition:** `axi_rd_sram_id >= NUM_CHANNELS` or `axi_wr_sram_id >= NUM_CHANNELS`

**Handling:** - Write: `axi_rd_sram_ready = 0` (not ready, data dropped) - Read: `axi_wr_sram_data = 0` (zero data output) - **No error signal** - engines should not generate invalid IDs

### FIFO Overflow

**Condition:** Write when FIFO full

**Prevention:** - Allocation controller tracks reserved + committed space - Read engine checks `rd_space_free` before issuing AR - Only issues AR when sufficient space available

**Should never happen if read engine follows protocol!**

### FIFO Underflow

**Condition:** Read when FIFO empty

**Prevention:** - Drain controller tracks available - reserved data - Write engine checks `wr_drain_data_avail` before issuing AW - Only issues AW when sufficient data available

**Should never happen if write engine follows protocol!**

---

## Performance Considerations

### Per-Channel FIFO Benefits

**Isolation:** - Channel failures don't affect others - No cross-channel contention - Simplified debug (each channel independent)

**Timing:** - No shared arbitration delays - Predictable latency per channel - Easier timing closure

**Scalability:** - Easy to add/remove channels - Parameterizable depth per channel - No global SRAM redesign needed

### SRAM Resource Usage

**Total SRAM:**  $\text{NUM\_CHANNELS} \times \text{SRAM\_DEPTH} \times \text{DATA\_WIDTH}$

**Example (8 channels, 512 depth, 512-bit data):** - Per channel:  $512 \times 512$  bits = 32KB - Total:  $8 \times 32\text{KB} = 256\text{KB}$

**Comparison to Segmented Approach:** - Segmented:  $1 \times (8 \times 512) \times 512$  bits = 256KB (same total) - But segmented requires complex pointer arithmetic and arbitration - Per-channel FIFOs trade complexity for slight area increase (FIFO overhead)

---

## Testing

### Test Location:

`projects/components/stream/dv/tests/fub_tests/sram_controller/`

### Key Test Scenarios:

1. **Single channel fill/drain** - Basic FIFO operation
2. **All channels concurrent** - Independent operation
3. **ID decode** - Correct channel selection
4. **Allocation before data** - Space reserved correctly
5. **Drain before read** - Data reserved correctly
6. **Latency bridge** - 1-cycle delay verified
7. **Invalid ID** - Graceful handling
8. **Back-to-back** - No bubbles between bursts

---

## Related Documentation

- **SRAM Controller Unit:** 09\_sram\_controller\_unit.md - Per-channel implementation
  - **Allocation Controller:** 07\_stream\_alloc\_ctrl.md - Write-side flow control
  - **Drain Controller:** 11\_stream\_drain\_ctrl.md - Read-side flow control
  - **AXI Read Engine:** 06\_axi\_read\_engine.md - Write interface usage
  - **AXI Write Engine:** 12\_axi\_write\_engine.md - Read interface usage
  - **Stream Core:** 01\_stream\_core.md - Top-level integration
- 

## Revision History

| Date       | Version | Changes                                                                                                                                                                                                                                                                                |
|------------|---------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 2025-10-18 | 0.5     | Initial draft documenting segmented SRAM approach                                                                                                                                                                                                                                      |
| 2025-11-16 | 1.0     | Updated to document per-channel FIFO implementation (actual RTL)                                                                                                                                                                                                                       |
| 2025-11-21 | 1.5     | <b>Merged documentation:-</b><br>Consolidated duplicate files-<br>Added design rationale section- Enhanced allocation/drain explanations with examples- Clarified per-channel FIFO architecture- Added performance comparison- Verified all content matches current RTL implementation |
| 2025-11-30 | 1.6     | Updated related documentation references                                                                                                                                                                                                                                               |

**Last Updated:** 2025-11-30 (verified against RTL implementation)

# SRAM Controller Unit

**Module:** sram\_controller\_unit.sv **Location:**

projects/components/stream/rtl/fub/ **Category:** FUB (Functional Unit Block)

**Parent:** sram\_controller.sv **Status:** Implemented **Last Updated:** 2025-11-30

---

## Overview

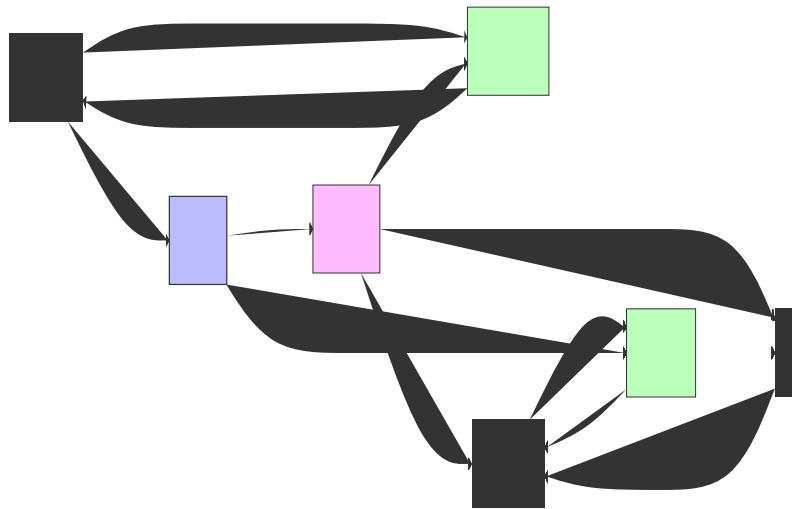
The sram\_controller\_unit module is a single-channel SRAM controller unit containing allocation controller, FIFO buffer, and latency bridge. It handles one channel's data flow from AXI read engine through buffering to AXI write engine with proper flow control.

## Key Features

- **Three-Component Architecture:**
    - Allocation controller (stream\_alloc\_ctrl) - Space tracking for reads
    - FIFO buffer (gaxi\_fifo\_sync) - Physical data storage
    - Latency bridge (stream\_latency\_bridge) - Timing compensation
  - **Drain Controller:** Tracks data availability for write engine
  - **Virtual FIFO Pattern:** Pointer arithmetic without data storage for flow control
  - **Registered Outputs:** Breaks combinatorial paths for timing closure
-

## Architecture

### Block Diagram



*SRAM Controller Unit Block Diagram*

**Source:** [06\\_sram\\_controller\\_unit\\_block.mmd](#)

### Component Hierarchy

```
sram_controller_unit
  stream_alloc_ctrl          # Allocation tracking (space availability)
  stream_drain_ctrl          # Drain tracking (data availability)
  gaxi_fifo_sync              # Physical data storage FIFO
  stream_latency_bridge       # 1-cycle latency compensation
```

### Data Flow

AXI Read Engine → FIFO Write Port → FIFO Storage → FIFO Read Port → Latency Bridge → AXI Write Engine

Allocation Controller (space tracking)

Drain Controller (data tracking)

### Controller Naming Convention (CRITICAL)

**Allocation Controller Perspective:** - wr side = ALLOCATE (reserve space, advance wr\_ptr) - rd side = FULFILL (data arrives, advance rd\_ptr, FREE space)

**Drain Controller Perspective:** - wr side = DATA WRITTEN (increment occupancy) - rd side = DRAIN REQUEST (reserve data for write burst)

This is OPPOSITE of normal FIFO naming conventions!

---

## Parameters

| Parameter       | Type | Default               | Description             |
|-----------------|------|-----------------------|-------------------------|
| DATA_WIDTH      | int  | 512                   | Data width in bits      |
| SRAM_DEPTH      | int  | 512                   | FIFO depth in entries   |
| SEG_COUNT_WIDTH | int  | \$clog2(SRAM_DEPTH)+1 | Width for count signals |

## Derived Parameters

| Parameter  | Derivation      | Description         |
|------------|-----------------|---------------------|
| DW         | DATA_WIDTH      | Short alias         |
| SD         | SRAM_DEPTH      | Short alias         |
| SCW        | SEG_COUNT_WIDTH | Segment count width |
| ADDR_WIDTH | \$clog2(SD)     | FIFO address width  |

## Port List

### Clock and Reset

| Signal | Direction | Width | Description                   |
|--------|-----------|-------|-------------------------------|
| clk    | input     | 1     | System clock                  |
| rst_n  | input     | 1     | Active-low asynchronous reset |

### Allocation Interface (Read Engine Flow Control)

| Signal                  | Direction | Width | Description              |
|-------------------------|-----------|-------|--------------------------|
| axi_rd_alloc_req        | input     | 1     | Request space allocation |
| axi_rd_alloc_size       | input     | 8     | Beats to reserve         |
| axi_rd_alloc_space_free | output    | SCW   | Free space available     |

### Write Interface (AXI Read Engine to FIFO)

| Signal                | Direction | Width | Description          |
|-----------------------|-----------|-------|----------------------|
| axi_rd_sram_v<br>alid | input     | 1     | Write data valid     |
| axi_rd_sram_r<br>eady | output    | 1     | Ready to accept data |
| axi_rd_sram_d<br>ata  | input     | DW    | Write data payload   |

### Drain Interface (Write Engine Flow Control)

| Signal                      | Direction | Width | Description              |
|-----------------------------|-----------|-------|--------------------------|
| axi_wr_drain_<br>data_avail | output    | SCW   | Data available for drain |
| axi_wr_drain_<br>req        | input     | 1     | Request to drain data    |
| axi_wr_drain_<br>size       | input     | 8     | Beats to drain           |

### Read Interface (FIFO to AXI Write Engine)

| Signal                | Direction | Width | Description          |
|-----------------------|-----------|-------|----------------------|
| axi_wr_sram_v<br>alid | output    | 1     | Read data valid      |
| axi_wr_sram_r<br>eady | input     | 1     | Ready to accept data |
| axi_wr_sram_d<br>ata  | output    | DW    | Read data payload    |

### Debug Interface

| Signal                   | Direction | Width | Description              |
|--------------------------|-----------|-------|--------------------------|
| dbg_bridge_pe<br>nding   | output    | 1     | Data in flight in bridge |
| dbg_bridge_o<br>ut_valid | output    | 1     | Bridge output valid      |

## Operation

### Allocation Flow

1. **Space Check:** Read engine checks axi\_rd\_alloc\_space\_free
2. **Reservation:** Read engine asserts axi\_rd\_alloc\_req with burst size
3. **Space Decrement:** Allocation controller decrements space\_free
4. **Data Arrival:** Data enters FIFO via axi\_rd\_sram\_\* interface
5. **Space Release:** When data exits (output handshake), space\_free increments

### Drain Flow

1. **Data Check:** Write engine checks axi\_wr\_drain\_data\_avail
2. **Reservation:** Write engine asserts axi\_wr\_drain\_req with burst size
3. **Data Decrement:** Drain controller decrements data\_available
4. **Data Consumption:** Write engine reads via axi\_wr\_sram\_\* interface

### Data Available Calculation

```
// Total data available = drain controller data + latency bridge
occupancy
assign axi_wr_drain_data_avail = drain_data_available +
SCW'(bridge_occupancy);
```

The latency bridge can hold 1 beat in flight plus up to 4 beats in its skid buffer, which must be accounted for in the data availability count.

---

### Integration Example

```
sram_controller_unit #(
    .DATA_WIDTH      (512),
    .SRAM_DEPTH     (512),
    .SEG_COUNT_WIDTH(10)
) u_sram_controller_unit (
    .clk            (clk),
    .rst_n          (rst_n),

    // Allocation interface
    .axi_rd_alloc_req      (axi_rd_alloc_req[ch]),
    .axi_rd_alloc_size     (axi_rd_alloc_size),
    .axi_rd_alloc_space_free(axi_rd_alloc_space_free[ch]),

    // Write interface (from AXI Read Engine)
    .axi_rd_sram_valid    (axi_rd_sram_valid[ch]),
    .axi_rd_sram_ready    (axi_rd_sram_ready[ch]),
```

```

    .axi_rd_sram_data      (axi_rd_sram_data),
    // Drain interface
    .axi_wr_drain_data_avail(axi_wr_drain_data_avail[ch]),
    .axi_wr_drain_req       (axi_wr_drain_req[ch]),
    .axi_wr_drain_size      (axi_wr_drain_size[ch]),

    // Read interface (to AXI Write Engine)
    .axi_wr_sram_valid      (axi_wr_sram_valid[ch]),
    .axi_wr_sram_ready       (axi_wr_sram_ready[ch]),
    .axi_wr_sram_data        (axi_wr_sram_data[ch]),

    // Debug
    .dbg_bridge_pending     (dbg_bridge_pending[ch]),
    .dbg_bridge_out_valid   (dbg_bridge_out_valid[ch])
);

```

---

## Common Issues

### Issue 1: Space Accounting Mismatch

**Symptom:** Read engine sees space but FIFO overflows

**Root Causes:** 1. Allocation controller not receiving fulfillment signals 2. Output handshake not connected to allocation controller rd\_valid

**Solution:** Ensure axi\_wr\_sram\_valid && axi\_wr\_sram\_ready connects to allocation controller.

### Issue 2: Data Available Undercount

**Symptom:** Write engine stalls despite data in FIFO

**Root Causes:** 1. Bridge occupancy not included in data\_available calculation 2. Drain controller not receiving write handshakes

**Solution:** Verify axi\_wr\_drain\_data\_avail = drain\_data\_available + bridge\_occupancy.

---

## Related Documentation

- **Parent:** 08\_sram\_controller.md - Multi-channel SRAM controller
- **Allocation Controller:** 07\_stream\_alloc\_ctrl.md - Space tracking details

- **Drain Controller:** 11\_stream\_drain\_ctrl.md - Data tracking details
  - **Latency Bridge:** 10\_stream\_latency\_bridge.md - Timing compensation
  - **Read Engine:** 06\_axi\_read\_engine.md - Data producer
  - **Write Engine:** 12\_axi\_write\_engine.md - Data consumer
- 

**Last Updated:** 2025-11-30 (verified against RTL implementation)

## Stream Latency Bridge

**Module:** stream\_latency\_bridge.sv **Location:**

projects/components/stream/rtl/fub/ **Category:** FUB (Functional Unit Block)

**Parent:** sram\_controller\_unit.sv **Status:** Implemented **Last Updated:** 2025-11-30

---

### Overview

The stream\_latency\_bridge module is a simple latency-1 bridge that compensates for registered FIFO output latency. It uses glue logic plus a skid buffer to handle backpressure from downstream consumers.

### Key Features

- **1-Cycle Latency Compensation:** Bridges registered FIFO output to consumer
  - **Skid Buffer Architecture:** 4-deep FIFO absorbs consumer backpressure
  - **Full Throughput:** Maintains back-to-back transfers when consumer is ready
  - **Simple Design:** No complex ready calculations - skid buffer handles it all
  - **Occupancy Tracking:** Reports total beats held for flow control
-

## Architecture

### Block Diagram



*Stream Latency Bridge Block Diagram*

**Source:** [06\\_stream\\_latency\\_bridge\\_block.mmd](#)

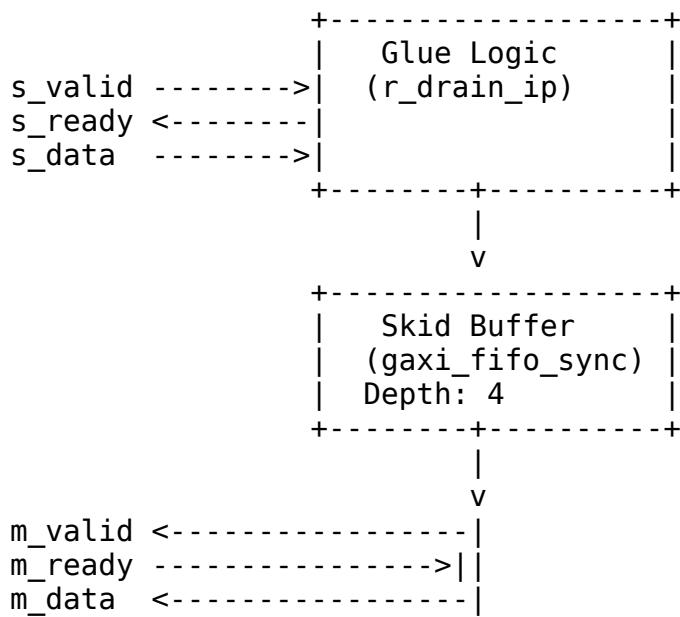
### Operation Flow

Cycle 0: Drain FIFO (`s_valid && skid_ready`)  $\Rightarrow r\_drain\_ip = 1$

Cycle 1: Data arrives from FIFO  $\Rightarrow$  Push to skid (`skid_valid = r_drain_ip`)

Cycle N: Consumer drains skid buffer at its own pace

### Component Diagram



## Backpressure Logic

The bridge uses a conservative backpressure approach:

- Track stalled writes (`wr_valid && !wr_ready`)
- Calculate pending count = `skid_count + stalled`
- Allow accept when pending < `SKID_DEPTH` OR consumer is draining

Truth Table (SKID\_DEPTH=4):

| count              | wr_valid | wr_ready | stalled | pending | room | s_ready |        |
|--------------------|----------|----------|---------|---------|------|---------|--------|
| 0                  | 0        | 1        | 0       | 0       | 1    | 1       | ?      |
| Empty              |          |          |         |         |      |         |        |
| 0                  | 1        | 1        | 0       | 0       | 1    | 1       | ?      |
| Writing, completes |          |          |         |         |      |         |        |
| 3                  | 0        | 1        | 0       | 3       | 1    | 1       | ? Room |
| for 1              |          |          |         |         |      |         |        |
| 3                  | 1        | 1        | 0       | 3       | 1    | 1       | ?      |
| Writing, will be 4 |          |          |         |         |      |         |        |
| 3                  | 1        | 0        | 1       | 4       | 0    | 0       | ?      |
| Write stalled      |          |          |         |         |      |         |        |
| 4                  | 0        | 0        | 0       | 4       | 0    | 0       | ? Full |

---

## Parameters

| Parameter               | Type | Default | Description                         |
|-------------------------|------|---------|-------------------------------------|
| <code>DATA_WIDTH</code> | int  | 64      | Data width in bits                  |
| <code>SKID_DEPTH</code> | int  | 4       | Skid buffer depth (2-4 recommended) |

## Derived Parameters

| Parameter | Derivation              | Description |
|-----------|-------------------------|-------------|
| DW        | <code>DATA_WIDTH</code> | Short alias |

## Port List

### Clock and Reset

| Signal             | Direction | Width | Description             |
|--------------------|-----------|-------|-------------------------|
| <code>clk</code>   | input     | 1     | System clock            |
| <code>rst_n</code> | input     | 1     | Active-low asynchronous |

| Signal | Direction | Width | Description |
|--------|-----------|-------|-------------|
|        |           |       | reset       |

#### Upstream Interface (from registered FIFO)

| Signal  | Direction | Width | Description                                    |
|---------|-----------|-------|------------------------------------------------|
| s_valid | input     | 1     | Upstream valid (FIFO not empty)                |
| s_ready | output    | 1     | Ready to accept from FIFO                      |
| s_data  | input     | DW    | Data from FIFO (valid 1 cycle after handshake) |

#### Downstream Interface (to consumer)

| Signal  | Direction | Width | Description            |
|---------|-----------|-------|------------------------|
| m_valid | output    | 1     | Data valid to consumer |
| m_ready | input     | 1     | Consumer ready         |
| m_data  | output    | DW    | Data to consumer       |

#### Status Interface

| Signal    | Direction | Width | Description                                    |
|-----------|-----------|-------|------------------------------------------------|
| occupancy | output    | 3     | Beats in bridge (0-5: 1 in flight + 4 in skid) |

#### Debug Interface

| Signal           | Direction | Width | Description                 |
|------------------|-----------|-------|-----------------------------|
| dbg_r_pending    | output    | 1     | Data in flight (r_drain_ip) |
| dbg_r_out_val_id | output    | 1     | Output valid (m_valid)      |

## Operation

### Glue Logic

The glue logic consists of a single flop (`r_drain_ip`) that tracks when a FIFO drain is in progress:

```
// Drain FIFO when upstream has data AND we can accept
wire w_drain_fifo = s_valid && s_ready;

// Flop the drain signal to track data in flight
always_ff @(posedge clk or negedge rst_n) begin
    if (!rst_n) begin
        r_drain_ip <= 1'b0;
    end else begin
        r_drain_ip <= w_drain_fifo;
    end
end

// When r_drain_ip asserted, data arrived from FIFO @ push to skid
assign skid_wr_valid = r_drain_ip;
assign skid_wr_data = s_data;
```

### Skid Buffer

A 4-deep `gaxi_fifo_sync` instance with: - Non-registered mode (REGISTERED=0) for minimum latency - Auto memory style selection

### Occupancy Calculation

```
// Total occupancy = data in skid buffer
// Note: r_drain_ip not counted separately (conservative)
assign occupancy = skid_count;
```

---

## Integration Example

```
stream_latency_bridge #(
    .DATA_WIDTH (512),
    .SKID_DEPTH (4)
) u_latency_bridge (
    .clk           (clk),
    .rst_n         (rst_n),

    // Slave (from FIFO)
    .s_data        (fifo_rd_data),
    .s_valid       (fifo_rd_valid),
    .s_ready       (fifo_rd_ready),
```

```

// Master (to AXI Write Engine)
.m_data      (axi_wr_sram_data),
.m_valid     (axi_wr_sram_valid),
.m_ready     (axi_wr_sram_ready),

// Status
.occupancy   (bridge_occupancy),

// Debug
.dbg_r_pending (dbg_bridge_pending),
.dbg_r_out_valid(dbg_bridge_out_valid)
);

```

---

## Design Rationale

### Why Use a Skid Buffer?

Previous designs attempted complex ready signal calculations to handle backpressure. This led to:

- Timing issues from deep combinatorial paths
- Edge cases with multi-cycle backpressure
- Difficult debugging of flow control bugs

The skid buffer approach:

- Simple glue logic (single flop)
- FIFO handles all backpressure complexity
- Proven, reusable component
- Easy to verify and debug

### Why 4-Deep?

- **2-deep:** Minimum for back-to-back (1 in flight + 1 buffered)
- **4-deep:** Allows 2-3 cycles of consumer stall without upstream backpressure
- **8-deep:** Overkill for most use cases

4-deep provides good balance between:

- Latency tolerance (handles brief stalls)
- Resource usage (small FIFO)
- Throughput (back-to-back capable)

---

## Related Documentation

- **Parent:** 09\_sram\_controller\_unit.md - Single-channel SRAM controller
  - **Consumer:** 12\_axi\_write\_engine.md - Receives bridge output
  - **Producer:** 06\_axi\_read\_engine.md - Feeds upstream FIFO
- 

**Last Updated:** 2025-11-30 (verified against RTL implementation)

# Stream Drain Controller

**Module:** stream\_drain\_ctrl.sv **Location:**

projects/components/stream/rtl/fub/ **Category:** FUB (Functional Unit Block)

**Parent:** sram\_controller\_unit.sv **Status:** Implemented **Last Updated:** 2025-11-30

---

## Overview

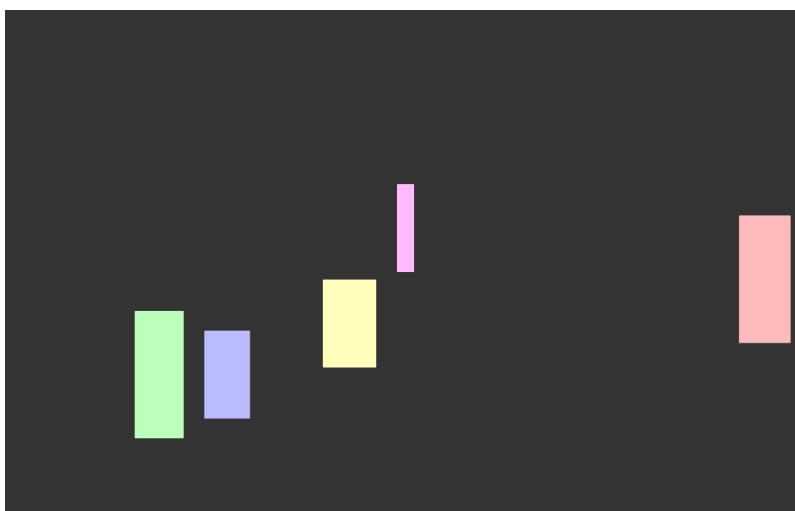
The `stream_drain_ctrl` module is a virtual FIFO for write engine flow control. It tracks data availability using FIFO pointer logic without storing any data. This allows the write engine to pre-reserve data before issuing AXI write commands.

## Key Features

- **Virtual FIFO Pattern:** Pointer arithmetic only, no data storage
  - **Pre-Reservation:** Write engine reserves data before AXI AW command
  - **Variable Size Drains:** Supports burst-sized reservations (not single beat)
  - **Registered Outputs:** Optional output registration for timing
  - **Underflow Prevention:** Ensures data exists before write engine commits
- 

## Architecture

### Block Diagram



*Stream Drain Controller Block Diagram*

**Source:** 09\_stream\_drain\_ctrl\_block.mmd

## Virtual FIFO Concept

Unlike a real FIFO that stores data, the drain controller only tracks pointers:

Real FIFO:                    Virtual FIFO (Drain Controller):

|         |             |                        |
|---------|-------------|------------------------|
| Data[0] | wr_ptr: 5   | Tracks entries written |
| Data[1] | rd_ptr: 2   | Tracks entries drained |
| Data[2] | available:3 | Difference             |
| ...     |             |                        |

Actual data stored in gaxi\_fifo\_sync (separate component)

## Pointer Operations

### Write Side (Data Entry):

When FIFO receives data:

```
wr_ptr += 1  
data_available += 1
```

### Read Side (Drain Request):

When write engine requests drain:

```
rd_ptr += rd_size (variable burst size!)  
data_available -= rd_size
```

---

## Parameters

| Parameter            | Type | Default | Description         |
|----------------------|------|---------|---------------------|
| DEPTH                | int  | 512     | Virtual FIFO depth  |
| ALMOST_WR_MAR<br>GIN | int  | 1       | Almost full margin  |
| ALMOST_RD_MAR<br>GIN | int  | 1       | Almost empty margin |
| REGISTERED           | int  | 1       | Registered outputs  |

## Derived Parameters

| Parameter | Derivation | Description   |
|-----------|------------|---------------|
| D         | DEPTH      | Short alias   |
| AW        | \$clog2(D) | Address width |

## Port List

### Clock and Reset

| Signal      | Direction | Width | Description                   |
|-------------|-----------|-------|-------------------------------|
| axi_aclk    | input     | 1     | System clock                  |
| axi_aresetn | input     | 1     | Active-low asynchronous reset |

### Write Interface (Data Entry)

| Signal   | Direction | Width | Description           |
|----------|-----------|-------|-----------------------|
| wr_valid | input     | 1     | Data written to FIFO  |
| wr_ready | output    | 1     | Not full (can accept) |

### Read Interface (Drain Requests)

| Signal   | Direction | Width | Description                |
|----------|-----------|-------|----------------------------|
| rd_valid | input     | 1     | Request to drain data      |
| rd_size  | input     | 8     | Number of entries to drain |
| rd_ready | output    | 1     | Data available (not empty) |

### Status Outputs

| Signal         | Direction | Width | Description          |
|----------------|-----------|-------|----------------------|
| data_available | output    | AW+1  | Available data count |
| wr_full        | output    | 1     | Full (no space)      |

| Signal          | Direction | Width | Description     |
|-----------------|-----------|-------|-----------------|
| wr_almost_full  | output    | 1     | Almost full     |
| rd_empty        | output    | 1     | Empty (no data) |
| rd_almost_empty | output    | 1     | Almost empty    |

## Operation

### Data Entry (Write Side)

Data entry is single-beat increments, tracking each beat written to the physical FIFO:

```
// Write pointer uses counter_bin for single-beat increments
counter_bin #(
    .WIDTH (AW + 1),
    .MAX   (D)
) write_pointer_inst (
    .clk           (axi_aclk),
    .rst_n         (axi_aresetn),
    .enable        (w_write && !r_wr_full),
    .counter_bin_curr (r_wr_ptr_bin),
    .counter_bin_next (w_wr_ptr_bin_next)
);
```

### Drain Requests (Read Side)

Drain requests support variable-size bursts:

```
// Read pointer advances by rd_size on each drain request
always_ff @(posedge axi_aclk or negedge axi_aresetn) begin
    if (!axi_aresetn) begin
        r_rd_ptr_bin <= '0;
    end else begin
        if (w_read && !r_rd_empty) begin
            r_rd_ptr_bin <= r_rd_ptr_bin + (AW+1)'(rd_size);
        end
    end
end
```

### Status Generation

Uses fifo\_control block for full/empty/almost flags:

```

fifo_control #(
    .DEPTH           (D),
    .ADDR_WIDTH     (AW),
    .ALMOST_RD_MARGIN (ALMOST_RD_MARGIN),
    .ALMOST_WR_MARGIN (ALMOST_WR_MARGIN),
    .REGISTERED     (REGISTERED)
) fifo_control_inst (
    // ... pointer inputs
    .count          (w_count),
    .wr_full        (r_wr_full),
    .wr_almost_full (r_wr_almost_full),
    .rd_empty       (r_rd_empty),
    .rd_almost_empty (r_rd_almost_empty)
);

```

---

## Comparison with stream\_alloc\_ctrl

| Aspect            | stream_alloc_ctrl         | stream_drain_ctrl        |
|-------------------|---------------------------|--------------------------|
| <b>Purpose</b>    | Track space for reads     | Track data for writes    |
| <b>Write Side</b> | Reservation (burst)       | Data entry (single beat) |
| <b>Read Side</b>  | Fulfillment (single beat) | Drain request (burst)    |
| <b>Consumer</b>   | AXI Read Engine           | AXI Write Engine         |

Both use virtual FIFO pattern but with opposite semantics for read/write operations.

---

## Integration Example

```

stream_drain_ctrl #(
    .DEPTH           (512),
    .REGISTERED     (1)
) u_drain_ctrl (
    .axi_aclk        (clk),
    .axi_arstn      (rst_n),

    // Write interface: Connected to FIFO write handshake
    .wr_valid        (fifo_wr_valid && fifo_wr_ready),
    .wr_ready        (), // Not used (tracks, doesn't control)

    // Read interface: Connected to write engine drain requests

```

```
.rd_valid          (axi_wr_drain_req),
.rd_size           (axi_wr_drain_size),
.rd_ready          (), // Not used (polling interface)

// Status
.data_available   (drain_data_available),
.wr_full          (),
.wr_almost_full   (),
.rd_empty          (),
.rd_almost_empty   ()

);
```

---

## Common Issues

### Issue 1: Data Available Undercount

**Symptom:** Write engine sees 0 available when FIFO has data

**Root Causes:** 1. wr\_valid not connected to FIFO write handshake 2. Bridge occupancy not added to data\_available

**Solution:** Ensure connection: wr\_valid = fifo\_wr\_valid && fifo\_wr\_ready

### Issue 2: Drain Request Overflow

**Symptom:** rd\_ptr advances past wr\_ptr

**Root Causes:** 1. Write engine requesting more than available 2. Multiple simultaneous drain requests

**Solution:** Write engine must check data\_available >= requested\_size before draining.

---

## Related Documentation

- **Parent:** 09\_sram\_controller\_unit.md - Integration context
  - **Counterpart:** 07\_stream\_alloc\_ctrl.md - Space tracking (read side)
  - **Consumer:** 12\_axi\_write\_engine.md - Uses drain interface
  - **Control Block:** See fifo\_control in rtl/amba/gaxi/
- 

**Last Updated:** 2025-11-30 (verified against RTL implementation)

# AXI Write Engine

**Module:** axi\_write\_engine.sv **Location:**

projects/components/stream/rtl/fub/ **Category:** FUB (Functional Unit Block)

**Parent:** stream\_core.sv **Status:** Implemented **Last Updated:** 2025-11-30

---

## Overview

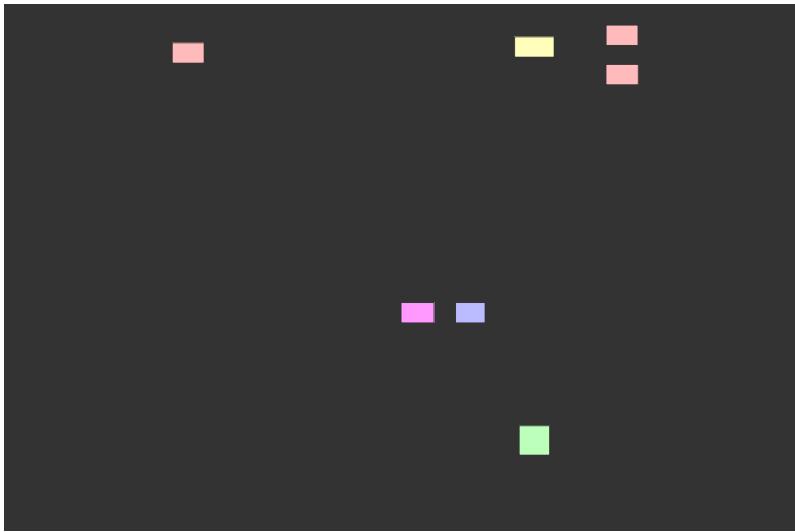
The axi\_write\_engine module is a high-performance multi-channel AXI4 write engine with data-aware arbitration. It serves all 8 STREAM channels through a single AXI master interface with intelligent flow control.

## Key Features

- **Round-Robin Arbitration:** Fair scheduling across channels
  - **Data-Aware Masking:** Only arbitrate channels with sufficient SRAM data
  - **Pre-Drain Handshake:** Reserve SRAM data before AW command issues
  - **FIFO-Based W Tracking:** Single shared FIFO for W-phase order preservation
  - **Per-Channel B Tracking:** Separate FIFOs for out-of-order B responses
  - **Channel ID in AXI ID:** Enables per-channel response routing
  - **Pipelined/Non-Pipelined Modes:** Configurable outstanding transaction depth
-

## Architecture

### Block Diagram



*AXI Write Engine Block Diagram*

**Source:** [10\\_axi\\_write\\_engine\\_block.mmd](#)

### Operation Flow

1. Scheduler Interface: Each channel can request write bursts
2. Data Checking: Mask channels without sufficient SRAM data
3. Arbitration: Round-robin arbiter selects next channel to service
4. AXI AW Issue: Issue write command to AXI, assert wr\_drain to reserve data
5. W-Phase FIFO: Push transaction to shared FIFO (preserves AW order)
6. AXI W Stream: Stream write data from SRAM controller (ID-based mux)
7. AXI B Response: Pop per-channel B-phase FIFO, complete transaction

### Key Design Decisions

**FIFO-Based W Tracking (No FSM):** Instead of a state machine, uses a single shared FIFO to track pending W-phase transactions. This preserves the AW command order which is critical for W-phase correctness.

**Separate W and B Phase FIFOs:** - **W-Phase FIFO:** Single shared FIFO (in-order with AW) - **B-Phase FIFOs:** Per-channel FIFOs (out-of-order responses allowed by AXI4)

**ID-Based SRAM Interface:** The write engine drives `axi_wr_sram_id` to select which channel's data to drain. The SRAM controller muxes the appropriate channel's data to `axi_wr_sram_data`.

---

## Parameters

| Parameter          | Type | Default | Description                                  |
|--------------------|------|---------|----------------------------------------------|
| NUM_CHANNELS       | int  | 8       | Number of channels                           |
| ADDR_WIDTH         | int  | 64      | AXI address width                            |
| DATA_WIDTH         | int  | 512     | AXI data width                               |
| ID_WIDTH           | int  | 8       | AXI ID width                                 |
| USER_WIDTH         | int  | 8       | AXI USER width                               |
| SEG_COUNT_WIDTH    | int  | 8       | Width of space/count signals                 |
| PIPELINE           | int  | 0       | 0: non-pipelined, 1: pipelined               |
| AW_MAX_OUTSTANDING | int  | 8       | Maximum outstanding AW requests (PIPELINE=1) |
| W_PHASE_FIFO_DEPTH | int  | 64      | W-phase transaction FIFO depth               |
| B_PHASE_FIFO_DEPTH | int  | 16      | B-phase transaction FIFO depth per channel   |

## Derived Parameters

| Parameter | Derivation      | Description                  |
|-----------|-----------------|------------------------------|
| NC        | NUM_CHANNELS    | Short alias                  |
| AW        | ADDR_WIDTH      | Short alias                  |
| DW        | DATA_WIDTH      | Short alias                  |
| IW        | ID_WIDTH        | Short alias                  |
| UW        | USER_WIDTH      | Short alias                  |
| SCW       | SEG_COUNT_WIDTH | Segment count width          |
| CIW       | \$clog2(NC)     | Channel ID width (min 1 bit) |
| AXSIZE    | \$clog2(DW/8)   | AXI burst size               |

## Port List

### Clock and Reset

| Signal | Direction | Width | Description                   |
|--------|-----------|-------|-------------------------------|
| clk    | input     | 1     | System clock                  |
| rst_n  | input     | 1     | Active-low asynchronous reset |

### Configuration Interface

| Signal                | Direction | Width | Description                           |
|-----------------------|-----------|-------|---------------------------------------|
| cfg_axi_wr_xfer_beats | input     | 8     | Transfer size in beats (all channels) |

### Scheduler Interface (Per-Channel)

| Signal                 | Direction | Width   | Description                        |
|------------------------|-----------|---------|------------------------------------|
| sched_wr_val_id[ch]    | input     | NC      | Channel requests write             |
| sched_wr_ready[ch]     | output    | NC      | Engine ready (descriptor complete) |
| sched_wr_addr[ch]      | input     | NC x AW | Destination addresses              |
| sched_wr_beats[ch]     | input     | NC x 32 | Beats remaining to write           |
| sched_wr_burst_len[ch] | input     | NC x 8  | Requested burst length             |

### Completion Interface (Per-Channel)

| Signal                   | Direction | Width   | Description                     |
|--------------------------|-----------|---------|---------------------------------|
| sched_wr_done_strobe[ch] | output    | NC      | Burst completed (1 cycle pulse) |
| sched_wr_beats_done[ch]  | output    | NC x 32 | Number of beats completed       |

### SRAM Drain Interface

| Signal                | Direction | Width  | Description             |
|-----------------------|-----------|--------|-------------------------|
| axi_wr_drain_req[ch]  | output    | NC     | Request to reserve data |
| axi_wr_drain_size[ch] | output    | NC x 8 | Beats to reserve        |

| Signal                      | Direction | Width    | Description                |
|-----------------------------|-----------|----------|----------------------------|
| axi_wr_drain_data_avail[ch] | input     | NC x SCW | Data available per channel |

### SRAM Read Interface

| Signal                | Direction | Width | Description                      |
|-----------------------|-----------|-------|----------------------------------|
| axi_wr_sram_valid[ch] | input     | NC    | Per-channel valid                |
| axi_wr_sram_drain     | output    | 1     | Drain request                    |
| axi_wr_sram_id        | output    | CIW   | Channel ID select                |
| axi_wr_sram_data      | input     | DW    | Muxed data from selected channel |

### AXI4 AW Channel

| Signal        | Direction | Width | Description             |
|---------------|-----------|-------|-------------------------|
| m_axi_awvalid | output    | 1     | Address valid           |
| m_axi_awready | input     | 1     | Address ready           |
| m_axi_awid    | output    | IW    | Transaction ID          |
| m_axi_awaddr  | output    | AW    | Address                 |
| m_axi_awlen   | output    | 8     | Burst length - 1        |
| m_axi_awsize  | output    | 3     | Burst size (log2 bytes) |
| m_axi_awburst | output    | 2     | Burst type (INCR)       |

### AXI4 W Channel

| Signal       | Direction | Width | Description      |
|--------------|-----------|-------|------------------|
| m_axi_wvalid | output    | 1     | Write data valid |
| m_axi_wready | input     | 1     | Write data ready |
| m_axi_wdata  | output    | DW    | Write data       |
| m_axi_wstrb  | output    | DW/8  | Write strobes    |

| Signal      | Direction | Width | Description             |
|-------------|-----------|-------|-------------------------|
| m_axi_wlast | output    | 1     | Last beat of burst      |
| m_axi_wuser | output    | UW    | Channel ID for tracking |

### AXI4 B Channel

| Signal       | Direction | Width | Description    |
|--------------|-----------|-------|----------------|
| m_axi_bvalid | input     | 1     | Response valid |
| m_axi_bready | output    | 1     | Response ready |
| m_axi_bid    | input     | IW    | Transaction ID |
| m_axi_bresp  | input     | 2     | Response       |

### Error Interface

| Signal                    | Direction | Width | Description                   |
|---------------------------|-----------|-------|-------------------------------|
| sched_wr_error[<br>r[ch]] | output    | NC    | Sticky error flag per channel |

### Debug Interface

| Signal                     | Direction | Width | Description                  |
|----------------------------|-----------|-------|------------------------------|
| dbg_wr_all_completions[ch] | output    | NC    | All writes complete          |
| dbg_aw_transactions        | output    | 32    | Total AW transactions issued |
| dbg_w_beats                | output    | 32    | Total W beats written to AXI |

## Operation

### Data-Aware Request Masking

```
// Only request arbitration if:
// 1. Scheduler is requesting (sched_wr_valid)
// 2. Sufficient SRAM data available (w_data_ok)
// 3. Below outstanding limit (w_no_outstanding)
w_arb_request[i] = sched_wr_valid[i] && w_data_ok[i] &&
```

```
w_no_outstanding[i];

// Data check includes final burst handling
w_data_ok[i] = w_has_data[i] || w_final_burst[i];
```

## W-Phase Transaction FIFO

Single shared FIFO preserves AW command order:

```
typedef struct packed {
    logic [7:0] beats;           // Number of beats for this W
    logic [CIW-1:0] channel_id; // Channel ID for this transaction
} w_phase_txn_t;

// Push on AW handshake (preserves order)
if (m_axi_awvalid && m_axi_awready) begin
    w_phase_txn_fifo_wr = 1'b1;
    w_phase_txn_fifo_din.beats = m_axi_awlen + 8'd1;
    w_phase_txn_fifo_din.channel_id = r_aw_channel_id;
end
```

## B-Phase Transaction FIFOs

Per-channel FIFOs track beats and last flag for completion:

```
typedef struct packed {
    logic [7:0] beats;           // Number of beats in this transaction
    logic       last;            // Is this the last transfer for
descriptor?
} b_phase_txn_t;

// last flag determines when sched_wr_ready asserts
if (m_axi_bvalid && m_axi_bready) begin
    if (b_phase_txn_fifo_dout[ch_id].last) begin
        r_sched_ready[ch_id] <= 1'b1;
    end
end
```

## Outstanding Transaction Tracking

**PIPELINE=0 (Non-Pipelined):** - Boolean flag per channel (0 or 1 outstanding) -  
Set when AW issues, clear when B arrives

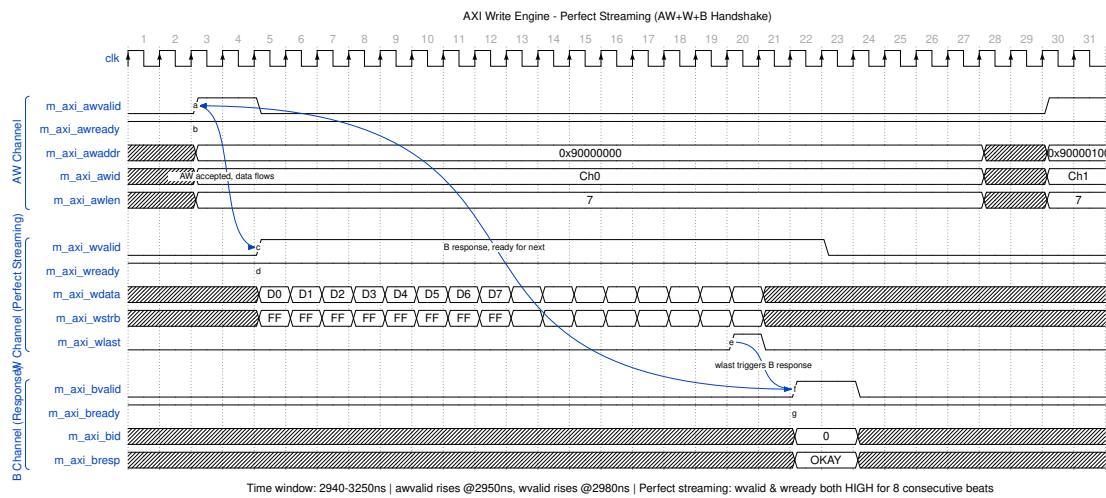
**PIPELINE=1 (Pipelined):** - Counter per channel (0 to AW\_MAX\_OUTSTANDING) -  
Increment on AW, decrement on B

---

# Timing Diagrams

## Perfect Streaming - AXI Write Transaction

The following timing diagram shows the AXI write engine operating at maximum throughput with **perfect streaming** - where both wvalid and wready remain HIGH for consecutive clock cycles, achieving one data beat per cycle.



## AXI Write Engine - Perfect Streaming

### Transaction Flow:

- 1. AW Channel Handshake (Address Phase)**
  - `m_axi_awvalid` rises to initiate a write request
  - `m_axi_awready` is already HIGH (slave ready)
  - Address (`awaddr`), ID (`awid`), and length (`awlen=7` for 8 beats) are captured
  - Handshake completes in a single cycle when both valid and ready are HIGH
  - Simultaneously, `axi_wr_drain_req` reserves data from SRAM controller
- 2. W Channel Streaming (Data Phase)**
  - After AW acceptance, the engine begins streaming data from SRAM
  - `m_axi_wvalid` rises and **stays HIGH** for all 8 beats
  - `m_axi_wready` remains HIGH throughout (no backpressure from AXI slave)
  - Perfect streaming:** One data beat transferred every clock cycle
  - `m_axi_wlast` rises on the final beat (beat 7) to mark burst completion

- `wstrb` is all-ones (0xFF) indicating full data width valid

### 3. B Channel Response (Completion Phase)

- After `wlast`, the AXI slave returns a write response
- `m_axi_bvalid` rises with `bid` matching the transaction ID
- `m_axi_bready` is HIGH (engine always ready for responses)
- `bresp=OKAY` indicates successful write
- Response triggers `sched_wr_done_strobe` to notify scheduler

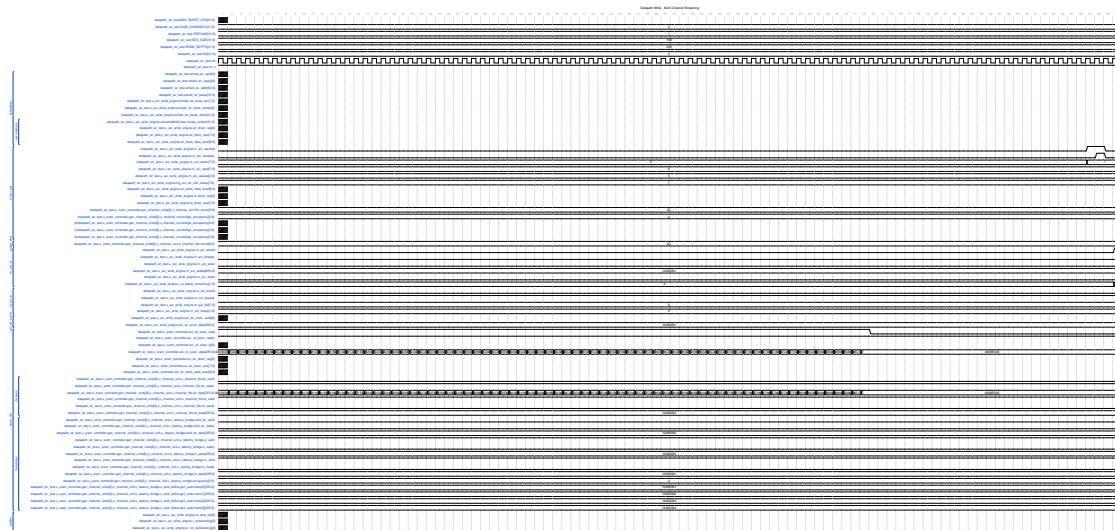
### 4. Next Transaction

- After B response, the engine can immediately issue the next AW request
- The cycle repeats for the next channel or next burst

**Key Performance Indicators:** - **No bubbles:** `wvalid` and `wready` both HIGH during data phase - **Full bandwidth:** Data width (256/512 bits) x clock frequency - **Zero-wait states:** SRAM controller provides data at line rate - **Minimal latency:** B response arrives shortly after `wlast`

## Multi-Channel Streaming

For multi-channel operation showing channel switching while maintaining streaming performance, see:



## Datapath Write - Multi-Channel

This diagram shows how the engine arbitrates between channels while maintaining high throughput. Note how:

- Different channel IDs (`awid`) appear in sequence
- W-phase FIFO preserves AW order for correct data association
- B responses can arrive out-of-order (per AXI4 spec)

---

## Integration Example

```
axi_write_engine #(
    .NUM_CHANNELS      (8),
    .ADDR_WIDTH        (64),
    .DATA_WIDTH        (512),
    .ID_WIDTH          (8),
    .SEG_COUNT_WIDTH   (10),
    .PIPELINE           (0),
    .AW_MAX_OUTSTANDING (8),
    .W_PHASE_FIFO_DEPTH (64),
    .B_PHASE_FIFO_DEPTH (16)
) u_axi_write_engine (
    .clk                (clk),
    .rst_n              (rst_n),
    // Configuration
    .cfg_axi_wr_xfer_beats (cfg_axi_wr_xfer_beats),
    // Scheduler interface
    .sched_wr_valid      (sched_wr_valid),
    .sched_wr_ready       (sched_wr_ready),
    .sched_wr_addr        (sched_wr_addr),
    .sched_wr_beats       (sched_wr_beats),
    .sched_wr_burst_len   (sched_wr_burst_len),
    // Completion interface
    .sched_wr_done_strobe (sched_wr_done_strobe),
    .sched_wr_beats_done  (sched_wr_beats_done),
    // SRAM drain interface
    .axi_wr_drain_req     (axi_wr_drain_req),
    .axi_wr_drain_size    (axi_wr_drain_size),
    .axi_wr_drain_data_avail(axi_wr_drain_data_avail),
    // SRAM read interface
    .axi_wr_sram_valid     (axi_wr_sram_valid),
    .axi_wr_sram_drain    (axi_wr_sram_drain),
    .axi_wr_sram_id        (axi_wr_sram_id),
    .axi_wr_sram_data      (axi_wr_sram_data),
    // AXI master
    .m_axi_awvalid        (m_axi_wr_awvalid),
    .m_axi_awready         (m_axi_wr_awready),
    .m_axi_awid            (m_axi_wr_awid),
    .m_axi_awaddr          (m_axi_wr_awaddr),
    .m_axi_awlen            (m_axi_wr_awlen),
```

```

.m_axi_awsize          (m_axi_wr_awsize),
.m_axi_awburst         (m_axi_wr_awburst),

.m_axi_wvalid          (m_axi_wr_wvalid),
.m_axi_wready          (m_axi_wr_wready),
.m_axi_wdata           (m_axi_wr_wdata),
.m_axi_wstrb          (m_axi_wr_wstrb),
.m_axi_wlast           (m_axi_wr_wlast),
.m_axi_wuser           (m_axi_wr_wuser),

.m_axi_bvalid          (m_axi_wr_bvalid),
.m_axi_bready          (m_axi_wr_bready),
.m_axi_bid              (m_axi_wr_bid),
.m_axi_bresp            (m_axi_wr_bresp),

// Error and debug
.sched_wr_error         (sched_wr_error),
.dbg_wr_all_complete   (dbg_wr_all_complete),
.dbg_aw_transactions   (dbg_aw_transactions),
.dbg_w_beats            (dbg_w_beats)
);

```

---

## Related Documentation

- **Parent:** 01\_stream\_core.md - Top-level integration
  - **Scheduler Array:** 02\_scheduler\_group\_array.md - Provides sched\_wr\_\* signals
  - **SRAM Controller:** 08\_sram\_controller.md - Data source
  - **Drain Controller:** 11\_stream\_drain\_ctrl.md - Data availability tracking
  - **Read Engine:** 06\_axi\_read\_engine.md - Complementary read datapath
- 

**Last Updated:** 2025-12-13 (added timing diagrams)

## APB to Descriptor Router

**Module:** apbtodescr.sv **Location:** projects/components/stream/rtl/fub/  
**Category:** FUB (Functional Unit Block) **Parent:** stream\_top.sv **Status:**  
 Implemented **Last Updated:** 2025-11-30

---

## Overview

The apbtodescr module is an APB-to-descriptor engine router that handles address-based routing of APB writes to descriptor engine kick-off ports. It decodes the APB address to determine the target channel and routes the 64-bit descriptor address using a two-write sequence.

## Key Features

- **Address-Based Routing:** APB address bits select target channel
  - **64-bit Address Assembly:** Two 32-bit APB writes combine into 64-bit descriptor address
  - **Backpressure Handling:** Delays APB response if descriptor engine busy
  - **Address Range Checking:** Error on out-of-range addresses
  - **Integration Signal:** apb\_descriptor\_kickoff\_hit for response muxing
- 

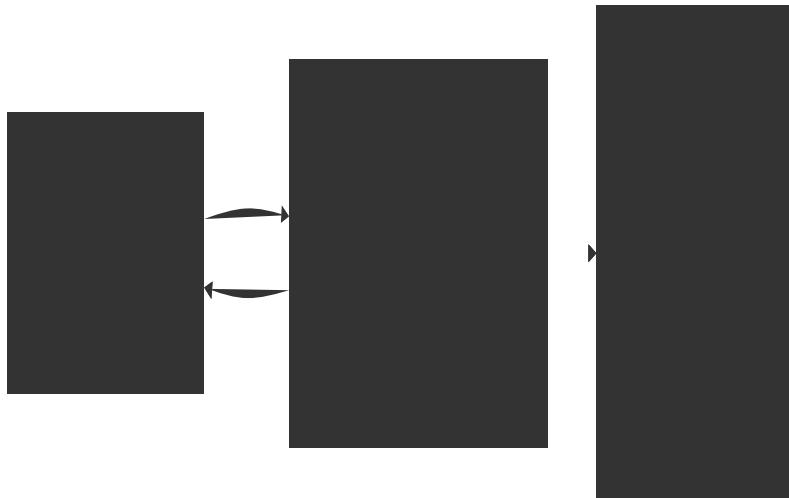
## Architecture

### Address Map

Relative to BASE\_ADDR:

|              |                                   |         |
|--------------|-----------------------------------|---------|
| BASE + 0x00: | Channel 0 descriptor address LOW  | [31:0]  |
| BASE + 0x04: | Channel 0 descriptor address HIGH | [63:32] |
| BASE + 0x08: | Channel 1 descriptor address LOW  | [31:0]  |
| BASE + 0x0C: | Channel 1 descriptor address HIGH | [63:32] |
| ...          |                                   |         |
| BASE + 0x38: | Channel 7 descriptor address LOW  | [31:0]  |
| BASE + 0x3C: | Channel 7 descriptor address HIGH | [63:32] |

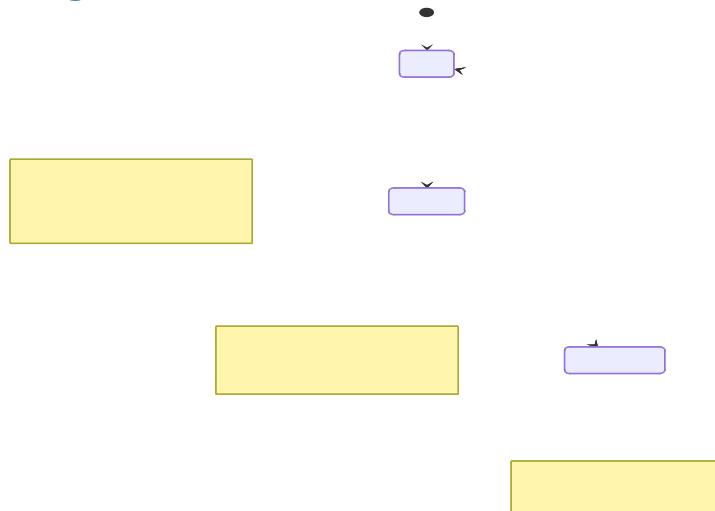
## Block Diagram



*APB to Descriptor Block Diagram*

**Source:** [02\\_apbtodescr\\_block.mmd](#)

## FSM Diagram



*APB to Descriptor FSM*

**Source:** [02\\_apbtodescr\\_fsm.mmd](#)

## Write Flow

1. Software writes LOW word to CHx\_CTRL\_LOW register
2. PeakRDL APB slave presents write on cmd/rsp interface
3. Module captures LOW word, responds with OKAY
4. Software writes HIGH word to CHx\_CTRL\_HIGH register
5. Module assembles 64-bit address from LOW + HIGH

6. Module asserts desc\_apb\_valid[channel\_id]
7. Module drives desc\_apb\_addr[channel\_id]
8. Waits for desc\_apb\_ready[channel\_id]
9. Completes APB transaction (asserts apb\_rsp\_valid)

### FSM States

```
typedef enum logic [2:0] {
    IDLE      = 3'b000,      // Waiting for APB command (LOW write)
    RESPOND_LOW = 3'b001,    // Sending response after LOW write
    WAIT_HIGH  = 3'b010,    // Waiting for HIGH write
    ROUTE      = 3'b011,    // Routing to descriptor engine
    RESPOND_HIGH = 3'b100   // Sending final response after HIGH
write
} state_t;
```

---

### Parameters

| Parameter    | Type | Default | Description            |
|--------------|------|---------|------------------------|
| ADDR_WIDTH   | int  | 32      | APB address width      |
| DATA_WIDTH   | int  | 32      | APB data width         |
| NUM_CHANNELS | int  | 8       | Number of DMA channels |

### Port List

#### Clock and Reset

| Signal | Direction | Width | Description                   |
|--------|-----------|-------|-------------------------------|
| clk    | input     | 1     | System clock                  |
| rst_n  | input     | 1     | Active-low asynchronous reset |

#### APB Slave CMD Interface

| Signal        | Direction | Width | Description     |
|---------------|-----------|-------|-----------------|
| apb_cmd_valid | input     | 1     | Command valid   |
| apb_cmd_ready | output    | 1     | Ready to accept |

| Signal        | Direction | Width      | Description                          |
|---------------|-----------|------------|--------------------------------------|
|               |           |            | command                              |
| apb_cmd_addr  | input     | ADDR_WIDTH | Command address                      |
| apb_cmd_wdata | input     | DATA_WIDTH | Write data                           |
| apb_cmd_write | input     | 1          | Write enable<br>(1=write,<br>0=read) |

### APB Slave RSP Interface

| Signal        | Direction | Width      | Description             |
|---------------|-----------|------------|-------------------------|
| apb_rsp_valid | output    | 1          | Response valid          |
| apb_rsp_ready | input     | 1          | Response ready          |
| apb_rsp_rdata | output    | DATA_WIDTH | Read data<br>(always 0) |
| apb_rsp_error | output    | 1          | Error flag              |

### Descriptor Engine APB Ports

| Signal                | Direction | Width                 | Description                 |
|-----------------------|-----------|-----------------------|-----------------------------|
| desc_apb_val_id[ch]   | output    | NUM_CHAN<br>NELS      | Per-channel valid           |
| desc_apb_ready_dy[ch] | input     | NUM_CHAN<br>NELS      | Per-channel ready           |
| desc_apb_addr[r[ch]]  | output    | NUM_CHAN<br>NELS x 64 | 64-bit descriptor addresses |

### Integration Control

| Signal                       | Direction | Width | Description                    |
|------------------------------|-----------|-------|--------------------------------|
| apb_descript or_kickoff_hi t | output    | 1     | Indicates kick-off in progress |

## Operation

### Address Decode

```
// Extract channel ID from address (dword-aligned: addr[5:3])
// Address bits [1:0] are ignored (word-aligned)
// Address bit [2] selects LOW (0) or HIGH (1) register
// Address bits [5:3] select channel 0-7
assign channel_id = apb_cmd_addr[5:3];
assign r_is_high_write = apb_cmd_addr[2];

// Valid range: 0x00 to 0x3F (8 channels x 8 bytes)
assign addr_in_range = ({20'h0, apb_cmd_addr[11:0]} < (NUM_CHANNELS * 8));
```

### 64-bit Address Assembly

```
// LOW write captured in IDLE state
if (r_state == IDLE && apb_cmd_valid) begin
    r_wdata_low <= apb_cmd_wdata;
end

// HIGH write captured in WAIT_HIGH state
if (r_state == WAIT_HIGH && apb_cmd_valid) begin
    r_wdata_high <= apb_cmd_wdata;
end

// Assembled 64-bit address
desc_apb_addr[ch] = {r_wdata_high, r_wdata_low};
```

### Error Conditions

1. **Read Operation:** APB reads not supported
2. **Address Out of Range:** Beyond channel address space
3. **HIGH Before LOW:** HIGH write without preceding LOW write
4. **Wrong Channel:** HIGH write to different channel than LOW

---

## Integration Example

```
apbtodescr #(
    .ADDR_WIDTH      (32),
    .DATA_WIDTH      (32),
    .NUM_CHANNELS    (8)
) u_apbtodescr (
    .clk                  (clk),
    .rst_n                (rst_n),
    // APB CMD/RSP from PeakRDL
    .apb_cmd_valid        (apb_kickoff_cmd_valid),
```

```

.apb_cmd_ready          (apb_kickoff_cmd_ready),
.apb_cmd_addr           (apb_kickoff_cmd_addr),
.apb_cmd_wdata           (apb_kickoff_cmd_wdata),
.apb_cmd_write           (apb_kickoff_cmd_write),

.apb_rsp_valid           (apb_kickoff_rsp_valid),
.apb_rsp_ready           (apb_kickoff_rsp_ready),
.apb_rsp_rdata           (apb_kickoff_rsp_rdata),
.apb_rsp_error           (apb_kickoff_rsp_error),

// To descriptor engines
.desc_apb_valid          (desc_apb_valid),
.desc_apb_ready           (desc_apb_ready),
.desc_apb_addr            (desc_apb_addr),

// Integration
.apb_descriptor_kickoff_hit (kickoff_hit)
);

```

---

## Common Issues

### Issue 1: APB Response Not Returning

**Symptom:** APB write hangs without response

**Root Causes:** 1. Descriptor engine not asserting ready 2. FSM stuck in ROUTE state 3. apb\_rsp\_ready not being asserted

**Solution:** Check descriptor engine acceptance and FSM state.

### Issue 2: Wrong Channel Kicked Off

**Symptom:** Different channel starts than expected

**Root Cause:** HIGH write sent to different channel than LOW

**Solution:** Ensure software writes LOW then HIGH to same channel address.

---

## Related Documentation

- **Parent:** 01\_stream\_core.md - Top-level integration
  - **Consumer:** 05\_descriptor\_engine.md - Receives kick-off
  - **Register File:** PeakRDL-generated stream\_regs.sv
-

**Last Updated:** 2025-11-30 (verified against RTL implementation)

## APB Configuration Block

**Module:** stream\_config\_block.sv **Location:**

projects/components/stream/rtl/top/ **Category:** TOP (Integration) **Parent:** stream\_top.sv **Status:** Implemented **Last Updated:** 2025-11-30

---

### Overview

The stream\_config\_block module maps PeakRDL-generated register outputs to STREAM core configuration inputs. It provides a clean interface layer between the register file and the STREAM DMA engine core, handling field extraction, width conversion, and global enable gating.

### Key Features

- **Field Extraction:** Extracts fields from packed register values
  - **Width Conversion:** Converts register fields to configuration signal widths
  - **Global Enable Gating:** Gates enables by global enable register
  - **Address Extension:** Zero-extends 32-bit register addresses to 64-bit
  - **Clean Interface:** Isolates register naming from core logic
-

# Architecture

## Block Diagram



*Stream Config Block Diagram*

**Source:** [14\\_stream\\_config\\_block.mmd](#)

### Configuration Groups

```
stream_config_block
  Global and Channel Control
    cfg_channel_enable (gated by global_en)
    cfg_channel_reset (OR'd with global_rst)
  Scheduler Configuration
    cfg_sched_enable (gated by global_en)
    cfg_sched_timeout_cycles
    cfg_sched_*_enable flags
  Descriptor Engine Configuration
    cfg_desceng_enable (gated by global_en)
    cfg_desceng_prefetch
    cfg_desceng_addr*_base/limit (zero-extended)
  Monitor Configurations (3 sets)
    Descriptor AXI Monitor (cfg_desc_mon_*)
    Read Engine Monitor (cfg_rdeng_mon_*)
    Write Engine Monitor (cfg_wreng_mon_*)
  AXI Transfer Configuration
    cfg_axi_rd_xfer_beats
    cfg_axi_wr_xfer_beats
  Performance Profiler
    cfg_perf_enable (gated by global_en)
    cfg_perf_mode
    cfg_perf_clear
```

---

## Parameters

| Parameter    | Type | Default | Description                      |
|--------------|------|---------|----------------------------------|
| NUM_CHANNELS | int  | 8       | Number of DMA channels           |
| ADDR_WIDTH   | int  | 64      | Address width for config outputs |

## Port List

### Clock and Reset

| Signal | Direction | Width | Description                   |
|--------|-----------|-------|-------------------------------|
| clk    | input     | 1     | System clock                  |
| rst_n  | input     | 1     | Active-low asynchronous reset |

### PeakRDL Register Inputs

#### Global Control:

| Signal                   | Direction | Width | Description             |
|--------------------------|-----------|-------|-------------------------|
| reg_global_ct            | input     | 1     | Global enable           |
| rl_global_en             |           |       |                         |
| reg_global_ct            | input     | 1     | Global reset            |
| rl_global_rst            |           |       |                         |
| reg_channel_enable_ch_en | input     | 8     | Per-channel enable bits |
| reg_channel_reset_ch_rst | input     | 8     | Per-channel reset bits  |

#### Scheduler Configuration:

| Signal                     | Direction | Width | Description       |
|----------------------------|-----------|-------|-------------------|
| reg_sched_timeout_cycles_* | input     | 16    | Timeout threshold |
| reg_sched_config_*         | input     | 1     | Various enables   |

## Descriptor Engine Configuration:

| Signal                | Direction | Width  | Description          |
|-----------------------|-----------|--------|----------------------|
| reg_desceng_c_onfig_* | input     | varies | Enable and threshold |
| reg_desceng_a_ddr*_*  | input     | 32     | Address range limits |

## Monitor Configurations (3 sets):

Each monitor (DAXMON, RDMON, WRMON) has: - reg\_\*\_enable\_\* - Enable flags - reg\_\*\_timeout\_\* - Timeout configuration - reg\_\*\_latency\_thresh\_\* - Latency threshold - reg\_\*\_pkt\_mask\_\* - Packet type mask - reg\_\*\_err\_cfg\_\* - Error configuration - reg\_\*\_mask\*\_\* - Event masks

## AXI Transfer Configuration:

| Signal                             | Direction | Width | Description      |
|------------------------------------|-----------|-------|------------------|
| reg_axi_xfer_config_rd_xfe_r_beats | input     | 8     | Read burst size  |
| reg_axi_xfer_config_wr_xfe_r_beats | input     | 8     | Write burst size |

## Performance Profiler:

| Signal                      | Direction | Width | Description     |
|-----------------------------|-----------|-------|-----------------|
| reg_perf_conf ig_perf_en    | input     | 1     | Profiler enable |
| reg_perf_conf ig_perf_mode  | input     | 1     | Profiling mode  |
| reg_perf_conf ig_perf_clear | input     | 1     | Clear profiler  |

## Configuration Outputs

### Global and Channel:

| Signal             | Direction | Width         | Description             |
|--------------------|-----------|---------------|-------------------------|
| cfg_channel_enable | output    | NUM_CHAN NELS | Gated channel enables   |
| cfg_channel_reset  | output    | NUM_CHAN      | Combined channel resets |

| Signal | Direction | Width | Description |
|--------|-----------|-------|-------------|
| NELS   |           |       |             |

### Scheduler:

| Signal                   | Direction | Width | Description            |
|--------------------------|-----------|-------|------------------------|
| cfg_sched_enable         | output    | 1     | Gated scheduler enable |
| cfg_sched_timeout_cycles | output    | 16    | Timeout value          |
| cfg_sched_*_enable       | output    | 1     | Feature enables        |

### Descriptor Engine:

| Signal                  | Direction | Width     | Description             |
|-------------------------|-----------|-----------|-------------------------|
| cfg_desceng_enable      | output    | 1         | Gated desceng enable    |
| cfg_desceng_prefetch    | output    | 1         | Prefetch enable         |
| cfg_desceng_fifo_thresh | output    | 4         | FIFO threshold          |
| cfg_desceng_addr*_base  | output    | ADDR_WIDT | Zero-extended address H |
| cfg_desceng_addr*_limit | output    | ADDR_WIDT | Zero-extended address H |

### Monitor Outputs (3 sets):

Each monitor outputs ~16 configuration signals covering: - Enable flags - Timeout values - Thresholds - Masks

### AXI Transfer:

| Signal                | Direction | Width | Description      |
|-----------------------|-----------|-------|------------------|
| cfg_axi_rd_xfer_beats | output    | 8     | Read burst size  |
| cfg_axi_wr_xfer_beats | output    | 8     | Write burst size |

### Performance Profiler:

| Signal          | Direction | Width | Description           |
|-----------------|-----------|-------|-----------------------|
| cfg_perf_enable | output    | 1     | Gated profiler enable |
| cfg_perf_mode   | output    | 1     | Profiling mode        |
| cfg_perf_clear  | output    | 1     | Clear signal          |

## Operation

### Global Enable Gating

```
// Gate all channel enables by global enable
assign cfg_channel_enable = reg_channel_enable_ch_en &
{NUM_CHANNELS{reg_global_ctrl_global_en}};

// Gate scheduler enable
assign cfg_sched_enable = reg_sched_config_sched_en &
reg_global_ctrl_global_en;

// Gate monitor enables
assign cfg_desc_mon_enable = reg_daxmon_enable_mon_en &
reg_global_ctrl_global_en;
```

### Global Reset OR

```
// Channel resets are OR'd with global reset
assign cfg_channel_reset = reg_channel_reset_ch_RST | 
{NUM_CHANNELS{reg_global_ctrl_global_RST}};
```

### Address Zero Extension

```
// Zero-extend 32-bit register addresses to ADDR_WIDTH (typically 64-bit)
assign cfg_desceng_addr0_base = {{(ADDR_WIDTH-32){1'b0}}, 
reg_desceng_addr0_base_addr0_base};
assign cfg_desceng_addr0_limit = {{(ADDR_WIDTH-32){1'b0}}, 
reg_desceng_addr0_limit_addr0_limit};
```

---

## Integration Example

```
stream_config_block #(
    .NUM_CHANNELS      (8),
    .ADDR_WIDTH        (64)
) u_config_block (
    .clk                (clk),
    .rst_n              (rst_n),
```

```

// From PeakRDL register file
.reg_global_ctrl_global_en          (regs_global_ctrl_global_en),
.reg_global_ctrl_global_RST          (regs_global_ctrl_global_RST),
.reg_channel_enable_ch_en           (regs_channel_enable_ch_en),
// ... more register inputs

// To stream_core
.cfg_channel_enable                 (cfg_channel_enable),
.cfg_channel_RESET                  (cfg_channel_RESET),
.cfg_sched_enable                   (cfg_sched_enable),
// ... more config outputs
);

```

---

## Related Documentation

- **Parent:** 01\_stream\_core.md - Receives configuration signals
  - **Register File:** PeakRDL-generated stream\_regs.sv
  - **Scheduler:** 04\_scheduler.md - Uses sched\_\* config
  - **Descriptor Engine:** 05\_descriptor\_engine.md - Uses desceng\_\* config
  - **Monitors:** Monitor documentation for each AXI monitor
- 

**Last Updated:** 2025-11-30 (verified against RTL implementation)

## Performance Profiler

**Module:** perf\_profiler.sv **Location:** projects/components/stream/rtl/fub/  
**Category:** FUB (Functional Unit Block) **Parent:** stream\_top.sv **Status:**  
 Implemented **Last Updated:** 2025-11-30

---

## Overview

The perf\_profiler module captures timing information for channel activity to enable performance analysis and bottleneck identification. It monitors channel idle signals and records either timestamps or elapsed times to a FIFO for software retrieval.

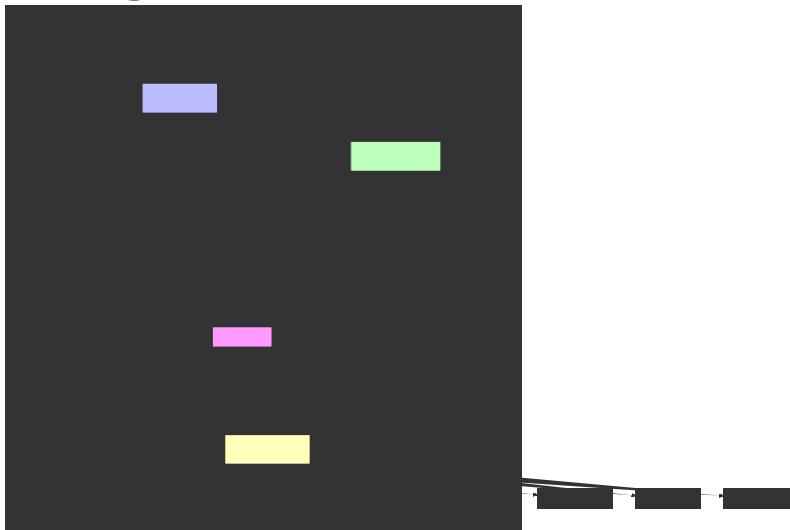
## Key Features

- **Two Profiling Modes:**
  - Mode 0 (TIMESTAMP): Captures timestamps on idle signal edges

- Mode 1 (ELAPSED): Captures elapsed time directly
  - **Per-Channel Profiling:** Tracks all 8 channels independently
  - **256-Entry FIFO:** Buffers performance data for software polling
  - **Channel ID Tagging:** Each entry includes source channel
  - **Configurable via APB:** Enable, mode, and clear controls
- 

## Architecture

### Block Diagram



*Performance Profiler Block Diagram*

**Source:** [15\\_perf\\_profiler\\_block.mmd](#)

### Profiling Modes

#### Mode 0 - TIMESTAMP:

Records timestamp when:

- Channel transitions idle  $\rightarrow$  active (start)
- Channel transitions active  $\rightarrow$  idle (end)

Software calculates:  $\text{elapsed} = \text{timestamp\_end} - \text{timestamp\_start}$

Advantages:

- Simpler hardware
- More flexible analysis
- Can detect partial transfers

#### Mode 1 - ELAPSED:

Records elapsed time when:  
 - Channel returns to idle

`elapsed_time = current_timestamp - start_timestamp`

Advantages:

- Hardware calculates duration
- Simpler software processing
- Direct performance metric

### FIFO Entry Format (36-bit)

| Bits    | Field             | Description                        |
|---------|-------------------|------------------------------------|
| [31:0]  | timestamp/elapsed | Timestamp or elapsed time          |
| [34:32] | channel_id        | Source channel (0-7)               |
| [35]    | event_type        | 0=start, 1=end<br>(timestamp mode) |

### Parameters

| Parameter       | Type | Default                        | Description                   |
|-----------------|------|--------------------------------|-------------------------------|
| NUM_CHANNELS    | int  | 8                              | Number of channels to monitor |
| CHANNEL_WIDTH   | int  | $\$clog2(\text{NUM_CHANNELS})$ | Channel ID width              |
| TIMESTAMP_WIDTH | int  | 32                             | Timestamp counter width       |
| FIFO_DEPTH      | int  | 256                            | Performance FIFO depth        |
| FIFO_ADDR_WIDTH | int  | $\$clog2(\text{FIFO_DEPTH})$   | FIFO address width            |

### Port List

#### Clock and Reset

| Signal | Direction | Width | Description  |
|--------|-----------|-------|--------------|
| clk    | input     | 1     | System clock |

| Signal | Direction | Width | Description                   |
|--------|-----------|-------|-------------------------------|
| rst_n  | input     | 1     | Active-low asynchronous reset |

### Channel Monitoring

| Signal       | Direction | Width        | Description              |
|--------------|-----------|--------------|--------------------------|
| channel_idle | input     | NUM_CHANNELS | Per-channel idle signals |

### Configuration Interface

| Signal     | Direction | Width | Description               |
|------------|-----------|-------|---------------------------|
| cfg_enable | input     | 1     | Enable profiling          |
| cfg_mode   | input     | 1     | 0=timestamp,<br>1=elapsed |
| cfg_clear  | input     | 1     | Clear FIFO and counters   |

### FIFO Read Interface

| Signal              | Direction | Width | Description                     |
|---------------------|-----------|-------|---------------------------------|
| perf_fifo_rd        | input     | 1     | Read strobe (pops FIFO)         |
| perf_fifo_data_low  | output    | 32    | Timestamp/elapsed [31:0]        |
| perf_fifo_data_high | output    | 32    | {28'b0, event_type, channel_id} |
| perf_fifo_empty     | output    | 1     | FIFO empty flag                 |
| perf_fifo_full      | output    | 1     | FIFO full flag                  |
| perf_fifo_count     | output    | 16    | Number of entries               |

## Operation

### Edge Detection

```
// Detect idle signal transitions
assign w_idle_rising = channel_idle & ~r_idle_prev; // Active @ Idle
```

```

(end)
assign w_idle_falling = ~channel_idle & r_idle_prev; // Idle → Active
(start)

```

### Timestamp Counter

```

// Free-running 32-bit counter
// Increments every cycle when profiling enabled
// Wraps at 2^32 - 1 (software handles rollover)
always_ff @(posedge clk or negedge rst_n) begin
    if (!rst_n) begin
        r_timestamp_counter <= '0;
    end else if (cfg_clear) begin
        r_timestamp_counter <= '0;
    end else if (cfg_enable) begin
        r_timestamp_counter <= r_timestamp_counter + 1'b1;
    end
end

```

### Start Time Capture (Elapsed Mode)

```

// Per-channel start time tracking
generate
    for (ch = 0; ch < NUM_CHANNELS; ch++) begin
        always_ff @(posedge clk) begin
            if (cfg_enable && cfg_mode == MODE_ELAPSED) begin
                if (w_idle_falling[ch]) begin
                    r_start_time[ch] <= r_timestamp_counter;
                    r_channel_active[ch] <= 1'b1;
                end
                else if (w_idle_rising[ch]) begin
                    r_channel_active[ch] <= 1'b0;
                end
            end
        end
    end
endgenerate

```

### Two-Register Read Sequence

Software reads FIFO via APB:

1. Check perf\_fifo\_empty == 0
  2. Read PERF\_FIFO\_DATA\_LOW:
    - Triggers perf\_fifo\_rd strobe
    - Pops FIFO, latches 36-bit entry
    - Returns timestamp/elapsed [31:0]
  3. Read PERF\_FIFO\_DATA\_HIGH:
    - Returns {28'b0, event\_type, channel\_id}
    - No FIFO pop
  4. Parse data and repeat
-

## Integration Example

```
perf_profiler #(
    .NUM_CHANNELS      (8),
    .TIMESTAMP_WIDTH   (32),
    .FIFO_DEPTH        (256)
) u_perf_profiler (
    .clk                (clk),
    .rst_n              (rst_n),

    // From schedulers
    .channel_idle       (scheduler_idle),

    // Configuration
    .cfg_enable         (cfg_perf_enable),
    .cfg_mode           (cfg_perf_mode),
    .cfg_clear          (cfg_perf_clear),

    // FIFO read interface (to APB)
    .perf_fifo_rd       (perf_fifo_rd),
    .perf_fifo_data_low (perf_fifo_data_low),
    .perf_fifo_data_high(perf_fifo_data_high),
    .perf_fifo_empty    (perf_fifo_empty),
    .perf_fifo_full     (perf_fifo_full),
    .perf_fifo_count    (perf_fifo_count)
);
```

---

## Software Usage

### Example: Measure Channel 0 Transfer Time

```
// Enable timestamp mode
write_reg(PERF_CONFIG, PERF_EN | MODE_TIMESTAMP);

// Start transfer on channel 0
write_reg(CH0_CTRL_LOW, desc_addr_low);
write_reg(CH0_CTRL_HIGH, desc_addr_high);

// Wait for completion
while (!read_reg(CH0_STATUS) & COMPLETE);

// Read performance data
while (!(read_reg(PERF_STATUS) & FIFO_EMPTY)) {
    uint32_t low = read_reg(PERF_FIFO_DATA_LOW);    // Pops FIFO
    uint32_t high = read_reg(PERF_FIFO_DATA_HIGH);   // Reads latched

    uint32_t timestamp = low;
```

```
uint8_t channel    = high & 0x7;
uint8_t event      = (high >> 3) & 0x1;

if (channel == 0) {
    if (event == 0) // Start
        start_time = timestamp;
    else           // End
        elapsed = timestamp - start_time;
}
}
```

---

## Common Issues

### Issue 1: FIFO Overflow

**Symptom:** Performance data lost

**Root Cause:** Software not polling fast enough

**Solution:** - Poll more frequently - Use larger FIFO depth - Consider interrupt on half-full

### Issue 2: Timestamp Rollover

**Symptom:** Negative elapsed times calculated

**Root Cause:** 32-bit counter wrapped during measurement

**Solution:** Handle rollover in software:

```
if (end_time < start_time)
    elapsed = (0xFFFFFFFF - start_time) + end_time + 1;
```

---

## Related Documentation

- **Parent:** 01\_stream\_core.md - Top-level integration
  - **Data Source:** 04\_scheduler.md - Provides idle signals
  - **Configuration:** 14\_apb\_config.md - Config register mapping
- 

**Last Updated:** 2025-11-30 (verified against RTL implementation)

# MonBus AXI-Lite Group

**Module:** monbus\_axil\_group.sv **Location:**

projects/components/stream/rtl/macro/ **Category:** MACRO (Integration)

**Parent:** stream\_top.sv **Status:** Implemented **Last Updated:** 2025-11-30

---

## Overview

The monbus\_axil\_group module receives monitor bus packets from STREAM channels, applies configurable filtering, and routes filtered packets to either an error/interrupt FIFO (accessible via AXI-Lite slave) or a master write interface (writes to memory).

## Key Features

- **Single Monitor Bus Input:** STREAM is memory-to-memory (no network paths)
- **Per-Protocol Filtering:** Configurable packet type masking
- **Dual Output Paths:**
  - Error/Interrupt FIFO - generates interrupt when not empty
  - Master Write FIFO - writes to configurable address range
- **Protocol Support:** AXI, AXIS, CORE (3 protocols)
- **Built-in AXI-Lite Skid Buffering:** For timing closure

## Simplified from RAPIDS

RAPIDS has source + sink data paths requiring TWO monitor buses. STREAM has memory-to-memory only, so this module has ONE monitor bus input (no arbitration needed).

---

## Architecture

### Block Diagram



*MonBus AXI-Lite Group Block Diagram*

**Source:** [16\\_monbus\\_axil\\_group\\_block.mmd](#)

### Filter Decision Tree

For each incoming packet:

1. Extract: pkt\_type, pkt\_protocol, pkt\_event\_code
  2. Check protocol-specific masks:
    - pkt\_mask: Drop if bit[pkt\_type] = 1
    - err\_select: Route to error FIFO if bit[pkt\_type] = 1
    - event\_mask: Drop if bit[event\_code] = 1
  3. Route:
    - Dropped packets: Consumed silently
    - Error-selected: To error/interrupt FIFO
    - Remaining: To master write FIFO
- 

### Parameters

| Parameter            | Type | Default | Description                |
|----------------------|------|---------|----------------------------|
| FIFO_DEPTH_ER<br>R   | int  | 64      | Error/interrupt FIFO depth |
| FIFO_DEPTH_WR<br>ITE | int  | 32      | Master write FIFO depth    |
| ADDR_WIDTH           | int  | 32      | AXI address width          |

| Parameter     | Type | Default | Description        |
|---------------|------|---------|--------------------|
| DATA_WIDTH    | int  | 32      | AXI data width     |
| NUM_PROTOCOLS | int  | 3       | AXI, AXIS,<br>CORE |

## Port List

### Clock and Reset

| Signal      | Direction | Width | Description             |
|-------------|-----------|-------|-------------------------|
| axi_aclk    | input     | 1     | AXI clock               |
| axi_aresetn | input     | 1     | AXI active-low<br>reset |

### Monitor Bus Input

| Signal        | Direction | Width | Description              |
|---------------|-----------|-------|--------------------------|
| monbus_valid  | input     | 1     | Packet valid             |
| monbus_ready  | output    | 1     | Ready to<br>accept       |
| monbus_packet | input     | 64    | 64-bit monitor<br>packet |

### AXI-Lite Slave Read Interface

| Signal           | Direction | Width      | Description           |
|------------------|-----------|------------|-----------------------|
| s_axil_arval_i_d | input     | 1          | Read address<br>valid |
| s_axil_arready_y | output    | 1          | Read address<br>ready |
| s_axil_araddr    | input     | ADDR_WIDTH | Read address          |
| s_axil_arprot    | input     | 3          | Protection bits       |
| s_axil_rvalid    | output    | 1          | Read data<br>valid    |
| s_axil_rready    | input     | 1          | Read data<br>ready    |
| s_axil_rdata     | output    | DATA_WIDTH | Read data             |
| s_axil_rresp     | output    | 2          | Read response         |

### AXI-Lite Master Write Interface

| Signal           | Direction | Width        | Description          |
|------------------|-----------|--------------|----------------------|
| m_axil_awvali_d  | output    | 1            | Write address valid  |
| m_axil_awready_y | input     | 1            | Write address ready  |
| m_axil_awaddr    | output    | ADDR_WIDTH   | Write address        |
| m_axil_awprot    | output    | 3            | Protection bits      |
| m_axil_wvalid    | output    | 1            | Write data valid     |
| m_axil_wready    | input     | 1            | Write data ready     |
| m_axil_wdata     | output    | DATA_WIDTH   | Write data           |
| m_axil_wstrb     | output    | DATA_WIDTH/8 | Write strobes        |
| m_axil_bvalid    | input     | 1            | Write response valid |
| m_axil_bready    | output    | 1            | Write response ready |
| m_axil_bresp     | input     | 2            | Write response       |

### Interrupt Output

| Signal  | Direction | Width | Description                      |
|---------|-----------|-------|----------------------------------|
| irq_out | output    | 1     | Interrupt (error FIFO not empty) |

### Configuration Interface

| Signal         | Direction | Width      | Description             |
|----------------|-----------|------------|-------------------------|
| cfg_base_addr  | input     | ADDR_WIDTH | Base for master writes  |
| cfg_limit_addr | input     | ADDR_WIDTH | Limit for master writes |

### Protocol Configuration (per protocol)

AXI Protocol:

| Signal             | Direction | Width | Description                |
|--------------------|-----------|-------|----------------------------|
| cfg_axi_pkt_mask   | input     | 16    | Drop mask for packet types |
| cfg_axi_err_select | input     | 16    | Error FIFO select          |
| cfg_axi_*_mask     | input     | 16    | Per-event-type masks       |

### AXIS Protocol:

| Signal              | Direction | Width | Description          |
|---------------------|-----------|-------|----------------------|
| cfg_axis_pkt_mask   | input     | 16    | Drop mask            |
| cfg_axis_err_select | input     | 16    | Error FIFO select    |
| cfg_axis_*_mask     | input     | 16    | Per-event-type masks |

### CORE Protocol:

| Signal              | Direction | Width | Description          |
|---------------------|-----------|-------|----------------------|
| cfg_core_pkt_mask   | input     | 16    | Drop mask            |
| cfg_core_err_select | input     | 16    | Error FIFO select    |
| cfg_core_*_mask     | input     | 16    | Per-event-type masks |

### Debug/Status

| Signal           | Direction | Width | Description      |
|------------------|-----------|-------|------------------|
| err_fifo_full    | output    | 1     | Error FIFO full  |
| write_fifo_full  | output    | 1     | Write FIFO full  |
| err_fifo_count   | output    | 8     | Error FIFO count |
| write_fifo_count | output    | 8     | Write FIFO count |

## Operation

### Packet Type Filtering

```
// Protocol-specific filtering (example: AXI)
case (pkt_protocol)
    3'b000: begin // AXI
        // Check if packet type is dropped
        pkt_drop = cfg_axi_pkt_mask[pkt_type];

        // Check if packet goes to error FIFO
        pkt_to_err_fifo = cfg_axi_err_select[pkt_type] && !pkt_drop;

        // Check individual event masking
        case (pkt_type)
            PktTypeError:     pkt_event_masked =
                cfg_axi_error_mask[pkt_event_code];
            PktTypeTimeout:   pkt_event_masked =
                cfg_axi_timeout_mask[pkt_event_code];
            // ... more packet types
        endcase
    end
endcase
```

### Master Write Address Generation

```
// Address wraps within configured range
always_comb begin
    next_write_addr = current_write_addr + (DATA_WIDTH == 64 ? 8 : 4);
    if (next_write_addr > cfg_limit_addr) begin
        next_write_addr = cfg_base_addr;
    end
end
```

### Master Write FSM

```
typedef enum logic [2:0] {
    WRITE_IDLE      = 3'b000,
    WRITE_ADDR      = 3'b001,
    WRITE_DATA_LOW  = 3'b010,
    WRITE_DATA_HIGH = 3'b011, // For 32-bit data width only
    WRITE RESP      = 3'b100
} write_state_t;
```

---

## Integration Example

```
monbus_axil_group #(
    .FIFO_DEPTH_ERR      (64),
    .FIFO_DEPTH_WRITE    (32),
    .ADDR_WIDTH          (32),
    .DATA_WIDTH          (32),
```

```

    .NUM_PROTOCOLS      (3)
) u_monbus_axil_group (
    .axi_aclk          (clk),
    .axi_aresetn        (rst_n),

    // Monitor bus input
    .monbus_valid      (stream_mon_valid),
    .monbus_ready      (stream_mon_ready),
    .monbus_packet     (stream_mon_packet),

    // AXI-Lite slave read (for error FIFO access)
    .s_axil_arvalid    (axil_slave_arvalid),
    .s_axil_arready    (axil_slave_arready),
    // ... more slave signals

    // AXI-Lite master write (for logging)
    .m_axil_awvalid    (axil_master_awvalid),
    .m_axil_awready    (axil_master_awready),
    // ... more master signals

    // Interrupt
    .irq_out           (monbus_interrupt),

    // Configuration
    .cfg_base_addr     (cfg_monbus_base_addr),
    .cfg_limit_addr    (cfg_monbus_limit_addr),
    .cfg_axi_pkt_mask  (cfg_axi_pkt_mask),
    // ... more protocol configuration
);

```

---

## Common Issues

### Issue 1: Missing Monitor Packets

**Symptom:** Expected events not appearing in FIFO

**Root Causes:** 1. Packet type masked in `cfg_pkt_mask` 2. Event masked in `cfg_*_mask` 3. FIFO full (packets dropped)

**Solution:** Check mask configuration and FIFO status.

### Issue 2: Master Writes Stall

**Symptom:** Write FIFO fills up, backpressure to monitor bus

**Root Causes:** 1. AXI-Lite master target not responding 2. Address range exhausted (wrapping too fast)

**Solution:** Increase address range or reduce logging rate via masks.

---

## Related Documentation

- **Parent:** 01\_stream\_core.md - Provides monitor bus packets
  - **Packet Format:** Monitor Bus Protocol documentation
  - **Configuration:** 14\_apb\_config.md - Register mapping
- 

**Last Updated:** 2025-11-30 (verified against RTL implementation)

## Chapter 3: External Interfaces

This chapter documents the external interfaces of the STREAM DMA engine (stream\_top\_ch8.sv).

### Interface Specifications

STREAM has seven primary external interfaces:

#### 01\_axi4\_interface\_spec.md

- **AXI4 Master Interfaces (3 total)**
  - Descriptor fetch (256-bit, read-only)
  - Data read (parameterizable, default 512-bit)
  - Data write (parameterizable, default 512-bit)
- Protocol specifications and assumptions
- Transfer modes and alignment requirements
- Timing and performance characteristics

#### 02\_axil4\_interface\_spec.md

- **AXI4-Lite Interfaces (2 total, in monbus\_axil\_group)**
  - Error FIFO slave read (s\_axil\_err\_\*): 32-bit, read-only
  - Monitor data master write (m\_axil\_mon\_\*): 32-bit, write-only
- Used for monitor bus packet access and logging
- Part of monbus\_axil\_group integration (USE\_AXI\_MONITORS=1)

## [03\\_apb\\_interface\\_spec.md](#)

- **APB4 Slave Interface**
- Full APB4 protocol with PSEL/PENABLE
- Optional CDC for asynchronous clock domains (CDC\_ENABLE parameter)
- Address map: 0x000-0x03F kick-off, 0x100-0x3FF registers

## [05\\_monbus\\_interface\\_spec.md](#)

- **MonBus Internal Protocol**
  - Unified 64-bit monitoring bus (stream\_core → monbus\_axil\_group)
  - Event packet format and encoding
  - Agent ID assignments for STREAM components
  - Converted to AXI-Lite at top level
- 

## [STREAM Interface Summary \(stream\\_top\\_ch8.sv\)](#)

| Interface          | Type         | Direction | Width                    | Purpose                          |
|--------------------|--------------|-----------|--------------------------|----------------------------------|
| APB4               | Slave        | Input     | 32-bit data, 12-bit addr | Configuration and kick-off       |
| AXI4 Descriptor    | Master Read  | Output    | 256-bit                  | Descriptor fetch                 |
| AXI4 Data Read     | Master Read  | Output    | 512-bit (param)          | Source data read                 |
| AXI4 Data Write    | Master Write | Output    | 512-bit (param)          | Destination data write           |
| AXIL Error FIFO    | Slave Read   | Input     | 32-bit                   | Monitor error/interrupt FIFO     |
| AXIL Monitor Write | Master Write | Output    | 32-bit                   | Monitor data to memory           |
| IRQ                | Output       | Output    | 1-bit                    | Interrupt (error FIFO not empty) |

**Note:** AXIL interfaces are active when USE\_AXI\_MONITORS=1. MonBus is internal (stream\_core → monbus\_axil\_group).

---

**Last Updated:** 2025-12-01

**Dependencies:** - AMBA AXI4 Protocol Specification v4.0 - AMBA Monitor Bus Protocol (internal spec) - STREAM Architecture Overview

## AXI4 Interface Specification for STREAM

### Overview

This document defines the AXI4 interface specification for the STREAM DMA engine. STREAM uses three AXI4 master interfaces for memory access:

1. **Descriptor Fetch Master** - 256-bit read-only interface for fetching descriptors
2. **Data Read Master** - Parameterizable width (default 512-bit) for reading source data
3. **Data Write Master** - Parameterizable width (default 512-bit) for writing destination data

**Note:** This document focuses on STREAM-specific implementation details. For generic AXI4 protocol information, refer to the AMBA AXI4 Protocol Specification v4.0.

---

## STREAM AXI4 Interface Summary

### Number of Interfaces

STREAM implements **3 AXI4 Master Interfaces**:

| Interface               | Type         | Width              | Channels | Purpose                        |
|-------------------------|--------------|--------------------|----------|--------------------------------|
| <b>Descriptor Fetch</b> | Master Read  | 256-bit<br>(fixed) | AR, R    | Fetch descriptors from memory  |
| <b>Data Read</b>        | Master Read  | 512-bit<br>(param) | AR, R    | Read source data to SRAM       |
| <b>Data Write</b>       | Master Write | 512-bit<br>(param) | AW, W, B | Write SRAM data to destination |

## 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                             |
|--------------------------------|----------|-------------------|-------------------|---------|-----|-----------------------------------|
| <b>AXI4 Master Read-Split</b>  | AR, R    | <b>Flexible</b>   | 4-byte            | Yes     | Yes | Dat a interface with chunk enable |
| <b>AXI4 Master Write-Split</b> | AW, W, B | <b>Flexible</b>   | 4-byte            | Yes     | Yes | Dat a interface with chunk enable |
| <b>AXI4</b>                    | AR, R    | <b>Simplified</b> | Bus-width         | No      | Yes | Con                               |

| Interface Group          | Channels    | Transfer Mode     | Address Alignment | Monitor | DCG | Notes                                                          |
|--------------------------|-------------|-------------------|-------------------|---------|-----|----------------------------------------------------------------|
| <b>Master Read</b>       |             |                   |                   |         |     | trol<br>inte<br>rfac<br>es,<br>full<br>y<br>alig<br>ned        |
| <b>AXI4 Master Write</b> | AW,<br>W, B | <b>Simplified</b> | Bus-width         | Yes     | Yes | Con<br>trol<br>inte<br>rfac<br>es,<br>full<br>y<br>alig<br>ned |

### Interface Group Parameter Settings

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

### Interface Configuration Summary

| Interface Type         | Interface Group         | Transfer Mode     | Alignment      | Notes                 |
|------------------------|-------------------------|-------------------|----------------|-----------------------|
| <b>Descriptor Sink</b> | <b>AXI4 Master Read</b> | <b>Simplified</b> | 32-bit aligned | Contr<br>ol<br>interf |

| Interface Type           | Interface Group                | Transfer Mode     | Alignment      | Notes               |
|--------------------------|--------------------------------|-------------------|----------------|---------------------|
| <b>Descriptor Source</b> | <b>AXI4 Master Read</b>        | <b>Simplified</b> | 32-bit aligned | Control interface   |
| <b>Data Source</b>       | <b>AXI4 Master Read-Split</b>  | <b>Flexible</b>   | 4-byte aligned | High-bandwidth data |
| <b>Data Sink</b>         | <b>AXI4 Master Write-Split</b> | <b>Flexible</b>   | 4-byte aligned | High-bandwidth data |
| <b>Program Sink</b>      | <b>AXI4 Master Write</b>       | <b>Simplified</b> | 32-bit aligned | Control interface   |
| <b>Program Source</b>    | <b>AXI4 Master Write</b>       | <b>Simplified</b> | 32-bit aligned | Control interface   |
| <b>Flag Sink</b>         | <b>AXI4 Master Read</b>        | <b>Simplified</b> | 32-bit aligned | Control interface   |
| <b>Flag Source</b>       | <b>AXI4 Master Read</b>        | <b>Simplified</b> | 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                                       |
|----------------------------|---------------------------------------------------|
| <b>Address Alignment</b>   | All addresses aligned to full data bus width      |
| <b>Transfer Size</b>       | All transfers use maximum size equal to bus width |
| <b>Burst Type</b>          | Incrementing bursts only ( $AxBURST = 2'b01$ )    |
| <b>Transfer Complexity</b> | 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                                       |
|---------------------------|---------------------------------------------------|
| <b>Address Alignment</b>  | 4-byte aligned addresses (minimum alignment)      |
| <b>Transfer Sizes</b>     | Multiple sizes supported: 4, 8, 16, 32, 64 bytes  |
| <b>Burst Type</b>         | Incrementing bursts only ( $AxBURST = 2'b01$ )    |
| <b>Alignment Strategy</b> | Progressive alignment to optimize bus utilization |

## Mode 1: Simplified Transfer Mode Specification

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

| Aspect                          | Requirement                                    |
|---------------------------------|------------------------------------------------|
| <b>Alignment Rule</b>           | All AXI transactions aligned to data bus width |
| <b>32-bit bus (4 bytes)</b>     | Address[1:0] must be 2'b00                     |
| <b>64-bit bus (8 bytes)</b>     | Address[2:0] must be 3'b000                    |
| <b>128-bit bus (16 bytes)</b>   | Address[3:0] must be 4'b0000                   |
| <b>256-bit bus (32 bytes)</b>   | Address[4:0] must be 5'b00000                  |
| <b>512-bit bus (64 bytes)</b>   | Address[5:0] must be 6'b000000                 |
| <b>1024-bit bus (128 bytes)</b> | Address[6:0] must be 7'b0000000                |

| Aspect           | Requirement                                                         |
|------------------|---------------------------------------------------------------------|
| <b>Rationale</b> | Maximizes bus efficiency and eliminates unaligned access complexity |

### Assumption 2: Fixed Transfer Size

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

## Mode 2: Flexible Transfer Mode Specification

### Assumption 1: 4-Byte Address Alignment

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

### Assumption 2: Multiple Transfer Sizes

| Transfer Size  | AxSIZE Value | Use Case                           |
|----------------|--------------|------------------------------------|
| <b>4 bytes</b> | 3'b010       | Initial alignment, small transfers |
| <b>8 bytes</b> | 3'b011       | Progressive alignment              |

| Transfer Size    | AxSIZE Value | Use Case                          |
|------------------|--------------|-----------------------------------|
| <b>16 bytes</b>  | 3'b100       | Progressive alignment             |
| <b>32 bytes</b>  | 3'b101       | Progressive alignment             |
| <b>64 bytes</b>  | 3'b110       | Optimal full-width transfers      |
| <b>128 bytes</b> | 3'b111       | Maximum efficiency (1024-bit bus) |

### Assumption 3: Progressive Alignment Strategy

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

### Assumption 4: Chunk Enable Support

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

## Common Protocol Assumptions (Both Modes)

### Assumption 1: Incrementing Bursts Only

| Aspect            | Requirement                                                    |
|-------------------|----------------------------------------------------------------|
| <b>Burst Type</b> | All AXI bursts use incrementing address mode (AxBURST = 2'b01) |

| Aspect                | Requirement                                                      |
|-----------------------|------------------------------------------------------------------|
| <b>Excluded Types</b> | No FIXED (2'b00) or WRAP (2'b10) bursts supported                |
| <b>Rationale</b>      | Simplifies address generation logic and covers most use cases    |
| <b>Benefit</b>        | Eliminates wrap boundary calculations and fixed address handling |

### Assumption 2: No Address Wraparound

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

## Flexible Mode: Address Calculation Examples

### Progressive Alignment Examples

#### Example 1: Address 0x1004 → 0x1040 (64-byte boundary)

| Step | Address | Size     | AxSIZE | Length | Bytes Transferred | Notes                 |
|------|---------|----------|--------|--------|-------------------|-----------------------|
| 1    | 0x1004  | 4 bytes  | 3'b010 | 1 beat | 4                 | Initial alignment     |
| 2    | 0x1008  | 8 bytes  | 3'b011 | 1 beat | 8                 | Progressive alignment |
| 3    | 0x1010  | 16 bytes | 3'b100 | 1 beat | 16                | Progressive alignment |
| 4    | 0x1020  | 32       | 3'b101 | 1 beat | 32                | Progressive alignment |

| Step | Address | Size            | AxSIZE | Length  | Bytes Transferred | Notes                    |
|------|---------|-----------------|--------|---------|-------------------|--------------------------|
|      |         | bytes           |        |         |                   | ssive alignment          |
| 5    | 0x1040  | <b>64 bytes</b> | 3'b110 | N beats | 64×N              | <b>Optimal transfers</b> |

### 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  | <b>64 bytes</b> | 3'b110 | N beats | 64×N              | <b>Optimal transfers</b> |

## Chunk Enable Pattern Examples

### 512-bit Bus with 16×32-bit chunks

| Transfer Size   | Address Offset | Chunk Pattern | Description  |
|-----------------|----------------|---------------|--------------|
| <b>4 bytes</b>  | 0x04           | 16'h0002      | Chunk 1 only |
| <b>8 bytes</b>  | 0x08           | 16'h000C      | Chunks 2-3   |
| <b>16 bytes</b> | 0x10           | 16'h00F0      | Chunks 4-7   |
| <b>32 bytes</b> | 0x20           | 16'hFF00      | Chunks 8-15  |
| <b>64 bytes</b> | 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 | <b>Bus-width aligned</b> | <b>4-byte aligned</b>       | Read address          |
| ar_len    | 8          | Master→Slave | 0-255                    | 0-255                       | Burst length - 1      |
| ar_size   | 3          | Master→Slave | <b>Fixed per bus</b>     | <b>Variable: 4-64 bytes</b> | Transfer size         |
| ar_burst  | 2          | Master→Slave | <b>2'b01 (INCR only)</b> | <b>2'b01 (INCR only)</b>    | 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_cach   | 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_val_id | 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 |

| Signal  | Width      | Direction    | Description             |
|---------|------------|--------------|-------------------------|
| 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 | <b>Bus-width aligned</b> | <b>4-byte aligned</b>       | Write address         |
| aw_len    | 8          | Master→Slave | 0-255                    | 0-255                       | Burst length - 1      |
| aw_size   | 3          | Master→Slave | <b>Fixed per bus</b>     | <b>Variable: 4-64 bytes</b> | Transfer size         |
| aw_burst  | 2          | Master→Slave | <b>2'b01 (INCR only)</b> | <b>2'b01 (INCR only)</b>    | 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                |

| Signal    | Width      | Direction    | Simplified Mode | Flexible Mode | Description   |
|-----------|------------|--------------|-----------------|---------------|---------------|
|           | h          |              | ave             |               | identifier    |
| aw_use_r  | USER_WIDTH | Master→Slave | Optional        | Optional      | User-defined  |
| aw_val_id | 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         |
|------------------------|---------------------------------------|---------------------|
| <b>First Address</b>   | Must be bus-width aligned             | Starting address    |
| <b>Address N</b>       | First_Address + (N × Bus_Width_Bytes) | Address for beat N  |
| <b>Alignment Check</b> | (Address % Bus_Width_Bytes) == 0      | Must always be true |

### Flexible Mode Address Generation

| Parameter                    | Formula                                 | Description           |
|------------------------------|-----------------------------------------|-----------------------|
| <b>First Address</b>         | Must be 4-byte aligned                  | Starting address      |
| <b>Address N</b>             | First_Address + (N × Transfer_Size)     | Address for beat N    |
| <b>Alignment Check</b>       | (Address % 4) == 0                      | Must always be true   |
| <b>Progressive Alignment</b> | Choose largest size ≤ bytes_to_boundary | Optimization strategy |

### 4KB Boundary Considerations (Both Modes)

| Validation Rule              | Formula                                                   | Description          |
|------------------------------|-----------------------------------------------------------|----------------------|
| <b>4KB Boundary</b>          | Bursts cannot cross 4KB (0x1000) boundaries               | AXI specification    |
| <b>Max Burst Calculation</b> | Max_Beats = (4KB - (Start_Address % 4KB)) / Transfer_Size | Burst limit          |
| <b>Boundary Check</b>        | 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'hFFFFFFF FFFF FFFF | 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'h0000000000<br>000000F | First 4 bytes  |
| 16 bytes      | 16'h000F      | 64'h0000000000<br>00000FF | First 16 bytes |
| 32 bytes      | 16'h0OFF      | 64'h0000000000<br>0FFFFF  | First 32 bytes |
| 64 bytes      | 16'hFFFF      | 64'hFFFFFFF FFFF<br>FFFFF | All 64 bytes   |

## Response Codes

### Response Code Specification

| Value | Name     |                  | Simplified Mode Usage    | Flexible Mode Usage   |
|-------|----------|------------------|--------------------------|-----------------------|
|       | e        | Description      |                          |                       |
| 2'b00 | OKA<br>Y | Normal<br>access | Bus-width aligned access | 4-byte aligned access |

| Value        | Name       | Description      | Simplified Mode Usage         | Flexible Mode Usage        |
|--------------|------------|------------------|-------------------------------|----------------------------|
|              |            | success          |                               |                            |
| <b>2'b01</b> | EXO<br>KAY | Exclusive access | Bus-width aligned exclusive   | 4-byte aligned exclusive   |
|              |            | success          |                               |                            |
| <b>2'b10</b> | SLVE<br>RR | Slave error      | Slave-specific error          | Slave-specific error       |
| <b>2'b11</b> | DECE<br>RR | Decode error     | <b>Bus-width misalignment</b> | <b>4-byte misalignment</b> |

## Implementation Benefits

### Simplified Mode Benefits

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

### Flexible Mode Benefits

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

## Mode Selection Guidelines

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

## Validation Requirements

### Simplified Mode Validation

| Validation Area          | Requirements                                   |
|--------------------------|------------------------------------------------|
| <b>Address Alignment</b> | Verify all addresses aligned to full bus width |
| <b>Fixed Size</b>        | Verify AxSIZE always matches DATA_WIDTH        |
| <b>Full Strobes</b>      | Verify w_strb is always all 1's                |
| <b>Burst Type</b>        | Verify AxBURST is always 2'b01                 |

### Flexible Mode Validation

| Validation Area          | Requirements                               |
|--------------------------|--------------------------------------------|
| <b>Address Alignment</b> | Verify all addresses are 4-byte aligned    |
| <b>Size Validation</b>   | Verify AxSIZE matches actual transfer size |

| Validation Area              | Requirements                                   |
|------------------------------|------------------------------------------------|
| <b>Chunk Consistency</b>     | Verify chunk enables match transfer size       |
| <b>Strobe Generation</b>     | Verify strobes generated correctly from chunks |
| <b>Progressive Alignment</b> | Verify alignment strategy optimization         |
| <b>Boundary Checking</b>     | Verify no 4KB boundary crossings               |

### Common Validation

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

## Performance Characteristics

### Simplified Mode Performance

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

### Flexible Mode Performance

| Metric         | Alignment Phase | Optimized Phase      | Description                 |
|----------------|-----------------|----------------------|-----------------------------|
| <b>Latency</b> | 3-15 cycles     | 3 cycles             | Variable based on alignment |
| <b>Through</b> | Variable        | 1 transfer per clock | Depends on                  |

| Metric            | Alignment Phase | Optimized Phase | Description              |
|-------------------|-----------------|-----------------|--------------------------|
| <b>put</b>        |                 |                 | alignment pattern        |
| <b>Efficiency</b> | 25-100%         | 100%            | Improves with alignment  |
| <b>Complexity</b> | 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-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  |

| Parameter  | Description                   | Valid Values | Default |
|------------|-------------------------------|--------------|---------|
| 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                                          |
|------------------------------|------------------------------------------------------|
| <b>Single Transfers Only</b> | No burst transactions supported                      |
| <b>No Transaction IDs</b>    | All transactions are in-order                        |
| <b>Fixed Transfer Size</b>   | Always uses full data bus width                      |
| <b>No User Signals</b>       | Simplified interface without user-defined extensions |

### Implementation Assumptions

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

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

#### *Assumption 2: Fixed Transfer Size*

| Aspect                    | Requirement                                       |
|---------------------------|---------------------------------------------------|
| <b>Transfer Size Rule</b> | All transfers use maximum size equal to bus width |
| <b>32-bit bus</b>         | AxSIZE = 3'b010 (4 bytes)                         |

| Aspect            | Requirement                                            |
|-------------------|--------------------------------------------------------|
| <b>64-bit bus</b> | AxSIZE = 3'b011 (8 bytes)                              |
| <b>Rationale</b>  | Maximizes bus utilization and simplifies control logic |
| <b>Benefit</b>    | No size decode logic required                          |

#### *Assumption 3: No Address Wraparound*

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

#### *Assumption 4: Standard Protection Attributes*

| Access Type              | AxPROT Value | Description                                             |
|--------------------------|--------------|---------------------------------------------------------|
| <b>Normal Access</b>     | 3'b000       | Data, secure, unprivileged                              |
| <b>Privileged Access</b> | 3'b001       | Data, secure, privileged                                |
| <b>Rationale</b>         |              | 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_WI DTH | Master→Slave | <b>8-byte aligned</b>   | Read address          |
| ar_prot  | 3           | Master→Slave | Implementation specific | Protection attributes |
| ar_valid | 1           | Master→Slave | 0 or 1                  | Address valid         |
| ar_ready | 1           | Slave→Master | 0 or 1                  | Address ready         |

### Read Data Channel (R)

| Signal | Width | Direction    | Description |
|--------|-------|--------------|-------------|
| r_data | 64    | Slave→Master | Read data   |

| Signal  | Width | Direction    | Description     |
|---------|-------|--------------|-----------------|
| 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                     |
|------------------|--------------------|--------------------------------------|
| <b>ar_id</b>     | Transaction ID     | Single transfers, no transaction IDs |
| <b>ar_len</b>    | Burst length       | Single transfers only                |
| <b>ar_size</b>   | Transfer size      | Fixed to bus width                   |
| <b>ar_burst</b>  | Burst type         | Single transfers only                |
| <b>ar_lock</b>   | Lock type          | Simplified access model              |
| <b>ar_cache</b>  | Cache attributes   | Simplified memory model              |
| <b>ar_qos</b>    | Quality of Service | Simplified priority model            |
| <b>ar_region</b> | Region identifier  | Simplified address space             |
| <b>ar_user</b>   | User-defined       | Simplified interface                 |
| <b>r_id</b>      | Transaction ID     | No transaction IDs needed            |
| <b>r_last</b>    | Last transfer      | Single transfers only                |
| <b>r_user</b>    | User-defined       | Simplified interface                 |

## Master Write Interface Specification

### Write Address Channel (AW)

| Signal  | Width       | Direction    | Required Values         | Description           |
|---------|-------------|--------------|-------------------------|-----------------------|
| aw_addr | ADDR_WI DTH | Master→Slave | <b>8-byte aligned</b>   | Write address         |
| aw_prot | 3           | Master→Slave | Implementation specific | Protection attributes |

| Signal   | Width | Direction    | Required Values | Description   |
|----------|-------|--------------|-----------------|---------------|
| 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<br>(byte enables) |
| w_valid | 1     | Master→Slave | Write data<br>valid             |
| w_ready | 1     | Slave→Master | Write data<br>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<br>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             |

| Removed Signal | AXI4 Usage     | AXI4-Lite Reason          |
|----------------|----------------|---------------------------|
| <b>aw_user</b> | User-defined   | Simplified interface      |
| <b>w_last</b>  | Last transfer  | Single transfers only     |
| <b>w_user</b>  | User-defined   | Simplified interface      |
| <b>b_id</b>    | Transaction ID | No transaction IDs needed |
| <b>b_user</b>  | User-defined   | Simplified interface      |

## Address Requirements

### Address Alignment Rules

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

### Address Validation Examples

| Address Category             | Examples                       | Status          |
|------------------------------|--------------------------------|-----------------|
| <b>Valid (8byte aligned)</b> | 0x1000, 0x1004, 0x1008, 0x100C | Accepted        |
| <b>Invalid (unaligned)</b>   | 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 | EXOK<br>AY | Exclusive access success | <b>Not used in AXI4-Lite</b>                  |
| 2'b10 | SLVER<br>R | Slave error              | Invalid register access                       |
| 2'b11 | DECER<br>R | Decode error             | <b>Address decode failure or misalignment</b> |

## Response Usage Guidelines

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

## Protection Signal Usage

### Protection Signal Encoding

| Bit | Name        | Description            | Recommended Usage              |
|-----|-------------|------------------------|--------------------------------|
| [0] | Privilege d | 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                |
|-------------------------------|--------------|----------------------------|
| <b>Normal Data Access</b>     | 3'b000       | Standard register access   |
| <b>Privileged Data Access</b> | 3'b001       | Privileged register access |
| <b>Debug Access</b>           | 3'b010       | Debug register access      |
| <b>Privileged Debug</b>       | 3'b011       | Privileged debug access    |

## Implementation Benefits

### Simplified Control Register Interface

| Benefit Area          | Simplification                | Impact         |
|-----------------------|-------------------------------|----------------|
| <b>Address Decode</b> | Simple 8-byte aligned address | Reduced decode |

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

### Address Decode Implementation

| Implementation Aspect         | Method                        | Benefit                |
|-------------------------------|-------------------------------|------------------------|
| <b>4-byte Alignment Check</b> | addr[1:0] == 2'b00            | Simple bit masking     |
| <b>Address Range Check</b>    | addr >= base && addr <= limit | Simple comparisons     |
| <b>Combined Check</b>         | alignment_ok && range_ok      | Single decode decision |

### Error Generation Logic

| Error Condition             | Check                    | Response        |
|-----------------------------|--------------------------|-----------------|
| <b>Address Misalignment</b> | addr[1:0] != 2'b00       | Generate DECERR |
| <b>Address Out of Range</b> | ! addr_in_range(addr)    | Generate DECERR |
| <b>Register Error</b>       | register_error_condition | Generate SLVERR |
| <b>Normal Access</b>        | All checks pass          | Generate OKAY   |

### Timing Requirements

#### Handshake Protocol

| Protocol Rule               | Requirement                               | Description            |
|-----------------------------|-------------------------------------------|------------------------|
| <b>Valid-Ready Transfer</b> | Transfer occurs when both VALID and READY | Standard AXI handshake |

| Protocol Rule             | Requirement                                         | Description           |
|---------------------------|-----------------------------------------------------|-----------------------|
| <b>Valid Independence</b> | VALID can be asserted independently of READY        | Master controls valid |
| <b>Ready Dependency</b>   | READY can depend on VALID state                     | Slave controls ready  |
| <b>Signal Stability</b>   | Once VALID asserted, all signals stable until READY | Data integrity        |

### Channel Dependencies

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

### Reset Behavior

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

### Validation Requirements

#### Functional Validation

| Validation Area          | Requirements                   |
|--------------------------|--------------------------------|
| <b>Address Alignment</b> | Verify all accesses are 8-byte |

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

### Timing Validation

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

### Error Injection Testing

| Test Type                 | Injection Method                                | Expected Response |
|---------------------------|-------------------------------------------------|-------------------|
| <b>Misaligned Address</b> | Inject addresses with $\text{addr}[2:0] \neq 0$ | DECERR response   |
| <b>Out of Range</b>       | Inject addresses outside valid range            | DECERR response   |
| <b>Register Errors</b>    | Inject register-specific errors                 | SLVERR response   |

### Example Transactions

#### 64-bit Register Write

| Parameter             | Value                   | Description            |
|-----------------------|-------------------------|------------------------|
| <b>Bus Width</b>      | 64 bits (8 bytes)       | Data bus configuration |
| <b>Target Address</b> | 0x1000 (8-byte aligned) | Valid aligned address  |

| Parameter                | Value                                                                                              | Description               |
|--------------------------|----------------------------------------------------------------------------------------------------|---------------------------|
| <b>Write Data</b>        | 0xDEADBEEFCA<br>FEBABE                                                                             | 64-bit data value         |
| <b>Required Settings</b> | aw_addr=0x100<br>0,<br>aw_prot=3'b000,<br>w_data=0xDEAD<br>BEEFCAFEBABE,<br>w_strb=8'b1111<br>1111 | 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,<br>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               |
|--------------------------|------------------------------------|---------------------------|
| <b>Bus Width</b>         | 64 bits (8 bytes)                  | Data bus configuration    |
| <b>Target Address</b>    | 0x1008 (8-byte aligned)            | Valid aligned address     |
| <b>Required Settings</b> | ar_addr=0x1008<br>, ar_prot=3'b000 | Transaction configuration |

### AR Transaction Flow

| Step | Action                       | Signal States                 |
|------|------------------------------|-------------------------------|
| 1    | Assert ar_valid with address | ar_valid=1,<br>ar_addr=0x1008 |
| 2    | Wait for address handshake   | ar_ready=1                    |

| Step | Action                 | Signal States             |
|------|------------------------|---------------------------|
| 3    | Wait for data response | r_valid=1,<br>r_resp=OKAY |
| 4    | Complete transaction   | r_ready=1                 |
| 5    | Capture data           | r_data (64 bits)          |

### Misaligned Address Example

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

### Common Use Cases

#### Typical Applications

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

#### Performance Considerations

| Consideration     | Impact                                 | Description      |
|-------------------|----------------------------------------|------------------|
| <b>Latency</b>    | Single-cycle responses preferred       | Simple registers |
| <b>Throughput</b> | Limited by single AXI4-Lite constraint |                  |

| Consideration     | Impact                                                                  | Description                |
|-------------------|-------------------------------------------------------------------------|----------------------------|
| <b>Efficiency</b> | outstanding transaction<br>64-bit transfers<br>maximize data efficiency | Modern system optimization |

## APB Programming Interface Specification for STREAM

### Overview

This document defines the APB programming interface for the STREAM DMA engine.

**Important:** STREAM uses a **simplified programming interface** for descriptor kick-off, NOT a full APB slave. The interface uses standard valid/ready handshaking per channel to initiate descriptor-based transfers.

This interface is separate from the optional APB configuration interface that may be added in future implementations for runtime configuration of STREAM parameters.

---

### Interface Summary

#### STREAM Programming Interface

STREAM provides a **simplified per-channel descriptor programming interface**:

| Signal Group       | Per-Channel Signals                              | Protocol              | Purpose                  |
|--------------------|--------------------------------------------------|-----------------------|--------------------------|
| <b>Programming</b> | apb_valid[ch],<br>apb_ready[ch],<br>apb_addr[ch] | Valid/Ready handshake | Descriptor address input |

**Note:** Despite the “apb\_” prefix, this is NOT a full AMBA APB interface. It’s a simplified programming interface using APB naming convention for historical reasons.

## Interface Parameters

| Parameter  | Description                    | Valid Values | Default |
|------------|--------------------------------|--------------|---------|
| DATA_WIDTH | APB data bus width in bits     | 8, 16, 32    | 32      |
| ADDR_WIDTH | APB address bus width in bits  | 16, 24, 32   | 32      |
| STRB_WIDTH | Write strobe width (APB4 only) | DATA_WIDTH/8 | 4       |
| NUM_SLAVES | Number of peripheral slaves    | 1-32         | 1       |
| DEPTH      | Internal buffer depth          | 2+           | 2       |

## Core Protocol Assumptions

### Inherent APB Simplifications

APB protocol inherently provides the following constraints:

1. **Single Master Only:** Only one bus master supported
2. **Non-Pipelined:** One transaction completes before next begins
3. **Simple 2-Phase Protocol:** Setup phase followed by access phase
4. **No Burst Transfers:** Only single transfers supported
5. **In-Order Completion:** No out-of-order transaction capability

### Implementation Assumptions

#### *Assumption 1: Word-Aligned Access Only*

| Aspect                  | Requirement                                              |
|-------------------------|----------------------------------------------------------|
| <b>Alignment Rule</b>   | All APB transfers are aligned to natural word boundaries |
| <b>8-bit transfers</b>  | Byte aligned (no restriction)                            |
| <b>16-bit transfers</b> | 2-byte aligned ( $\text{PADDR}[0] = 0$ )                 |
| <b>32-bit transfers</b> | 4-byte aligned ( $\text{PADDR}[1:0] = 2'b00$ )           |
| <b>Rationale</b>        | Simplifies peripheral decode logic and ensures           |

| Aspect | Requirement      |
|--------|------------------|
|        | efficient access |

### *Assumption 2: Standard Transfer Sizes*

| Bus Width         | Supported Transfer Sizes | Rationale                               |
|-------------------|--------------------------|-----------------------------------------|
| <b>32-bit bus</b> | 8-bit, 16-bit, 32-bit    | Covers typical register access patterns |
| <b>16-bit bus</b> | 8-bit, 16-bit            | Keeps decode logic simple               |
| <b>8-bit bus</b>  | 8-bit only               | Minimal complexity                      |

### *Assumption 3: Single-Cycle Default Operation*

| Aspect                  | Requirement                                                   |
|-------------------------|---------------------------------------------------------------|
| <b>Default Behavior</b> | Most peripherals respond in a single cycle (PREADY tied high) |
| <b>Optimization</b>     | PREADY optimization for simple registers                      |
| <b>Exception</b>        | Complex peripherals may use PREADY for wait states            |
| <b>Rationale</b>        | Minimizes latency for control register access                 |

## Interface Signal Specification

### Clock and Reset Signals

| Signal          | IO | Description             |
|-----------------|----|-------------------------|
| src_clk         | I  | Source APB clock signal |
| snk_rdata[31:0] | I  | Read data               |
| src_wdata[31:0] | O  | Write data              |

### Address and Control Signals

| Signal   | Width   | Direction    | Required Values                  | Description                    |
|----------|---------|--------------|----------------------------------|--------------------------------|
| src_addr | 11 bits | Master→Slave | <b>Aligned per transfer size</b> | Peripheral address (2KB space) |
| src_sel  | 1       | Master→Slave | 0 or 1                           | Peripheral select              |

| Signal     | Width   | Direction    | Required Values                  | Description                          |
|------------|---------|--------------|----------------------------------|--------------------------------------|
| src_enable | 1       | Master→Slave | 0 or 1                           | Enable signal                        |
| src_write  | 1       | Master→Slave | 0 or 1                           | Write enable<br>(1=write,<br>0=read) |
| snk_addr   | 11 bits | Master→Slave | <b>Aligned per transfer size</b> | Peripheral address (2KB space)       |
| snk_sel    | 1       | Master→Slave | 0 or 1                           | Peripheral select                    |
| snk_enable | 1       | Master→Slave | 0 or 1                           | Enable signal                        |
| snk_write  | 1       | Master→Slave | 0 or 1                           | Write enable<br>(1=write,<br>0=read) |

## Data Signals

| Signal    | Width | Direction    | Description |
|-----------|-------|--------------|-------------|
| src_wdata | 32    | Master→Slave | Write data  |
| src_rdata | 32    | Slave→Master | Read data   |
| snk_wdata | 32    | Master→Slave | Write data  |
| snk_rdata | 32    | Slave→Master | Read data   |

## Response Signals

| Signal     | Width | Direction    | Description    |
|------------|-------|--------------|----------------|
| src_ready  | 1     | Slave→Master | Transfer ready |
| src_slverr | 1     | Slave→Master | Slave error    |
| snk_ready  | 1     | Slave→Master | Transfer ready |
| snk_slverr | 1     | Slave→Master | Slave error    |

## Address Space Configuration

### Address Range Specification

| Parameter            | Value                       | Description             |
|----------------------|-----------------------------|-------------------------|
| <b>Address Width</b> | 11 bits<br>(src_addr[10:0]) | Full address space      |
| <b>Address Space</b> | 2KB (2048 bytes)            | Total addressable space |

| Parameter                 | Value                                    | Description         |
|---------------------------|------------------------------------------|---------------------|
| <b>Register Alignment</b> | 32-bit aligned<br>(4-byte<br>boundaries) | Address constraints |
| <b>Usable Addresses</b>   | 0x000 to 0x7FF                           | Valid address range |

### Address Decode Implementation

| Register    | Address Bits                | Address Value | Description                |
|-------------|-----------------------------|---------------|----------------------------|
| <b>reg0</b> | src_addr[10:2]<br>== 9'h000 | 0x000         | First register             |
| <b>reg1</b> | src_addr[10:2]<br>== 9'h001 | 0x004         | Second register            |
| <b>reg2</b> | src_addr[10:2]<br>== 9'h002 | 0x008         | Third register             |
| ...         | ...                         | ...           | Up to 512 32-bit registers |

### Transaction Protocol

#### 2-Phase Transaction States

| Phase               | State  | Duration        | Signal Requirements                             |
|---------------------|--------|-----------------|-------------------------------------------------|
| <b>Setup (T1)</b>   | SETUP  | 1 clock cycle   | src_sel=1, src_enable=0,<br>address/data stable |
| <b>Access (T2+)</b> | ACCESS | 1+ clock cycles | src_sel=1, src_enable=1, wait for<br>src_ready  |
| <b>Idle</b>         | IDLE   | Variable        | src_sel=0, src_enable=0                         |

#### State Transitions

| Current State | Next State | Condition         | Description                       |
|---------------|------------|-------------------|-----------------------------------|
| <b>IDLE</b>   | SETUP      | Transaction start | Master drives address and control |
| <b>SETUP</b>  | ACCESS     | Clock edge        | Master asserts src_enable         |
| <b>ACCESS</b> | IDLE       | src_ready=1       | Transaction completes             |
| <b>ACCESS</b> | ACCESS     | src_ready=0       | Wait state                        |

| Current State | Next State | Condition | Description |
|---------------|------------|-----------|-------------|
|               |            |           | continues   |

## Transaction Timing Requirements

| Timing Parameter    | Requirement                                     | Description      |
|---------------------|-------------------------------------------------|------------------|
| <b>Setup Time</b>   | All master signals stable before rising src_clk | Signal stability |
| <b>Hold Time</b>    | All master signals held after rising src_clk    | Signal stability |
| <b>Output Delay</b> | Slave outputs valid after rising src_clk        | Response timing  |

## Advanced Features

### Dynamic Clock Gating

| Feature                            | Description                                          | Benefit                    |
|------------------------------------|------------------------------------------------------|----------------------------|
| <b>Automatic Power Reduction</b>   | Clocks gated when no activity detected               | Significant power savings  |
| <b>Configurable Idle Threshold</b> | Programmable idle count before gating                | Tunable power/performance  |
| <b>Multi-Domain Support</b>        | Independent gating for APB (PCLK) and backend (ACLK) | Flexible power management  |
| <b>Graceful Handoff</b>            | Ready signals forced to zero during gating           | Protocol compliance        |
| <b>Activity Detection</b>          | Monitors valid signals across interfaces             | Intelligent gating control |

### Clock Domain Crossing (CDC) Handshaking

| Feature                            | Description                                    | Benefit            |
|------------------------------------|------------------------------------------------|--------------------|
| <b>Arbitrary Frequency Support</b> | Works across any clock frequency ratios        | Design flexibility |
| <b>Robust Reliability</b>          | Proven handshaking with proper synchronization | Zero data loss     |
| <b>Command/Response Separation</b> | Independent CDC paths                          | Maximum throughput |

| Feature                        | Description                              | Benefit               |
|--------------------------------|------------------------------------------|-----------------------|
| <b>Skid Buffer Integration</b> | Internal buffering prevents backpressure | Smooth operation      |
| <b>Deterministic Latency</b>   | Predictable timing characteristics       | System predictability |

## Buffering and Flow Control

| Feature                      | Description                                   | Benefit              |
|------------------------------|-----------------------------------------------|----------------------|
| <b>Internal Skid Buffers</b> | Configurable depth buffering                  | Improved performance |
| <b>Backpressure Handling</b> | Proper ready/valid handshaking                | Flow control         |
| <b>Command Pipelining</b>    | Backend processes while APB handles responses | Efficiency           |
| <b>Response Queuing</b>      | Responses buffered for varying latencies      | Latency tolerance    |

## Error Handling

### Error Response Specification

| Condition                   | src_slverr | Description                |
|-----------------------------|------------|----------------------------|
| <b>Successful access</b>    | 0          | Normal completion          |
| <b>Address decode error</b> | 1          | Invalid address            |
| <b>Protection violation</b> | 1          | Insufficient privilege     |
| <b>Backend error</b>        | 1          | Downstream error condition |

### Error Response Timing

| Timing Requirement           | Description                                   |
|------------------------------|-----------------------------------------------|
| <b>src_slverr Validity</b>   | Must be valid when src_ready is asserted      |
| <b>Error Completion</b>      | Error response completes transaction normally |
| <b>Master Responsibility</b> | Master must check src_slverr status           |

## Reset Behavior

### Reset Requirements

| Reset Phase              | Signal Requirements                    | Description         |
|--------------------------|----------------------------------------|---------------------|
| <b>Reset Assertion</b>   | All outputs to known states            | Deterministic reset |
| <b>Reset Deassertion</b> | src_sel=0, src_enable=0 in first cycle | Clean startup       |
| <b>Reset Values</b>      | src_ready=0, src_slverr=0, src_rdata=0 | Default states      |

## Implementation Variants

### Available Implementations

| Variant                          | Features                                | Use Case                |
|----------------------------------|-----------------------------------------|-------------------------|
| <b>Basic APB Slave</b>           | Single clock domain, internal buffering | Simple peripherals      |
| <b>CDC-Enabled APB Slave</b>     | Dual clock domain (PCLK/ACLK)           | Mixed-frequency systems |
| <b>Power-Optimized APB Slave</b> | Dynamic clock gating                    | Power-sensitive designs |

## Performance Characteristics

### Latency Specifications

| Metric                        | Value            | Description                    |
|-------------------------------|------------------|--------------------------------|
| <b>Minimum Read Latency</b>   | 2 clock cycles   | Setup + access phases          |
| <b>Minimum Write Latency</b>  | 2 clock cycles   | Setup + access phases          |
| <b>CDC Additional Latency</b> | 2-6 clock cycles | Depends on clock relationships |
| <b>Buffer Latency</b>         | Minimal          | Skid buffer design             |

### Throughput Specifications

| Metric                    | Value            | Description            |
|---------------------------|------------------|------------------------|
| <b>Single Transaction</b> | 2 clocks minimum | Basic transaction time |
| <b>Back-to-back</b>       | Limited by       | Depends on peripheral  |

| Metric                | Value                     | Description      |
|-----------------------|---------------------------|------------------|
| <b>Transactions</b>   | src_ready                 |                  |
| <b>CDC Throughput</b> | Maintained across domains | Proper buffering |

### Power Consumption

| Power Category         | Characteristics                         | Description              |
|------------------------|-----------------------------------------|--------------------------|
| <b>Active Power</b>    | Standard CMOS logic                     | Normal operation         |
| <b>Idle Power</b>      | Significantly reduced with clock gating | Power optimization       |
| <b>Gating Overhead</b> | Minimal additional logic                | Efficient implementation |

### Validation Requirements

#### Functional Validation

| Validation Area            | Requirements                                 |
|----------------------------|----------------------------------------------|
| <b>Protocol Compliance</b> | Verify 2-phase setup/access sequence         |
| <b>Address Decode</b>      | Verify 11-bit address handling and alignment |
| <b>Error Response</b>      | Verify src_slver generation and handling     |
| <b>Wait States</b>         | Verify src_ready functionality               |
| <b>CDC Operation</b>       | Verify cross-clock domain transfers          |
| <b>Clock Gating</b>        | Verify power management behavior             |
| <b>Buffer Operation</b>    | Verify skid buffer and flow control          |

#### Timing Validation

| Validation Area       | Requirements                      |
|-----------------------|-----------------------------------|
| <b>Setup/Hold</b>     | Verify signal timing requirements |
| <b>Reset Sequence</b> | Verify proper reset behavior      |

| Validation Area           | Requirements                              |
|---------------------------|-------------------------------------------|
| <b>Multi-Clock</b>        | Verify CDC timing across frequency ranges |
| <b>Gating Transitions</b> | Verify clock enable/disable timing        |

## Example Transactions

### 32-bit Register Write

| Clock Cycle | src_sel | src_enabl e | src_writ e | src_add r | src_wda ta | src_read y | Phas e |
|-------------|---------|-------------|------------|-----------|------------|------------|--------|
| 1           | 1       | 0           | 1          | 0x100     | 0xABCD     | 0          | Setup  |
| 2           | 1       | 1           | 1          | 0x100     | 0xABCD     | 1          | Access |
| 3           | 0       | 0           | -          | -         | -          | 0          | Idle   |

### 32-bit Register Read with Wait State

| Clock Cycle | src_sel | src_enabl e | src_writ e | src_add r | src_rdat a | src_read y | Phas e   |
|-------------|---------|-------------|------------|-----------|------------|------------|----------|
| 1           | 1       | 0           | 0          | 0x104     | -          | 0          | Setup    |
| 2           | 1       | 1           | 0          | 0x104     | -          | 0          | Access   |
| 3           | 1       | 1           | 0          | 0x104     | 0x1234     | 1          | Complete |
| 4           | 0       | 0           | -          | -         | -          | 0          | Idle     |

## Monitor Bus Architecture and Event Code Organization

### STREAM-Specific Context

**For STREAM DMA Engine:** This document describes the generic Monitor Bus (MonBus) protocol used across all AMBA-based components. STREAM uses MonBus for unified event reporting from:

- Descriptor engines (8 sources, agent IDs 16-23)
- Schedulers (8 sources, agent IDs 48-55)
- Descriptor AXI monitor (agent ID 8)
- Read/Write AXI monitors (configurable agent IDs)

All STREAM events use Unit ID = 1 and follow the standard 64-bit packet format defined below.

---

## 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, MNOC (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, MNOC, 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                     | 8            | 8       |

| Parameter        | Description      | Valid Values | Default |
|------------------|------------------|--------------|---------|
|                  | identifier width |              |         |
| EVENT_DATA_WIDTH | Event data width | 35           | 35      |

## Core Design Assumptions

### Assumption 1: Hierarchical Event Organization

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

### Assumption 2: Protocol Isolation

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

### Assumption 3: Two-Tier Memory Architecture

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

### Assumption 4: Configurable Packet Routing

| Aspect              | Requirement                                   |
|---------------------|-----------------------------------------------|
| <b>Routing Rule</b> | Different packet types can route to different |

| Aspect                  | Requirement                                            |
|-------------------------|--------------------------------------------------------|
|                         | destinations                                           |
| <b>Configuration</b>    | Base/limit registers define routing per packet type    |
| <b>Priority Support</b> | Configurable priority levels per packet type           |
| <b>Rationale</b>        | 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_val_id | 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_val_id | 1     | APB Monitor→Bus | APB event valid     |
| apb_event_ready  | 1     | Bus→APB Monitor | Ready for APB event |
| mnoc_event       | 64    | MNOC            | MNOC event          |

| Signal            | Width | Direction   | Description          |
|-------------------|-------|-------------|----------------------|
|                   |       | Monitor→Bus | packet               |
| mnoc_event_va_lid | 1     | MNOC        | MNOC event           |
|                   |       | Monitor→Bus | valid                |
| mnoc_event_ready  | 1     | Bus→MNOC    | Ready for MNOC event |
|                   |       | Monitor     |                      |
| arb_event         | 64    | ARB         | ARB event            |
|                   |       | Monitor→Bus | packet               |
| arb_event_val_id  | 1     | ARB         | ARB event            |
|                   |       | Monitor→Bus | valid                |
| arb_event_ready   | 1     | Bus→ARB     | Ready for ARB event  |
|                   |       | Monitor     |                      |
| core_event        | 64    | CORE        | CORE event           |
|                   |       | Monitor→Bus | packet               |
| core_event_va_lid | 1     | CORE        | CORE event           |
|                   |       | Monitor→Bus | valid                |
| core_event_ready  | 1     | Bus→CORE    | Ready for CORE event |
|                   |       | Monitor     |                      |

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

### Packet Type Definitions

| Value      | Name        | Purpose                              | Applicable Protocols |
|------------|-------------|--------------------------------------|----------------------|
| <b>0x0</b> | Error       | Protocol violations, response errors | All                  |
| <b>0x1</b> | Completion  | Successful transaction completion    | All                  |
| <b>0x2</b> | Threshold   | Threshold crossed events             | All                  |
| <b>0x3</b> | Timeout     | Timeout conditions                   | All                  |
| <b>0x4</b> | Performance | Performance metrics                  | All                  |
| <b>0x5</b> | Credit      | Credit management                    | MNOC only            |
| <b>0x6</b> | Channel     | Channel status                       | MNOC only            |
| <b>0x7</b> | Stream      | Stream events                        | MNOC only            |

| Value          | Name          | Purpose                | Applicable Protocols |
|----------------|---------------|------------------------|----------------------|
| <b>0x8</b>     | Address Match | Address matching       | AXI only             |
| <b>0x9</b>     | APB Specific  | APB protocol events    | APB only             |
| <b>0xA-0xE</b> | Reserved      | Future expansion       | -                    |
| <b>0xF</b>     | Debug         | Debug and trace events | All                  |

## Protocol-Specific Event Codes

### AXI Protocol Events

*Error Events (PktTypeError + PROTOCOL\_AXI)*

| Code       | Event Name                    | Description                  |
|------------|-------------------------------|------------------------------|
| <b>0x0</b> | AXI_ERR_RESP_SLVE<br>RR       | Slave error response         |
| <b>0x1</b> | AXI_ERR_RESP_DECE<br>RR       | Decode error response        |
| <b>0x2</b> | AXI_ERR_DATA_ORP<br>HAN       | Data without command         |
| <b>0x3</b> | AXI_ERR_RESP_ORP<br>HAN       | Response without transaction |
| <b>0x4</b> | AXI_ERR_PROTOCOL              | Protocol violation           |
| <b>0x5</b> | AXI_ERR_BURST_LE<br>NGTH      | Invalid burst length         |
| <b>0x6</b> | AXI_ERR_BURST_SIZ<br>E        | Invalid burst size           |
| <b>0x7</b> | AXI_ERR_BURST_TYP<br>E        | Invalid burst type           |
| <b>0x8</b> | AXI_ERR_ID_COLLISI<br>ON      | ID collision detected        |
| <b>0x9</b> | AXI_ERR_WRITE_BE<br>FORE_ADDR | Write data before address    |

| Code       | Event Name               | Description                   |
|------------|--------------------------|-------------------------------|
| <b>0xA</b> | AXI_ERR_RESP_BEFORE_DATA | Response before data complete |
| <b>0xB</b> | AXI_ERR_LAST_MISS        | Missing LAST signal           |
| <b>0xC</b> | AXI_ERR_STROBE_ERROR     | Write strobe error            |
| <b>0xD</b> | AXI_ERR_RESERVED_D       | Reserved                      |
| <b>0xE</b> | AXI_ERR_RESERVED_E       | Reserved                      |
| <b>0xF</b> | AXI_ERR_USER_DEFINED     | User-defined error            |

*Timeout Events (PktTypeTimeout + PROTOCOL\_AXI)*

| Code           | Event Name                | Description              |
|----------------|---------------------------|--------------------------|
| <b>0x0</b>     | AXI_TIMEOUT_CMD           | Command/Address timeout  |
| <b>0x1</b>     | AXI_TIMEOUT_DATA          | Data timeout             |
| <b>0x2</b>     | AXI_TIMEOUT_RESP          | Response timeout         |
| <b>0x3</b>     | AXI_TIMEOUT_HANDSHAKE     | Handshake timeout        |
| <b>0x4</b>     | AXI_TIMEOUT_BURST         | Burst completion timeout |
| <b>0x5</b>     | AXI_TIMEOUT_EXCLUSIVE_USE | Exclusive access timeout |
| <b>0x6-0xE</b> | Reserved                  | Future expansion         |
| <b>0xF</b>     | AXI_TIMEOUT_USER_DEFINED  | User-defined timeout     |

*Performance Events (PktTypePerf + PROTOCOL\_AXI)*

| Code       | Event Name            | Description           |
|------------|-----------------------|-----------------------|
| <b>0x0</b> | AXI_PERF_ADDR_LATENCY | Address phase latency |
| <b>0x1</b> | AXI_PERF_DATA_LATENCY | Data phase latency    |

| Code           | Event Name                     | Description               |
|----------------|--------------------------------|---------------------------|
| <b>0x2</b>     | AXI_PERF_RESP_LATENCY          | Response phase latency    |
| <b>0x3</b>     | AXI_PERF_TOTAL_LATENCY         | Total transaction latency |
| <b>0x4</b>     | AXI_PERF_THROUGHPUT            | Transaction throughput    |
| <b>0x5</b>     | AXI_PERF_ERROR RATE            | Error rate                |
| <b>0x6</b>     | AXI_PERF_ACTIVE_COUNT          | Active transaction count  |
| <b>0x7</b>     | AXI_PERF_BANDWIDTH_UTILIZATION | Bandwidth utilization     |
| <b>0x8</b>     | AXI_PERF_QUEUE_DEPTH           | Average queue depth       |
| <b>0x9</b>     | AXI_PERF_BURST_EFFICIENCY      | Burst efficiency metric   |
| <b>0xA-0xE</b> | Reserved                       | Future expansion          |
| <b>0xF</b>     | AXI_PERF_USER_DEFINED          | User-defined performance  |

## APB Protocol Events

*Error Events (PktTypeError + PROTOCOL\_APB)*

| Code       | Event Name               | Description                     |
|------------|--------------------------|---------------------------------|
| <b>0x0</b> | APB_ERR_PSLVERR          | Peripheral slave error          |
| <b>0x1</b> | APB_ERR_SETUP_VIOLATION  | Setup phase protocol violation  |
| <b>0x2</b> | APB_ERR_ACCESS_VIOLATION | Access phase protocol violation |
| <b>0x3</b> | APB_ERR_STROBE_ERROR     | Write strobe error              |
| <b>0x4</b> | APB_ERR_ADDR_DECODE      | Address decode error            |
| <b>0x5</b> | APB_ERR_PROT_VIO         | Protection violation            |

| Code           | Event Name               | Description           |
|----------------|--------------------------|-----------------------|
|                | LATION                   | (PPROT)               |
| <b>0x6</b>     | APB_ERR_ENABLE_E<br>RROR | Enable phase error    |
| <b>0x7</b>     | APB_ERR_READY_ER<br>ROR  | PREADY protocol error |
| <b>0x8-0xE</b> | Reserved                 | Future expansion      |
| <b>0xF</b>     | APB_ERR_USER_DEF<br>INED | User-defined error    |

*Timeout Events (PktTypeTimeout + PROTOCOL\_APB)*

| Code           | Event Name               | Description                         |
|----------------|--------------------------|-------------------------------------|
| <b>0x0</b>     | APB_TIMEOUT_SETUP        | Setup phase timeout                 |
| <b>0x1</b>     | APB_TIMEOUT_ACCESS       | Access phase timeout                |
| <b>0x2</b>     | APB_TIMEOUT_ENABLE       | Enable phase timeout (PREADY stuck) |
| <b>0x3</b>     | APB_TIMEOUT_PREADYSTUCK  | PREADY stuck low                    |
| <b>0x4</b>     | APB_TIMEOUT_TRANSFER     | Overall transfer timeout            |
| <b>0x5-0xE</b> | Reserved                 | Future expansion                    |
| <b>0xF</b>     | APB_TIMEOUT_USER_DEFINED | User-defined timeout                |

*Completion Events (PktTypeCompletion + PROTOCOL\_APB)*

| Code           | Event Name                     | Description                |
|----------------|--------------------------------|----------------------------|
| <b>0x0</b>     | APB_COMPL_TRANSACTION_COMPLETE | Transaction completed      |
| <b>0x1</b>     | APB_COMPL_READ_COMPLETE        | Read transaction complete  |
| <b>0x2</b>     | APB_COMPL_WRITE_COMPLETE       | Write transaction complete |
| <b>0x3-0xE</b> | Reserved                       | Future expansion           |

| Code       | Event Name             | Description             |
|------------|------------------------|-------------------------|
| <b>0xF</b> | APB_COMPL_USER_DEFINED | User-defined completion |

*Threshold Events (PktTypeThreshold + PROTOCOL\_APB)*

| Code           | Event Name              | Description              |
|----------------|-------------------------|--------------------------|
| <b>0x0</b>     | APB_THRESH_LATENCY      | APB latency threshold    |
| <b>0x1</b>     | APB_THRESH_ERRO_R_RATE  | APB error rate threshold |
| <b>0x2</b>     | APB_THRESH_ACCESS_COUNT | Access count threshold   |
| <b>0x3</b>     | APB_THRESH_BANDWIDTH    | Bandwidth threshold      |
| <b>0x4-0xE</b> | Reserved                | Future expansion         |
| <b>0xF</b>     | APB_THRESH_USER_DEFINED | User-defined threshold   |

*Performance Events (PktTypePerf + PROTOCOL\_APB)*

| Code           | Event Name                           | Description                 |
|----------------|--------------------------------------|-----------------------------|
| <b>0x0</b>     | APB_PERF_READ_LATENCY                | Read transaction latency    |
| <b>0x1</b>     | APB_PERF_WRITE_LATENCY               | Write transaction latency   |
| <b>0x2</b>     | APB_PERF_THROUGHPUT                  | Transaction throughput      |
| <b>0x3</b>     | APB_PERF_ERROR_RATE                  | Error rate                  |
| <b>0x4</b>     | APB_PERF_ACTIVE_COUNT                | Active transaction count    |
| <b>0x5</b>     | APB_PERF_COMPLETED_TRANSACTION_COUNT | Completed transaction count |
| <b>0x6-0xE</b> | Reserved                             | Future expansion            |
| <b>0xF</b>     | APB_PERF_USER_DEFINED                | User-defined performance    |

*Debug Events (PktTypeDebug + PROTOCOL\_APB)*

| Code           | Event Name              | Description        |
|----------------|-------------------------|--------------------|
| <b>0x0</b>     | APB_DEBUG_STATE_CHANGE  | APB state changed  |
| <b>0x1</b>     | APB_DEBUG_SETUP_PHASE   | Setup phase event  |
| <b>0x2</b>     | APB_DEBUG_ACCESS_PHASE  | Access phase event |
| <b>0x3</b>     | APB_DEBUG_ENABLE_PHASE  | Enable phase event |
| <b>0x4</b>     | APB_DEBUG_PSEL_T_RACE   | PSEL trace         |
| <b>0x5</b>     | APB_DEBUG_PENABLE_TRACE | PENABLE trace      |
| <b>0x6</b>     | APB_DEBUG_PREADY_TRACE  | PREADY trace       |
| <b>0x7</b>     | APB_DEBUG_PPROT_TRACE   | PPROT trace        |
| <b>0x8</b>     | APB_DEBUG_PSTRB_TRACE   | PSTRB trace        |
| <b>0x9-0xE</b> | Reserved                | Future expansion   |
| <b>0xF</b>     | APB_DEBUG_USER_DEFINED  | User-defined debug |

**MNOC Protocol Events**

*Error Events (PktTypeError + PROTOCOL\_MNOC)*

| Code       | Event Name                  | Description             |
|------------|-----------------------------|-------------------------|
| <b>0x0</b> | MNOC_ERR_PARITY             | Parity error            |
| <b>0x1</b> | MNOC_ERR_PROTOCOL_VIOLATION | Protocol violation      |
| <b>0x2</b> | MNOC_ERR_OVERFLOW           | Buffer/Credit overflow  |
| <b>0x3</b> | MNOC_ERR_UNDERFLOW          | Buffer/Credit underflow |
| <b>0x4</b> | MNOC_ERR_ORPHANED           | Orphaned                |

| Code           | Event Name                | Description              |
|----------------|---------------------------|--------------------------|
| <b>0x5</b>     | N                         | packet/ACK               |
|                | MNOC_ERR_INVALI           | Invalid                  |
|                | D                         | type/channel/payloa<br>d |
| <b>0x6</b>     | MNOC_ERR_HEADER<br>_CRC   | Header CRC error         |
| <b>0x7</b>     | MNOC_ERR_PAYLOA<br>D_CRC  | Payload CRC error        |
| <b>0x8</b>     | MNOC_ERR_SEQUEN<br>CE     | Sequence number<br>error |
| <b>0x9</b>     | MNOC_ERR_ROUTE            | Routing error            |
| <b>0xA</b>     | MNOC_ERR_DEADLO<br>CK     | Deadlock detected        |
| <b>0xB-0xE</b> | Reserved                  | Future expansion         |
| <b>0xF</b>     | MNOC_ERR_USER_D<br>EFINED | User-defined error       |

*Credit Events (PktTypeCredit + PROTOCOL\_MNOC)*

| Code       | Event Name                | Description                  |
|------------|---------------------------|------------------------------|
| <b>0x0</b> | MNOC_CREDIT_ALL<br>OCATED | Credits allocated            |
| <b>0x1</b> | MNOC_CREDIT_CON<br>SUMED  | Credits consumed             |
| <b>0x2</b> | MNOC_CREDIT_RET<br>URNED  | Credits returned             |
| <b>0x3</b> | MNOC_CREDIT_OVE<br>RFLOW  | Credit overflow<br>detected  |
| <b>0x4</b> | MNOC_CREDIT_UND<br>ERFLOW | Credit underflow<br>detected |
| <b>0x5</b> | MNOC_CREDIT_EXH<br>AUSTED | All credits exhausted        |
| <b>0x6</b> | MNOC_CREDIT_REST<br>ORED  | Credits restored             |
| <b>0x7</b> | MNOC_CREDIT_EFFI          | Credit efficiency            |

| Code           | Event Name              | Description               |
|----------------|-------------------------|---------------------------|
|                | CIENCY                  | metric                    |
| <b>0x8</b>     | MNOC_CREDIT_LEAK        | Credit leak detected      |
| <b>0x9-0xE</b> | Reserved                | Future expansion          |
| <b>0xF</b>     | MNOC_CREDIT_USE_DEFINED | User-defined credit event |

*Channel Events (PktTypeChannel + PROTOCOL\_MNOC)*

| Code           | Event Name                       | Description                 |
|----------------|----------------------------------|-----------------------------|
| <b>0x0</b>     | MNOC_CHANNEL_OPENED              | Channel opened              |
| <b>0x1</b>     | MNOC_CHANNEL_CLOSED              | Channel closed              |
| <b>0x2</b>     | MNOC_CHANNEL_STALLED             | Channel stalled             |
| <b>0x3</b>     | MNOC_CHANNEL_RESUMED             | Channel resumed             |
| <b>0x4</b>     | MNOC_CHANNEL_CONGESTION_DETECTED | Channel congestion detected |
| <b>0x5</b>     | MNOC_CHANNEL_PRIORITY_CHANGE     | Channel priority change     |
| <b>0x6-0xE</b> | Reserved                         | Future expansion            |
| <b>0xF</b>     | MNOC_CHANNEL_USER_DEFINED        | User-defined channel event  |

*Stream Events (PktTypeStream + PROTOCOL\_MNOC)*

| Code       | Event Name          | Description        |
|------------|---------------------|--------------------|
| <b>0x0</b> | MNOC_STREAM_STARTED | Stream started     |
| <b>0x1</b> | MNOC_STREAM_ENDED   | Stream ended (EOS) |
| <b>0x2</b> | MNOC_STREAM_PAUSED  | Stream paused      |
| <b>0x3</b> | MNOC_STREAM_RESUMED | Stream resumed     |

| Code           | Event Name                 | Description               |
|----------------|----------------------------|---------------------------|
| <b>0x4</b>     | MNOC_STREAM_OVF<br>RFLOW   | Stream buffer overflow    |
| <b>0x5</b>     | MNOC_STREAM_UND<br>DERFLOW | Stream buffer underflow   |
| <b>0x6-0xE</b> | Reserved                   | Future expansion          |
| <b>0xF</b>     | MNOC_STREAM_USE_R_DEFINED  | User-defined stream event |

## ARB Protocol Events

*Error Events (PktTypeError + PROTOCOL\_ARB)*

| Code       | Event Name                 | Description                  |
|------------|----------------------------|------------------------------|
| <b>0x0</b> | ARB_ERR_STARVATION         | Client request starvation    |
| <b>0x1</b> | ARB_ERR_ACK_TIMEOUT        | Grant ACK timeout            |
| <b>0x2</b> | ARB_ERR_PROTOCOL_VIOLATION | ACK protocol violation       |
| <b>0x3</b> | ARB_ERR_CREDIT_VIOLATION   | Credit system violation      |
| <b>0x4</b> | ARB_ERR_FAIRNESS_VIOLATION | Weighted fairness violation  |
| <b>0x5</b> | ARB_ERR_WEIGHT_UNDERFLOW   | Weight credit underflow      |
| <b>0x6</b> | ARB_ERR_CONCURRENT_GRANTS  | Multiple simultaneous grants |
| <b>0x7</b> | ARB_ERR_INVALID_GRANT_ID   | Invalid grant ID detected    |
| <b>0x8</b> | ARB_ERR_ORPHAN_ACK         | ACK without pending grant    |
| <b>0x9</b> | ARB_ERR_GRANT_OVERLAP      | Overlapping grant periods    |
| <b>0xA</b> | ARB_ERR_MASK_ERROR         | Round-robin mask error       |
| <b>0xB</b> | ARB_ERR_STATE_ERROR        | FSM state error              |

| Code           | Event Name            | Description           |
|----------------|-----------------------|-----------------------|
|                | CHINE                 |                       |
| <b>0xC</b>     | ARB_ERR_CONFIGURATION | Invalid configuration |
| <b>0xD-0xE</b> | Reserved              | Future expansion      |
| <b>0xF</b>     | ARB_ERR_USER_DEFINED  | User-defined error    |

*Timeout Events (PktTypeTimeout + PROTOCOL\_ARB)*

| Code           | Event Name                | Description           |
|----------------|---------------------------|-----------------------|
| <b>0x0</b>     | ARB_TIMEOUT_GRANT_ACK     | Grant ACK timeout     |
| <b>0x1</b>     | ARB_TIMEOUT_REQUEST_HOLD  | Request held too long |
| <b>0x2</b>     | ARB_TIMEOUT_WEIGHT_UPDATE | Weight update timeout |
| <b>0x3</b>     | ARB_TIMEOUT_BLOCK_RELEASE | Block release timeout |
| <b>0x4</b>     | ARB_TIMEOUT_CREDIT_UPDATE | Credit update timeout |
| <b>0x5</b>     | ARB_TIMEOUT_STATE_CHANGE  | State machine timeout |
| <b>0x6-0xE</b> | Reserved                  | Future expansion      |
| <b>0xF</b>     | ARB_TIMEOUT_USER_DEFINED  | User-defined timeout  |

*Completion Events (PktTypeCompletion + PROTOCOL\_ARB)*

| Code       | Event Name             | Description                      |
|------------|------------------------|----------------------------------|
| <b>0x0</b> | ARB_COMPL_GRANT_ISSUED | Grant successfully issued        |
| <b>0x1</b> | ARB_COMPL_ACK_RECEIVED | ACK successfully received        |
| <b>0x2</b> | ARB_COMPL_TRANSACTION  | Complete transaction (grant+ack) |
| <b>0x3</b> | ARB_COMPL_WEIGHT       | Weight update                    |

| Code           | Event Name                | Description              |
|----------------|---------------------------|--------------------------|
| <b>0x4</b>     | T_UPDATE                  | completed                |
| <b>0x5</b>     | ARB_COMPL_CREDIT_CYCLE    | Credit cycle completed   |
| <b>0x6</b>     | ARB_COMPL_FAIRNESS_PERIOD | Fairness analysis period |
| <b>0x7-0xE</b> | ARB_COMPL_BLOCK_PERIOD    | Block period completed   |
| <b>0xF</b>     | Reserved                  | Future expansion         |
|                | ARB_COMPL_USER_DEFINED    | User-defined completion  |

*Threshold Events (PktTypeThreshold + PROTOCOL\_ARB)*

| Code           | Event Name                  | Description                        |
|----------------|-----------------------------|------------------------------------|
| <b>0x0</b>     | ARB_THRESH_REQUEST_LATENCY  | Request-to-grant latency threshold |
| <b>0x1</b>     | ARB_THRESH_ACK_LATENCY      | Grant-to-ACK latency threshold     |
| <b>0x2</b>     | ARB_THRESH_FAIRNESS_DEV     | Fairness deviation threshold       |
| <b>0x3</b>     | ARB_THRESH_ACTIVE_REQUESTS  | Active request count threshold     |
| <b>0x4</b>     | ARB_THRESH_GRANT_RATE       | Grant rate threshold               |
| <b>0x5</b>     | ARB_THRESH_EFFICIENCY       | Grant efficiency threshold         |
| <b>0x6</b>     | ARB_THRESH_CREDIT_LOW       | Low credit threshold               |
| <b>0x7</b>     | ARB_THRESH_WEIGHT_IMBALANCE | Weight imbalance threshold         |
| <b>0x8</b>     | ARB_THRESH_STARVATION_TIME  | Starvation time threshold          |
| <b>0x9-0xE</b> | Reserved                    | Future expansion                   |
| <b>0xF</b>     | ARB_THRESH_USER_DEFINED     | User-defined threshold             |

*Performance Events (PktTypePerf + PROTOCOL\_ARB)*

| Code       | Event Name            | Description        |
|------------|-----------------------|--------------------|
| <b>0x0</b> | ARB_PERF_GRANT_ISSUED | Grant issued event |

| Code           | Event Name                  | Description                   |
|----------------|-----------------------------|-------------------------------|
| <b>0x1</b>     | ARB_PERF_ACK RECEIVED       | ACK received event            |
| <b>0x2</b>     | ARB_PERF_GRANT EFFICIENCY   | Grant completion efficiency   |
| <b>0x3</b>     | ARB_PERF_FAIRNESS_METRIC    | Fairness compliance metric    |
| <b>0x4</b>     | ARB_PERF_THROUGHPUT         | Arbitration throughput        |
| <b>0x5</b>     | ARB_PERF_LATENCY_AVG        | Average latency measurement   |
| <b>0x6</b>     | ARB_PERF_WEIGHT_COMPLIANCE  | Weight compliance metric      |
| <b>0x7</b>     | ARB_PERF_CREDIT_UTILIZATION | Credit utilization efficiency |
| <b>0x8</b>     | ARB_PERF_CLIENT_ACTIVITY    | Per-client activity metric    |
| <b>0x9</b>     | ARB_PERF_STARVATION_COUNT   | Starvation event count        |
| <b>0xA</b>     | ARB_PERF_BLOCK EFFICIENCY   | Block/unblock efficiency      |
| <b>0xB-0xE</b> | Reserved                    | Future expansion              |
| <b>0xF</b>     | ARB_PERF_USER_DEFINED       | User-defined performance      |

*Debug Events (PktTypeDebug + PROTOCOL\_ARB)*

| Code       | Event Name                | Description                  |
|------------|---------------------------|------------------------------|
| <b>0x0</b> | ARB_DEBUG_STATE_CHANGE    | Arbiter state machine change |
| <b>0x1</b> | ARB_DEBUG_MASK_UPDATE     | Round-robin mask update      |
| <b>0x2</b> | ARB_DEBUG_WEIGHT_CHANGE   | Weight configuration change  |
| <b>0x3</b> | ARB_DEBUG_CREDIT_UPDATE   | Credit level update          |
| <b>0x4</b> | ARB_DEBUG_CLIENT_MASK     | Client enable/disable mask   |
| <b>0x5</b> | ARB_DEBUG_PRIORITY_CHANGE | Priority level change        |

| Code           | Event Name                  | Description             |
|----------------|-----------------------------|-------------------------|
| <b>0x6</b>     | ARB_DEBUG_BLOCK_EVENT       | Block/unblock event     |
| <b>0x7</b>     | ARB_DEBUG_QUEUE_STATUS      | Request queue status    |
| <b>0x8</b>     | ARB_DEBUG_COUNT_ER_SNAPSHOT | Counter values snapshot |
| <b>0x9</b>     | ARB_DEBUG_FIFO_STATUS       | FIFO status change      |
| <b>0xA</b>     | ARB_DEBUG_FAIRNESS_STATE    | Fairness tracking state |
| <b>0xB</b>     | ARB_DEBUG_ACK_STATE         | ACK protocol state      |
| <b>0xC-0xE</b> | Reserved                    | Future expansion        |
| <b>0xF</b>     | ARB_DEBUG_USER_DEFINED      | User-defined debug      |

## CORE Protocol Events

*Error Events (PktTypeError + PROTOCOL\_CORE)*

| Code       | Event Name                    | Description                             |
|------------|-------------------------------|-----------------------------------------|
| <b>0x0</b> | CORE_ERR_DESCRIPTOR_MALFORMED | Missing magic number (0x900dc0de)       |
| <b>0x1</b> | CORE_ERR_DESCRIPTOR_BAD_ADDR  | Invalid descriptor address              |
| <b>0x2</b> | CORE_ERR_DATA_BAD_ADDR        | Invalid data address (fetch or runtime) |
| <b>0x3</b> | CORE_ERR_FLAG_COMPARISON      | Flag mask/compare mismatch              |
| <b>0x4</b> | CORE_ERR_CREDIT_UNDERFLOW     | Credit system violation                 |
| <b>0x5</b> | CORE_ERR_STATE_MACHINE        | Invalid FSM state transition            |
| <b>0x6</b> | CORE_ERR_DESCRIPTOR_ENGINE    | Descriptor engine FSM error             |
| <b>0x7</b> | CORE_ERR_FLAG_ENGINE          | Flag engine FSM error                   |
| <b>0x8</b> | CORE_ERR_PROGRAM_ENGINE       | Program engine FSM error                |

| Code           | Event Name                  | Description                |
|----------------|-----------------------------|----------------------------|
|                | E                           |                            |
| <b>0x9</b>     | CORE_ERR_DATA_ENGINE        | Data engine error          |
| <b>0xA</b>     | CORE_ERR_CHANNEL_INVALID_ID | Invalid channel ID         |
| <b>0xB</b>     | CORE_ERR_CONTROL_VIOLATION  | Control register violation |
| <b>0xC-0xE</b> | Reserved                    | Future expansion           |
| <b>0xF</b>     | CORE_ERR_USER_DEFINED       | User-defined error         |

*Timeout Events (PktTypeTimeout + PROTOCOL\_CORE)*

| Code           | Event Name                    | Description                   |
|----------------|-------------------------------|-------------------------------|
| <b>0x0</b>     | CORE_TIMEOUT_DESCRIPTOR_FETCH | Descriptor fetch timeout      |
| <b>0x1</b>     | CORE_TIMEOUT_FLAG_RETRY       | Flag comparison retry timeout |
| <b>0x2</b>     | CORE_TIMEOUT_PROGRAM_WRITE    | Program write timeout         |
| <b>0x3</b>     | CORE_TIMEOUT_DATA_TRANSFER    | Data transfer timeout         |
| <b>0x4</b>     | CORE_TIMEOUT_CREDIT_WAIT      | Credit wait timeout           |
| <b>0x5</b>     | CORE_TIMEOUT_CONTROL_WAIT     | Control enable wait timeout   |
| <b>0x6</b>     | CORE_TIMEOUT_ENGINE_RESPONSE  | Sub-engine response timeout   |
| <b>0x7</b>     | CORE_TIMEOUT_STATE_TRANSITION | FSM state transition timeout  |
| <b>0x8-0xE</b> | Reserved                      | Future expansion              |
| <b>0xF</b>     | CORE_TIMEOUT_USER_DEFINED     | User-defined timeout          |

*Completion Events (PktTypeCompletion + PROTOCOL\_CORE)*

| Code       | Event Name             | Description                    |
|------------|------------------------|--------------------------------|
| <b>0x0</b> | CORE_COMPL_DESCRIPTOR_ | Descriptor successfully loaded |

| Code           | Event Name                   | Description                 |
|----------------|------------------------------|-----------------------------|
|                | LOADED                       |                             |
| <b>0x1</b>     | CORE_COMPL_DESCRIPTOR_CHAIN  | Descriptor chain completed  |
| <b>0x2</b>     | CORE_COMPL_FLAG_MATCHED      | Flag comparison successful  |
| <b>0x3</b>     | CORE_COMPL_PROGRAM_COMPLETED | Post-programming completed  |
| <b>0x4</b>     | CORE_COMPL_DATA_TRANSFER     | Data transfer completed     |
| <b>0x5</b>     | CORE_COMPL_CREDIT_CYCLE      | Credit cycle completed      |
| <b>0x6</b>     | CORE_COMPL_CHANNEL_COMPLETE  | Channel processing complete |
| <b>0x7</b>     | CORE_COMPL_ENGINE_READY      | Sub-engine ready            |
| <b>0x8-0xE</b> | Reserved                     | Future expansion            |
| <b>0xF</b>     | CORE_COMPL_USER_DEFINED      | User-defined completion     |

*Threshold Events (PktTypeThreshold + PROTOCOL\_CORE)*

| Code       | Event Name                   | Description                      |
|------------|------------------------------|----------------------------------|
| <b>0x0</b> | CORE_THRESH_DESCRIPTOR_QUEUE | Descriptor queue depth threshold |
| <b>0x1</b> | CORE_THRESH_CREDIT_LOW       | Credit low threshold             |
| <b>0x2</b> | CORE_THRESH_FLAG_RETRY_COUNT | Flag retry count threshold       |
| <b>0x3</b> | CORE_THRESH_LATENCY          | Processing latency threshold     |
| <b>0x4</b> | CORE_THRESH_ERROR_RATE       | Error rate threshold             |
| <b>0x5</b> | CORE_THRESH_THROUGHPUT       | Throughput threshold             |
| <b>0x6</b> | CORE_THRESH_ACTIVE_CHANNELS  | Active channel count threshold   |
| <b>0x7</b> | CORE_THRESH_PROGRAM_LATENCY  | Program write latency threshold  |

| Code           | Event Name               | Description                  |
|----------------|--------------------------|------------------------------|
| <b>0x8</b>     | CORE_THRESH_DATA_RATE    | Data transfer rate threshold |
| <b>0x9-0xE</b> | Reserved                 | Future expansion             |
| <b>0xF</b>     | CORE_THRESH_USER_DEFINED | User-defined threshold       |

*Performance Events (PktTypePerf + PROTOCOL\_CORE)*

| Code           | Event Name                   | Description                  |
|----------------|------------------------------|------------------------------|
| <b>0x0</b>     | CORE_PERF_END_OF_DATA        | Stream continuation signal   |
| <b>0x1</b>     | CORE_PERF_END_OF_STREAM      | Stream termination signal    |
| <b>0x2</b>     | CORE_PERF_ENTERING_IDLE      | FSM returning to idle        |
| <b>0x3</b>     | CORE_PERF_CREDIT_INCREMENTED | Credit added by software     |
| <b>0x4</b>     | CORE_PERF_CREDIT_EXHAUSTED   | Credit blocking execution    |
| <b>0x5</b>     | CORE_PERF_STATE_TRANSITION   | FSM state change             |
| <b>0x6</b>     | CORE_PERF_DESCRIPTOR_ACTIVE  | Data processing started      |
| <b>0x7</b>     | CORE_PERF_FLAG_RETRY         | Flag comparison retry        |
| <b>0x8</b>     | CORE_PERF_CHANNEL_ENABLE     | Channel enabled by software  |
| <b>0x9</b>     | CORE_PERF_CHANNEL_DISABLE    | Channel disabled by software |
| <b>0xA</b>     | CORE_PERF_CREDIT_UTILIZATION | Credit utilization metric    |
| <b>0xB</b>     | CORE_PERF_PROCESSING_LATENCY | Total processing latency     |
| <b>0xC</b>     | CORE_PERF_QUEUE_DEPTH        | Current queue depth          |
| <b>0xD-0xE</b> | Reserved                     | Future expansion             |

| Code       | Event Name             | Description              |
|------------|------------------------|--------------------------|
| <b>0xF</b> | CORE_PERF_USER_DEFINED | User-defined performance |

*Debug Events (PktTypeDebug + PROTOCOL\_CORE)*

| Code           | Event Name                      | Description                 |
|----------------|---------------------------------|-----------------------------|
| <b>0x0</b>     | CORE_DEBUG_FSM_STATE_CHANGE     | Descriptor FSM state change |
| <b>0x1</b>     | CORE_DEBUG_DESCRIPTOR_CONTENT   | Descriptor content trace    |
| <b>0x2</b>     | CORE_DEBUG_FLAG_ENGINE_STATE    | Flag engine state trace     |
| <b>0x3</b>     | CORE_DEBUG_PROGRAM_ENGINE_STATE | Program engine state trace  |
| <b>0x4</b>     | CORE_DEBUG_CREDIT_OPERATION     | Credit system operation     |
| <b>0x5</b>     | CORE_DEBUG_CONTROL_REGISTER     | Control register access     |
| <b>0x6</b>     | CORE_DEBUG_ENGINE_HANDSHAKE     | Engine handshake trace      |
| <b>0x7</b>     | CORE_DEBUG_QUEUE_STATUS         | Queue status change         |
| <b>0x8</b>     | CORE_DEBUG_COUNTER_SNAPSHOT     | Counter values snapshot     |
| <b>0x9</b>     | CORE_DEBUG_ADDRESS_TRACE        | Address progression trace   |
| <b>0xA</b>     | CORE_DEBUG_PAYLOAD_TRACE        | Payload content trace       |
| <b>0xB-0xE</b> | Reserved                        | Future expansion            |
| <b>0xF</b>     | CORE_DEBUG_USER_DEFINED         | User-defined debug          |

# Memory Architecture and Packet Routing

## Two-Tier Memory Architecture

### *Local Error/Interrupt Memory*

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

### *Configurable External Memory*

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

## Routing Configuration

### *Base and Limit Registers*

| Register Set             | Purpose          | Configuration                           |
|--------------------------|------------------|-----------------------------------------|
| <b>Completion Config</b> | Type 0x1 routing | base_addr, limit_addr, enable, priority |
| <b>Threshold Config</b>  | Type 0x2 routing | base_addr, limit_addr, enable,          |

| Register Set              | Purpose          | Configuration                              |
|---------------------------|------------------|--------------------------------------------|
| <b>Performance Config</b> | Type 0x4 routing | priority<br>base_addr, limit_addr, enable, |
| <b>Debug Config</b>       | Type 0xF routing | priority<br>base_addr, limit_addr, enable, |

### *Routing Decision Logic*

| Packet Type              | Destination     | Address Calculation                   |
|--------------------------|-----------------|---------------------------------------|
| <b>Error (0x0)</b>       | Local Memory    | local_error_write_pointer             |
| <b>Timeout (0x3)</b>     | Local Memory    | local_error_write_pointer             |
| <b>Completion (0x1)</b>  | External Memory | completion_config.base_addr + offset  |
| <b>Performance (0x4)</b> | External Memory | performance_config.base_addr + offset |
| <b>Debug (0xF)</b>       | External Memory | debug_config.base_addr + offset       |

### *Address Space Management*

#### *Memory Layout Example*

| Address Range                    | Usage               | Description              |
|----------------------------------|---------------------|--------------------------|
| <b>0x1000_0000 - 0x1000_FFFF</b> | Local Error Memory  | Immediate access storage |
| <b>0x2000_0000 - 0x2001_FFFF</b> | Performance Packets | External bulk storage    |
| <b>0x2010_0000 - 0x2011_FFFF</b> | Completion Packets  | External bulk storage    |
| <b>0x2020_0000 - 0x202F_FFFF</b> | Debug Packets       | External bulk storage    |

### *Transaction State and Bus Transaction Structure*

#### *Transaction State Enumeration*

| State                   | Value  | Description                                                | Usage          |
|-------------------------|--------|------------------------------------------------------------|----------------|
| <b>TRANS_EMPT</b>       | 3'b000 | Unused entry                                               | Available slot |
| <b>TRANS_ADDR_PHASE</b> | 3'b001 | Address phase active (AXI) /<br>Packet sent (MNOC) / Setup | Initial phase  |

| State                     | Value  | Description                                                            | Usage                 |
|---------------------------|--------|------------------------------------------------------------------------|-----------------------|
|                           |        | phase (APB)                                                            |                       |
| <b>TRANS_DATA_PHASE</b>   | 3'b010 | Data phase active (AXI) / Waiting for ACK (MNOC) / Access phase (APB)  | Data transfer         |
| <b>TRANS_RESP_PHASE</b>   | 3'b011 | Response phase active (AXI) / ACK received (MNOC) / Enable phase (APB) | Response handling     |
| <b>TRANS_COMPLETE</b>     | 3'b100 | Transaction complete                                                   | Successful completion |
| <b>TRANS_ERROR</b>        | 3'b101 | Transaction has error                                                  | Error condition       |
| <b>TRANS_ORPHANED</b>     | 3'b110 | Orphaned transaction                                                   | Missing components    |
| <b>TRANS_CREDIT_STALL</b> | 3'b111 | Credit stall (MNOC only)                                               | MNOC-specific stall   |

#### *Enhanced Transaction Structure*

| Field           | Width | Description                                                  | Protocol Usage |
|-----------------|-------|--------------------------------------------------------------|----------------|
| <b>valid</b>    | 1     | Entry is valid                                               | All protocol's |
| <b>protocol</b> | 3     | Protocol type<br>(AXI/MNOC/APB/ARB/CORE)                     | All protocols  |
| <b>state</b>    | 3     | Transaction state                                            | All protocols  |
| <b>id</b>       | 32    | Transaction ID (AXI) / Sequence (MNOC) / PSEL encoding (APB) | All protocols  |
| <b>addr</b>     | 64    | Transaction address / Channel addr / PADDR                   | All protocols  |
| <b>len</b>      | 8     | Burst length (AXI) / Packet count (MNOC) / Always 0 (APB)    | AXI, MNOC      |
| <b>size</b>     | 3     | Access size (AXI) / Reserved (MNOC) / Transfer size (APB)    | AXI, APB       |
| <b>burst</b>    | 2     | Burst type (AXI) /                                           | All protocols  |

| Field | Width | Description                               | Protocol Usage |
|-------|-------|-------------------------------------------|----------------|
|       |       | Payload type (MNOC) /<br>PPROT[1:0] (APB) |                |

#### *Phase Completion Flags*

| Flag               | Description                | Protocol Usage |
|--------------------|----------------------------|----------------|
| <b>cmd_receiv</b>  | Address phase received /   | All protocols  |
| <b>ed</b>          | Packet sent / Setup phase  |                |
| <b>data_starte</b> | Data phase started / ACK   | All protocols  |
| <b>d</b>           | expected / Access phase    |                |
| <b>data_comp</b>   | Data phase completed / ACK | All protocols  |
| <b>leted</b>       | received / Enable phase    |                |
| <b>resp_recei</b>  | Response received / Final  | All protocols  |
| <b>ved</b>         | ACK / PREADY asserted      |                |

#### *Protocol-Specific Tracking Fields*

| Field                  | Width | Description                                            | Protocol      |
|------------------------|-------|--------------------------------------------------------|---------------|
| <b>channel</b>         | 6     | Channel ID (AXI ID / MNOC channel / PSEL bit position) | All protocols |
| <b>eos_seen</b>        | 1     | EOS marker seen                                        | MNOC only     |
| <b>parity_error</b>    | 1     | Parity error detected                                  | MNOC only     |
| <b>credit_at_sta</b>   | 8     | Credits available at start                             | MNOC only     |
| <b>rt</b>              |       |                                                        |               |
| <b>retry_count</b>     | 3     | Number of retries                                      | MNOC only     |
| <b>desc_addr_match</b> | 1     | Descriptor address match detected                      | AXI only      |
| <b>data_addr_match</b> | 1     | Data address match detected                            | AXI only      |
| <b>apb_phase</b>       | 2     | Current APB phase                                      | APB only      |
| <b>pslverr_seen</b>    | 1     | PSLVERR detected                                       | APB only      |
| <b>pprot_value</b>     | 3     | PPROT value                                            | APB only      |
| <b>pstrb_value</b>     | 4     | PSTRB value for writes                                 | APB only      |
| <b>arb_grant_id</b>    | 8     | Current grant ID                                       | ARB only      |
| <b>arb_weight</b>      | 8     | Current weight value                                   | ARB only      |

| Field                  | Width | Description             | Protocol  |
|------------------------|-------|-------------------------|-----------|
| <b>core_fsm_st ate</b> | 3     | Current CORE FSM state  | CORE only |
| <b>core_channel_id</b> | 6     | CORE channel identifier | CORE only |

### APB Transaction Phases

| Phase                   | Value | Description                       |
|-------------------------|-------|-----------------------------------|
| <b>APB_PHASE_IDLE</b>   | 2'b00 | Bus idle                          |
| <b>APB_PHASE_SETUP</b>  | 2'b01 | Setup phase (PSEL asserted)       |
| <b>APB_PHASE_ACCESS</b> | 2'b10 | Access phase (PENABLE asserted)   |
| <b>APB_PHASE_ENABLE</b> | 2'b11 | Enable phase (waiting for PREADY) |

### APB Protection Types

| Protection                  | Value  | Description        |
|-----------------------------|--------|--------------------|
| <b>APB_PROT_NORMAL</b>      | 3'b000 | Normal access      |
| <b>APB_PROT_PRIVILEGED</b>  | 3'b001 | Privileged access  |
| <b>APB_PROT_SECURE</b>      | 3'b010 | Secure access      |
| <b>APB_PROT_INSTRUCTION</b> | 3'b100 | Instruction access |

### MNOC Payload Types

| Payload                    | Value | Description |
|----------------------------|-------|-------------|
| <b>MNOC_PAYLOAD_CONFIG</b> | 2'b00 | CONFIG_PKT  |
| <b>MNOC_PAYLOAD_TS</b>     | 2'b01 | TS_PKT      |
| <b>MNOC_PAYLOAD_RDA</b>    | 2'b10 | RDA_PKT     |
| <b>MNOC_PAYLOAD_RAW</b>    | 2'b11 | RAW_PKT     |

| Payload   | Value | Description |
|-----------|-------|-------------|
| <b>AW</b> |       |             |

### MNOC ACK Types

| ACK Type                    | Value | Description      |
|-----------------------------|-------|------------------|
| <b>MNOC_ACK_STOP</b>        | 2'b00 | MSAP_STOP        |
| <b>MNOC_ACK_START</b>       | 2'b01 | MSAP_START       |
| <b>MNOC_ACK_CREDI T_ON</b>  | 2'b10 | MSAP_CREDIT_ON   |
| <b>MNOC_ACK_STOP_AT_EOS</b> | 2'b11 | MSAP_STOP_AT_EOS |

### ARB State Types

| State                       | Value  | Description                   |
|-----------------------------|--------|-------------------------------|
| <b>ARB_STATE_IDLE</b>       | 3'b000 | Idle state                    |
| <b>ARB_STATE_ARBIT RATE</b> | 3'b001 | Performing arbitration        |
| <b>ARB_STATE_GRANT</b>      | 3'b010 | Grant issued, waiting for ACK |
| <b>ARB_STATE_BLOCKED</b>    | 3'b011 | Arbitration blocked           |
| <b>ARB_STATE_WEIGHT_UPD</b> | 3'b100 | Weight update in progress     |
| <b>ARB_STATE_ERROR</b>      | 3'b101 | Error state                   |

### CORE State Types

| State                            | Value  | Description                |
|----------------------------------|--------|----------------------------|
| <b>CORE_STATE_IDLE</b>           | 3'b000 | Idle state                 |
| <b>CORE_STATE_DESC_FETCH</b>     | 3'b001 | Fetching descriptor        |
| <b>CORE_STATE_FLAG_CHECK</b>     | 3'b010 | Checking flag condition    |
| <b>CORE_STATE_PROG_RAM_WRITE</b> | 3'b011 | Writing program RAM_WRITE  |
| <b>CORE_STATE_DATA_TRANSFER</b>  | 3'b100 | Transferring data TRANSFER |

| State                  | Value  | Description         |
|------------------------|--------|---------------------|
| <b>CORE_STATE_CRED</b> | 3'b101 | Waiting for credits |
| <b>IT_WAIT</b>         |        |                     |
| <b>CORE_STATE_ERRO</b> | 3'b110 | Error state         |
| <b>R</b>               |        |                     |

## Configuration and Control

### Monitor Configuration Registers

#### *Global Configuration*

| Field                        | Width | Description                |
|------------------------------|-------|----------------------------|
| <b>monitor_enable</b>        | 1     | Global monitor enable      |
| <b>error_local_enable</b>    | 1     | Enable local error storage |
| <b>external_route_enable</b> | 1     | Enable external routing    |
| <b>unit_id</b>               | 4     | Unit identifier            |
| <b>agent_id</b>              | 8     | Agent identifier           |
| <b>packet_type_enable</b>    | 16    | Per-type enable bits       |
| <b>s</b>                     |       |                            |

#### *Packet Type Enable Mapping*

| Bit      | Enable                    | Description                  |
|----------|---------------------------|------------------------------|
| <b>0</b> | PKT_ENABLE_ERRO<br>R      | Enable error packets         |
| <b>1</b> | PKT_ENABLE_COMP<br>LETION | Enable completion packets    |
| <b>2</b> | PKT_ENABLE_THRES<br>HOLD  | Enable threshold packets     |
| <b>3</b> | PKT_ENABLE_TIME<br>OUT    | Enable timeout packets       |
| <b>4</b> | PKT_ENABLE_PERF           | Enable performance packets   |
| <b>5</b> | PKT_ENABLE_CREDI<br>T     | Enable credit packets (MNOC) |

| Bit | Enable                    | Description                      |
|-----|---------------------------|----------------------------------|
| 6   | PKT_ENABLE_CHAN<br>NEL    | Enable channel<br>packets (MNOC) |
| 7   | PKT_ENABLE_STREA<br>M     | Enable stream<br>packets (MNOC)  |
| 8   | PKT_ENABLE_ADDR_<br>MATCH | Enable address<br>match (AXI)    |
| 9   | PKT_ENABLE_APB            | Enable APB packets               |
| 15  | PKT_ENABLE_DEBU<br>G      | Enable debug<br>packets          |

## Protocol-Specific Configuration

### AXI Monitor Configuration

| Field                                    | Width | Description                       |
|------------------------------------------|-------|-----------------------------------|
| <b>active_trans_thresh</b><br><b>old</b> | 16    | Active transaction<br>threshold   |
| <b>latency_threshold</b>                 | 32    | Latency threshold<br>(cycles)     |
| <b>addr_timeout_cnt</b>                  | 4     | Address timeout<br>count          |
| <b>data_timeout_cnt</b>                  | 4     | Data timeout count                |
| <b>resp_timeout_cnt</b>                  | 4     | Response timeout<br>count         |
| <b>burst_boundary_ch<br/>eck</b>         | 1     | Enable burst<br>boundary checking |
| <b>address_match_ena<br/>ble</b>         | 1     | Enable address<br>matching        |
| <b>desc_addr_match_b<br/>ase</b>         | 64    | Descriptor address<br>match base  |
| <b>desc_addr_match_</b><br><b>mask</b>   | 64    | Descriptor address<br>match mask  |
| <b>data_addr_match_b<br/>ase</b>         | 64    | Data address match<br>base        |
| <b>data_addr_match_</b><br><b>mask</b>   | 64    | Data address match<br>mask        |

### *MNOC Monitor Configuration*

| Field                            | Width | Description                |
|----------------------------------|-------|----------------------------|
| <b>credit_low_threshold</b>      | 8     | Credit low threshold       |
| <b>packet_rate_threshold</b>     | 16    | Packet rate threshold      |
| <b>max_route_hops</b>            | 8     | Maximum routing hops       |
| <b>enable_credit_tracking</b>    | 1     | Enable credit tracking     |
| <b>enable_deadlock_detection</b> | 1     | Enable deadlock detection  |
| <b>deadlock_timeout</b>          | 4     | Deadlock detection timeout |

### *ARB Monitor Configuration*

| Field                       | Width | Description                    |
|-----------------------------|-------|--------------------------------|
| <b>grant_timeout_cnt</b>    | 16    | Grant ACK timeout count        |
| <b>fairness_window</b>      | 32    | Fairness analysis window       |
| <b>weight_update_enable</b> | 1     | Enable weight tracking         |
| <b>starvation_threshold</b> | 16    | Starvation detection threshold |
| <b>efficiency_threshold</b> | 8     | Grant efficiency threshold     |

### *CORE Monitor Configuration*

| Field                         | Width | Description                    |
|-------------------------------|-------|--------------------------------|
| <b>descriptor_timeout_cnt</b> | 16    | Descriptor fetch timeout count |
| <b>flag_retry_limit</b>       | 8     | Maximum flag retry count       |
| <b>credit_low_threshold</b>   | 8     | Credit low threshold           |

| Field                      | Width | Description        |
|----------------------------|-------|--------------------|
| <b>processing_timeout</b>  | 32    | Processing timeout |
| <b>_cnt</b>                |       | count              |
| <b>enable_descriptor_t</b> | 1     | Enable descriptor  |
| <b>race</b>                |       | content tracing    |
| <b>enable_fsm_trace</b>    | 1     | Enable FSM state   |
|                            |       | tracing            |

## Validation Requirements

### Functional Validation

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

### Performance Validation

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

## Error Handling Validation

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

## Usage Examples

### Creating Monitor Packets

| Packet Type            | Example Usage                                                        |
|------------------------|----------------------------------------------------------------------|
| <b>AXI Error</b>       | Protocol=AXI, Type=Error,<br>Code=AXI_ERR_RESP_SLVERR                |
| <b>MNOC Credit</b>     | Protocol=MNOC, Type=Credit,<br>Code=MNOC_CREDIT_EXHAUSTED            |
| <b>APB Performance</b> | Protocol=APB, Type=Performance,<br>Code=APB_PERF_TOTAL_LATENCY       |
| <b>ARB Threshold</b>   | Protocol=ARB, Type=Threshold,<br>Code=ARB_THRESH_FAIRNESS_DEV        |
| <b>CORE Completion</b> | Protocol=CORE, Type=Completion,<br>Code=CORE_COMPL_DESCRIPTOR_LOADED |

### Packet Decoding

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

## Monitor Bus Packet Helper Functions

### Packet Field Extraction

| Function                            | Return Type     | Description                 |
|-------------------------------------|-----------------|-----------------------------|
| <code>get_packet_type(pkt)</code>   | logic [3:0]     | Extract packet type [63:60] |
| <code>get_protocol_type(pkt)</code> | protocol_type_t | Extract protocol [59:57]    |
| <code>get_event_code(pkt)</code>    | logic [3:0]     | Extract event code [56:53]  |
| <code>get_channel_id(pkt)</code>    | logic [5:0]     | Extract channel ID [52:47]  |
| <code>get_unit_id(pkt)</code>       | logic [3:0]     | Extract unit ID [46:43]     |
| <code>get_agent_id(pkt)</code>      | logic [7:0]     | Extract agent ID [42:35]    |
| <code>get_event_data(pkt)</code>    | logic [34:0]    | Extract event data [34:0]   |

### Packet Creation Function

| Function                             | Parameters                                                                   | Description                   |
|--------------------------------------|------------------------------------------------------------------------------|-------------------------------|
| <code>create_monitor_packet()</code> | 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                       |
|---------------------------------------------|------------------------|-----------------------------------|
| <code>create_axi_error_event()</code>       | axi_error_code_t       | Create AXI error event code       |
| <code>create_axi_timeout_event()</code>     | axi_timeout_code_t     | Create AXI timeout event code     |
| <code>create_axi_completion_event()</code>  | axi_completion_code_t  | Create AXI completion event code  |
| <code>create_axi_threshold_event()</code>   | axi_threshold_code_t   | Create AXI threshold event code   |
| <code>create_axi_performance_event()</code> | axi_performance_code_t | Create AXI performance event code |
| <code>create_axi_addr_match_event()</code>  | axi_addr_match_code_t  | Create AXI address match          |

| Function                              | Parameter              | Description                       |
|---------------------------------------|------------------------|-----------------------------------|
| <b>tch_event()</b>                    |                        | event code                        |
| <b>create_axi_debug_event()</b>       | axi_debug_code_t       | Create AXI debug event code       |
| <b>create_apb_error_event()</b>       | apb_error_code_t       | Create APB error event code       |
| <b>create_apb_timeout_event()</b>     | apb_timeout_code_t     | Create APB timeout event code     |
| <b>create_apb_completion_event()</b>  | apb_completion_code_t  | Create APB completion event code  |
| <b>create_mnoc_error_event()</b>      | mnoc_error_code_t      | Create MNOC error event code      |
| <b>create_mnoc_timeout_event()</b>    | mnoc_timeout_code_t    | Create MNOC timeout event code    |
| <b>create_mnoc_completion_event()</b> | mnoc_completion_code_t | Create MNOC completion event code |
| <b>create_mnoc_credit_event()</b>     | mnoc_credit_code_t     | Create MNOC credit event code     |
| <b>create_mnoc_channel_event()</b>    | mnoc_channel_code_t    | Create MNOC channel event code    |
| <b>create_mnoc_stream_event()</b>     | mnoc_stream_code_t     | Create MNOC stream event code     |
| <b>create_arb_error_event()</b>       | arb_error_code_t       | Create ARB error event code       |
| <b>create_arb_timeout_event()</b>     | arb_timeout_code_t     | Create ARB timeout event code     |
| <b>create_arb_completion_event()</b>  | arb_completion_code_t  | Create ARB completion event code  |
| <b>create_arb_threshold_event()</b>   | arb_threshold_code_t   | Create ARB threshold event code   |
| <b>create_arb_performance_event()</b> | arb_performance_code_t | Create ARB performance event code |
| <b>create_arb_debug_event()</b>       | arb_debug_code_t       | Create ARB debug event code       |
| <b>create_core_error_event()</b>      | core_error_code_t      | Create CORE error event           |

| Function                               | Parameter               | Description                        |
|----------------------------------------|-------------------------|------------------------------------|
| <b>vent()</b>                          |                         | code                               |
| <b>create_core_timeout_event()</b>     | core_timeout_code_t     | Create CORE timeout event code     |
| <b>create_core_completion_event()</b>  | core_completion_code_t  | Create CORE completion event code  |
| <b>create_core_threshold_event()</b>   | core_threshold_code_t   | Create CORE threshold event code   |
| <b>create_core_performance_event()</b> | core_performance_code_t | Create CORE performance event code |
| <b>create_core_debug_event()</b>       | core_debug_code_t       | Create CORE debug event code       |

#### Validation Functions

| Function                                | Parameters                        | Description                                      |
|-----------------------------------------|-----------------------------------|--------------------------------------------------|
| <b>is_valid_event_for_packet_type()</b> | packet_type, protocol, event_code | Validate event code for packet type and protocol |

#### String Functions for Debugging

| Function                      | Parameter                         | Description                        |
|-------------------------------|-----------------------------------|------------------------------------|
| <b>get_axi_error_name()</b>   | axi_error_code_t                  | Get human-readable AXI error name  |
| <b>get_arb_error_name()</b>   | arb_error_code_t                  | Get human-readable ARB error name  |
| <b>get_core_error_name()</b>  | core_error_code_t                 | Get human-readable CORE error name |
| <b>get_packet_type_name()</b> | logic [3:0]                       | Get packet type name string        |
| <b>get_protocol_name()</b>    | protocol_type_t                   | Get protocol name string           |
| <b>get_event_name()</b>       | packet_type, protocol, event_code | Get comprehensive event name       |

## Debug and Monitoring Signals

#### Essential Debug Signals

| Signal                    | Width   | Purpose          |
|---------------------------|---------|------------------|
| <b>debug_packet_count</b> | 32 × 16 | Packet count per |

| Signal                             | Width         | Purpose                     |
|------------------------------------|---------------|-----------------------------|
| <b>ts</b>                          |               | type                        |
| <b>debug_protocol_counts</b>       | $32 \times 5$ | Packet count per protocol   |
| <b>debug_error_count_s</b>         | 32            | Total error packet count    |
| <b>debug_local_memory_level</b>    | 16            | Local memory usage level    |
| <b>debug_external_memory_level</b> | 16            | External memory usage level |

## Performance Counters

| Counter                        | Width | Purpose                         |
|--------------------------------|-------|---------------------------------|
| <b>total_packets_processed</b> | 32    | Total packets processed         |
| <b>packets_dropped</b>         | 32    | Packets dropped due to overflow |
| <b>routing_errors</b>          | 32    | Routing configuration errors    |
| <b>memory_full_events</b>      | 32    | Memory full occurrences         |

## Protocol Coverage Summary

### Complete Protocol Event Matrix

| Protocol    | Error     | Timeout   | Completion | Threshold | Performance | Debug     | Protocol-Specific                    |
|-------------|-----------|-----------|------------|-----------|-------------|-----------|--------------------------------------|
| <b>AXI</b>  | YES<br>16 | YES<br>16 | YES 16     | YES 16    | YES 16      | YES<br>16 | AddrMatch<br>YES 16                  |
| <b>MNOC</b> | YES<br>16 | YES<br>16 | YES 16     | NO 0      | NO 0        | NO<br>0   | Credit/<br>Channel/<br>Stream YES 48 |
| <b>APB</b>  | YES<br>16 | YES<br>16 | YES 16     | YES 16    | YES 16      | YES<br>16 | None                                 |
| <b>ARB</b>  | YES<br>16 | YES<br>16 | YES 16     | YES 16    | YES 16      | YES<br>16 | None                                 |

| Protocol    | Error     | Timeout   | Completion | Threshold | Performance | Debug     | Protocol-Specific |
|-------------|-----------|-----------|------------|-----------|-------------|-----------|-------------------|
| <b>CORE</b> | YES<br>16 | YES<br>16 | YES 16     | YES 16    | YES 16      | YES<br>16 | None              |

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

## STREAM Register Map

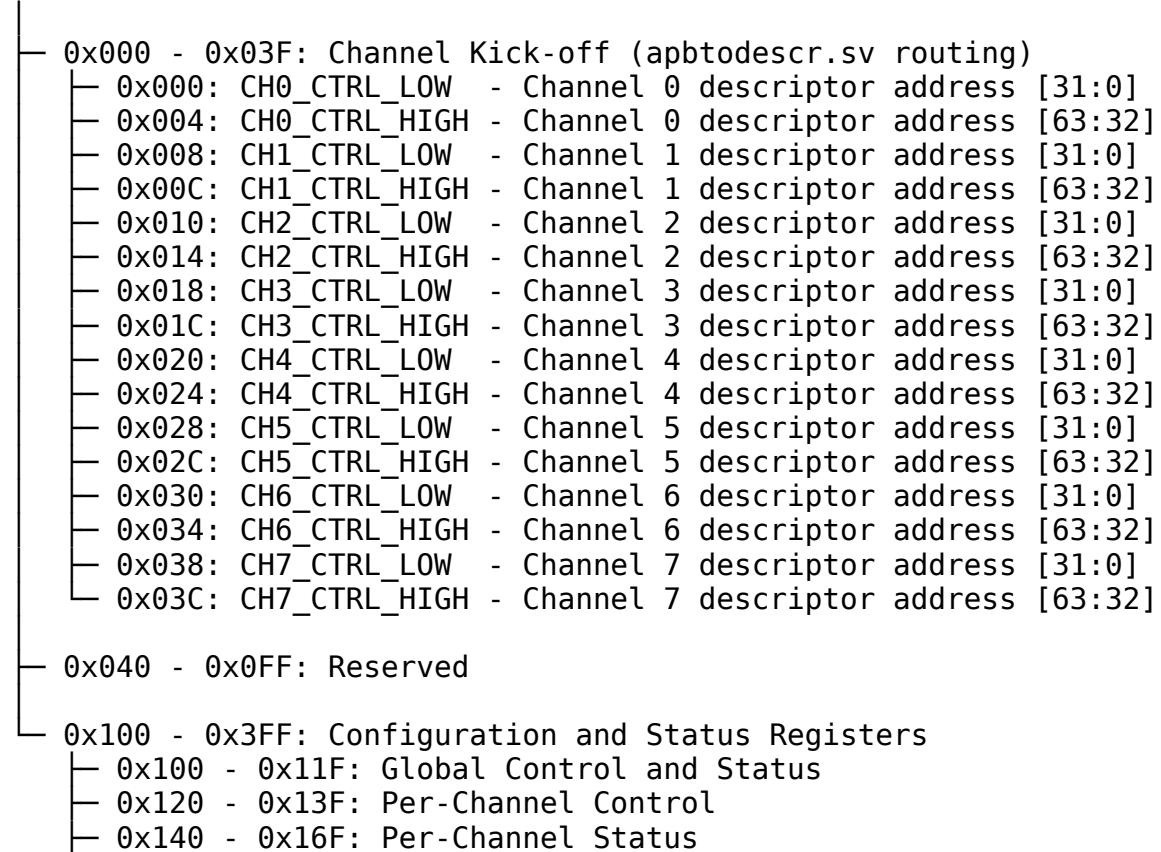
### Overview

The STREAM DMA engine register interface consists of two distinct regions:

1. **Channel Kick-off Registers** (0x000 - 0x03F) - Direct routing to descriptor engines
2. **Configuration and Status Registers** (0x100 - 0xFFFF) - PeakRDL-generated register file

### Address Space Layout

Base Address (configurable parameter)



- └─ 0x170 - 0x17F: Engine Completion and Error Status
- └─ 0x180 - 0x1FF: Monitor FIFO Status
- └─ 0x200 - 0x21F: Scheduler Configuration
- └─ 0x220 - 0x23F: Descriptor Engine Configuration
- └─ 0x240 - 0x25F: Descriptor AXI Monitor Configuration
- └─ 0x260 - 0x27F: Read Engine AXI Monitor Configuration
- └─ 0x280 - 0x29F: Write Engine AXI Monitor Configuration
- └─ 0x2A0 - 0x2AF: AXI Transfer Configuration
- └─ 0x2B0 - 0x2BF: Performance Profiler Configuration

## Register Details

### Channel Kick-off Registers (0x000 - 0x03F)

These registers are **NOT** traditional registers. Writes are routed directly to descriptor engine APB ports via `apbtodescr.sv`.

**Note:** Descriptor addresses are 64-bit (ADDR\_WIDTH parameter, default 64). On 32-bit APB bus, each channel requires TWO registers (LOW/HIGH).

| Offset |            | Type | Reset | Description                          |
|--------|------------|------|-------|--------------------------------------|
| t      | Register   | e    | t     |                                      |
| 0x00   | CH0_CTRL_L | WO   | N/A   | Channel 0 descriptor address [31:0]  |
| 0      | OW         |      |       |                                      |
| 0x00   | CH0_CTRL_H | WO   | N/A   | Channel 0 descriptor address [63:32] |
| 4      | IGH        |      |       |                                      |
| 0x00   | CH1_CTRL_L | WO   | N/A   | Channel 1 descriptor address [31:0]  |
| 8      | OW         |      |       |                                      |
| 0x00   | CH1_CTRL_H | WO   | N/A   | Channel 1 descriptor address [63:32] |
| C      | IGH        |      |       |                                      |
| 0x01   | CH2_CTRL_L | WO   | N/A   | Channel 2 descriptor address [31:0]  |
| 0      | OW         |      |       |                                      |
| 0x01   | CH2_CTRL_H | WO   | N/A   | Channel 2 descriptor address [63:32] |
| 4      | IGH        |      |       |                                      |
| 0x01   | CH3_CTRL_L | WO   | N/A   | Channel 3 descriptor address [31:0]  |
| 8      | OW         |      |       |                                      |
| 0x01   | CH3_CTRL_H | WO   | N/A   | Channel 3 descriptor address [63:32] |
| C      | IGH        |      |       |                                      |
| 0x02   | CH4_CTRL_L | WO   | N/A   | Channel 4 descriptor address [31:0]  |
| 0      | OW         |      |       |                                      |
| 0x02   | CH4_CTRL_H | WO   | N/A   | Channel 4 descriptor address [63:32] |

| Offset |            | Type | Reset | Description                          |
|--------|------------|------|-------|--------------------------------------|
| t      | Register   | e    | t     |                                      |
| 4      | IGH        |      |       |                                      |
| 0x02   | CH5_CTRL_L | WO   | N/A   | Channel 5 descriptor address [31:0]  |
| 8      | OW         |      |       |                                      |
| 0x02   | CH5_CTRL_H | WO   | N/A   | Channel 5 descriptor address [63:32] |
| C      | IGH        |      |       |                                      |
| 0x03   | CH6_CTRL_L | WO   | N/A   | Channel 6 descriptor address [31:0]  |
| 0      | OW         |      |       |                                      |
| 0x03   | CH6_CTRL_H | WO   | N/A   | Channel 6 descriptor address [63:32] |
| 4      | IGH        |      |       |                                      |
| 0x03   | CH7_CTRL_L | WO   | N/A   | Channel 7 descriptor address [31:0]  |
| 8      | OW         |      |       |                                      |
| 0x03   | CH7_CTRL_H | WO   | N/A   | Channel 7 descriptor address [63:32] |
| C      | IGH        |      |       |                                      |

**Write Behavior:** - Descriptor address is 64-bit, split across LOW and HIGH registers  
 - Write to HIGH register triggers descriptor engine kick-off - LOW register write is buffered, HIGH register write initiates transfer - Both registers must be written in order (LOW then HIGH) - Write blocks until descriptor engine accepts (back-pressure) - Read not supported (returns error)

### Example:

```
// Start DMA transfer on channel 0
// Descriptor at physical address 0x0000_0001_8000_0000 (64-bit)

// Write lower 32 bits first (buffered)
write32(BASE + CH0_CTRL_LOW, 0x8000_0000);

// Write upper 32 bits second (triggers kick-off)
write32(BASE + CH0_CTRL_HIGH, 0x0000_0001); // Blocks until accepted

// For descriptors in lower 4GB (typical case):
write32(BASE + CH0_CTRL_LOW, 0x8000_0000);
write32(BASE + CH0_CTRL_HIGH, 0x0000_0000); // Upper 32 bits = 0
```

---

## Global Control and Status (0x100 - 0x11F)

### GLOBAL\_CTRL (0x100)

Master control register for entire STREAM engine.

| Bits | Field          | Type | Reset | Description                           |
|------|----------------|------|-------|---------------------------------------|
| 31:2 | Reserved       | RO   | 0x0   | Reserved                              |
| 1    | GLOBAL_RS<br>T | RW   | 0     | Global reset (self-clearing)          |
| 0    | GLOBAL_E<br>N  | RW   | 0     | Global enable (1=enabled, 0=disabled) |

#### Usage:

```
// Enable STREAM engine
write32(BASE + GLOBAL_CTRL, 0x1);

// Reset all channels
write32(BASE + GLOBAL_CTRL, 0x3); // Set both EN and RST
// RST self-clears after one cycle
```

### GLOBAL\_STATUS (0x104)

Overall system status.

| Bits | Field       | Type | Description                     |
|------|-------------|------|---------------------------------|
| 31:1 | Reserved    | RO   | Reserved                        |
| 0    | SYSTEM_IDLE | RO   | System idle (all channels idle) |

### VERSION (0x108)

Version and configuration information (read-only).

| Bits  | Field        | Type | Value | Description                       |
|-------|--------------|------|-------|-----------------------------------|
| 31:24 | Reserved     | RO   | 0x00  | Reserved                          |
| 23:16 | NUM_CHANNELS | RO   | 0x08  | Number of channels (8)            |
| 15:8  | MAJOR        | RO   | 0x00  | Major version (0)                 |
| 7:0   | MINOR        | RO   | 0x5A  | Minor version (90 decimal = 0.90) |

## Per-Channel Control and Status (0x120 - 0x17F)

### CHANNEL\_ENABLE (0x120)

Per-channel enable control (bit vector).

| Bits | Field    | Type | Reset | Description                      |
|------|----------|------|-------|----------------------------------|
| 31:8 | Reserved | RO   | 0x0   | Reserved                         |
| 7:0  | CH_EN    | RW   | 0x00  | Channel enable [7:0] (1=enabled) |

#### Usage:

```
// Enable channels 0, 1, 2
write32(BASE + CHANNEL_ENABLE, 0x07);

// Disable channel 1, keep others
uint32_t val = read32(BASE + CHANNEL_ENABLE);
val &= ~(1 << 1); // Clear bit 1
write32(BASE + CHANNEL_ENABLE, val);
```

### CHANNEL\_RESET (0x124)

Per-channel reset control (bit vector, self-clearing).

| Bits | Field    | Type | Reset | Description                            |
|------|----------|------|-------|----------------------------------------|
| 31:8 | Reserved | RO   | 0x0   | Reserved                               |
| 7:0  | CH_RST   | RW   | 0x00  | Channel reset [7:0] (write 1 to reset) |

#### Usage:

```
// Reset channels 0 and 3
write32(BASE + CHANNEL_RESET, 0x09); // Bits 0 and 3
// Self-clears after reset completes
```

### CHANNEL\_IDLE (0x140)

Per-channel idle status (bit vector, read-only).

| Bits | Field    | Type | Description                           |
|------|----------|------|---------------------------------------|
| 31:8 | Reserved | RO   | Reserved                              |
| 7:0  | CH_IDLE  | RO   | Channel idle [7:0] (1=idle, 0=active) |

### *DESC\_ENGINE\_IDLE (0x144)*

Per-channel descriptor engine idle status.

| Bits | Field     | Type | Description                                     |
|------|-----------|------|-------------------------------------------------|
| 31:8 | Reserved  | RO   | Reserved                                        |
| 7:0  | DESC_IDLE | RO   | Descriptor engine idle [7:0] (1=idle, 0=active) |

### *SCHEDULER\_IDLE (0x148)*

Per-channel scheduler idle status.

| Bits | Field     | Type | Description                             |
|------|-----------|------|-----------------------------------------|
| 31:8 | Reserved  | RO   | Reserved                                |
| 7:0  | SCHED_IDL | RO   | Scheduler idle [7:0] (1=idle, 0=active) |

### *CH\_STATE[0..7] (0x150 - 0x16C)*

Per-channel scheduler FSM state (8 registers, stride 0x4).

| Offset | Register | Bits | Field     | Type | Description                         |
|--------|----------|------|-----------|------|-------------------------------------|
| 0x150  | CH0_STAT | 31:7 | Reserve d | RO   | Reserved                            |
|        |          | 6:0  | STATE     | RO   | Channel 0 scheduler state (one-hot) |
| 0x154  | CH1_STAT | 6:0  | STATE     | RO   | Channel 1 scheduler state (one-hot) |
| 0x158  | CH2_STAT | 6:0  | STATE     | RO   | Channel 2 scheduler state (one-hot) |
| 0x15C  | CH3_STAT | 6:0  | STATE     | RO   | Channel 3 scheduler state (one-hot) |
| 0x160  | CH4_STAT | 6:0  | STATE     | RO   | Channel 4 scheduler state (one-hot) |
| 0x164  | CH5_STAT | 6:0  | STATE     | RO   | Channel 5 scheduler state (one-hot) |

| Offset | Register   | Bits | Field | Type | Description                         |
|--------|------------|------|-------|------|-------------------------------------|
| 0x168  | CH6_STAT_E | 6:0  | STATE | RO   | Channel 6 scheduler state (one-hot) |
| 0x16C  | CH7_STAT_E | 6:0  | STATE | RO   | Channel 7 scheduler state (one-hot) |

### State Encoding (One-Hot):

|                              |                                        |
|------------------------------|----------------------------------------|
| Bit 0 (0x01) = CH_IDLE       | - Channel idle, waiting for descriptor |
| Bit 1 (0x02) = CH_FETCH_DESC | - Fetching descriptor from memory      |
| Bit 2 (0x04) = CH_XFER_DATA  | - Concurrent read AND write transfer   |
| Bit 3 (0x08) = CH_COMPLETE   | - Transfer complete                    |
| Bit 4 (0x10) = CH_NEXT_DESC  | - Fetching next chained descriptor     |
| Bit 5 (0x20) = CH_ERROR      | - Error state                          |
| Bit 6 (0x40) = CH_RESERVED   | - Reserved for future use              |

**Note:** Only ONE bit should be set at a time (one-hot encoding). Multiple bits set indicates a logic error.

---

### Engine Completion and Error Status (0x170 - 0x17F)

#### *SCHED\_ERROR (0x170)*

Per-channel scheduler error flags.

| Bits | Field      | Type | Description                                      |
|------|------------|------|--------------------------------------------------|
| 31:8 | Reserved   | RO   | Reserved                                         |
| 7:0  | SCHED_ER_R | RO   | Scheduler error bits [7:0] (1=error, 0=no error) |

#### *AXI\_RD\_COMPLETE (0x174)*

Per-channel AXI read engine completion status.

| Bits | Field        | Type | Description                                        |
|------|--------------|------|----------------------------------------------------|
| 31:8 | Reserved     | RO   | Reserved                                           |
| 7:0  | RD_COMPL_ETE | RO   | Read completion bits [7:0] (1=complete, 0=pending) |

### *AXI\_WR\_COMPLETE (0x178)*

Per-channel AXI write engine completion status.

| Bits | Field           | Type | Description                                         |
|------|-----------------|------|-----------------------------------------------------|
| 31:8 | Reserved        | RO   | Reserved                                            |
| 7:0  | WR_COMP<br>LETE | RO   | Write completion bits [7:0] (1=complete, 0=pending) |

### *Monitor FIFO Status (0x180 - 0x1FF)*

These registers are active when USE\_AXI\_MONITORS=1.

#### *MON\_FIFO\_STATUS (0x180)*

Monitor bus FIFO status indicators.

| Bits | Field              | Type | Description                            |
|------|--------------------|------|----------------------------------------|
| 31:4 | Reserved           | RO   | Reserved                               |
| 3    | MON_FIFO_UNF<br>L  | RO   | FIFO underflow detected (1=error)      |
| 2    | MON_FIFO_OVF<br>L  | RO   | FIFO overflow detected (1=error)       |
| 1    | MON_FIFO_EMP<br>TY | RO   | FIFO empty (1=empty, 0=data available) |
| 0    | MON_FIFO_FULL      | RO   | FIFO full (1=full, 0=space available)  |

#### *MON\_FIFO\_COUNT (0x184)*

Monitor bus FIFO entry count.

| Bits  | Field      | Type | Description                      |
|-------|------------|------|----------------------------------|
| 31:16 | Reserved   | RO   | Reserved                         |
| 15:0  | FIFO_COUNT | RO   | Number of entries in FIFO [15:0] |

## Scheduler Configuration (0x200 - 0x21F)

### *SCED\_TIMEOUT\_CYCLES (0x200)*

Timeout threshold for scheduler (global for all channels).

| Bits  | Field              | Type | Reset | Description             |
|-------|--------------------|------|-------|-------------------------|
| 31:16 | Reserved           | RO   | 0x0   | Reserved                |
| 15:0  | TIMEOUT_CYCL<br>ES | RW   | 1000  | Timeout in clock cycles |

### *SCED\_CONFIG (0x204)*

Scheduler feature enables (global for all channels).

| Bits | Field          | Type | Reset | Description                   |
|------|----------------|------|-------|-------------------------------|
| 31:5 | Reserved       | RO   | 0x0   | Reserved                      |
| 4    | PERF_EN        | RW   | 0     | Performance monitoring enable |
| 3    | COMPL_EN       | RW   | 1     | Completion reporting enable   |
| 2    | ERR_EN         | RW   | 1     | Error reporting enable        |
| 1    | TIMEOUT_E<br>N | RW   | 1     | Timeout detection enable      |
| 0    | SCHED_EN       | RW   | 1     | Scheduler enable              |

## Descriptor Engine Configuration (0x220 - 0x23F)

### *DESCENG\_CONFIG (0x220)*

Descriptor engine feature enables (global for all channels).

| Bits | Field           | Type | Reset | Description                      |
|------|-----------------|------|-------|----------------------------------|
| 31:6 | Reserved        | RO   | 0x0   | Reserved                         |
| 5:2  | FIFO_THRES<br>H | RW   | 0x8   | Prefetch FIFO threshold (4 bits) |
| 1    | PREFETCH_E<br>N | RW   | 0     | Prefetch enable                  |

| Bits | Field      | Typ<br>e | Reset | Description              |
|------|------------|----------|-------|--------------------------|
| 0    | DESCENG_EN | RW       | 1     | Descriptor engine enable |

#### *DESCENG\_ADDR0\_BASE (0x224)*

Base address for descriptor address range 0 (lower 32 bits).

| Bits | Field          | Typ<br>e | Reset           | Description          |
|------|----------------|----------|-----------------|----------------------|
| 31:0 | ADDR0_BA<br>SE | RW       | 0x00000000<br>0 | Address range 0 base |

#### *DESCENG\_ADDR0\_LIMIT (0x228)*

Limit address for descriptor address range 0 (lower 32 bits).

| Bits | Field           | Typ<br>e | Reset           | Description           |
|------|-----------------|----------|-----------------|-----------------------|
| 31:0 | ADDR0_LIM<br>IT | RW       | 0xFFFFFFF<br>FF | Address range 0 limit |

#### *DESCENG\_ADDR1\_BASE (0x22C)*

Base address for descriptor address range 1 (lower 32 bits).

| Bits | Field          | Typ<br>e | Reset           | Description          |
|------|----------------|----------|-----------------|----------------------|
| 31:0 | ADDR1_BA<br>SE | RW       | 0x00000000<br>0 | Address range 1 base |

#### *DESCENG\_ADDR1\_LIMIT (0x230)*

Limit address for descriptor address range 1 (lower 32 bits).

| Bits | Field           | Typ<br>e | Reset           | Description           |
|------|-----------------|----------|-----------------|-----------------------|
| 31:0 | ADDR1_LIM<br>IT | RW       | 0xFFFFFFF<br>FF | Address range 1 limit |

## Descriptor AXI Monitor Configuration (0x240 - 0x25F)

### DAXMON\_ENABLE (0x240)

Descriptor AXI master monitor enable controls.

| Bits | Field          | Type | Reset | Description                          |
|------|----------------|------|-------|--------------------------------------|
| 31:5 | Reserved       | RO   | 0x0   | Reserved                             |
| 4    | PERF_EN        | RW   | 0     | Performance packet enable            |
| 3    | TIMEOUT_E<br>N | RW   | 1     | Timeout detection enable             |
| 2    | COMPL_EN       | RW   | 0     | Completion packet enable             |
| 1    | ERR_EN         | RW   | 1     | Error detection enable               |
| 0    | MON_EN         | RW   | 1     | Master enable for descriptor monitor |

### DAXMON\_TIMEOUT (0x244)

Descriptor AXI monitor timeout threshold.

| Bits | Field          | Type | Reset | Description             |
|------|----------------|------|-------|-------------------------|
| 31:0 | TIMEOUT_CYCLES | RW   | 10000 | Timeout in clock cycles |

### DAXMON\_LATENCY\_THRESH (0x248)

Descriptor AXI monitor latency threshold.

| Bits | Field          | Type | Reset | Description                       |
|------|----------------|------|-------|-----------------------------------|
| 31:0 | LATENCY_THRESH | RW   | 5000  | Latency threshold in clock cycles |

### DAXMON\_PKT\_MASK (0x24C)

Descriptor AXI monitor packet type filtering.

| Bits  | Field    | Type | Reset | Description |
|-------|----------|------|-------|-------------|
| 31:16 | Reserved | RO   | 0x0   | Reserved    |

| Bits | Field     | Type | Reset    | Description                            |
|------|-----------|------|----------|----------------------------------------|
| 15:0 | PKT_MAS_K | RW   | 0xFFFF F | Packet type mask (1=enable, 0=disable) |

#### [DAXMON\\_ERR\\_CFG \(0x250\)](#)

Descriptor AXI monitor error selection and filtering.

| Bits  | Field       | Type | Reset | Description               |
|-------|-------------|------|-------|---------------------------|
| 31:16 | Reserved    | RO   | 0x0   | Reserved                  |
| 15:8  | ERR_MASK    | RW   | 0xFF  | Error type filtering mask |
| 7:4   | Reserved    | RO   | 0x0   | Reserved                  |
| 3:0   | ERR_SELE_CT | RW   | 0x0   | Error type selection      |

#### [DAXMON\\_MASK1 \(0x254\)](#)

Descriptor AXI monitor timeout and completion masks.

| Bits  | Field       | Type | Reset | Description     |
|-------|-------------|------|-------|-----------------|
| 31:16 | Reserved    | RO   | 0x0   | Reserved        |
| 15:8  | COMPL_MASK  | RW   | 0x00  | Completion mask |
| 7:0   | TIMEOUT_MSK | RW   | 0xFF  | Timeout mask    |

#### [DAXMON\\_MASK2 \(0x258\)](#)

Descriptor AXI monitor threshold and performance masks.

| Bits  | Field      | Type | Reset | Description      |
|-------|------------|------|-------|------------------|
| 31:16 | Reserved   | RO   | 0x0   | Reserved         |
| 15:8  | PERF_MAS_K | RW   | 0x00  | Performance mask |
| 7:0   | THRESH_MSK | RW   | 0xFF  | Threshold mask   |

### *DAXMON\_MASK3 (0x25C)*

Descriptor AXI monitor address and debug masks.

| Bits  | Field          | Type | Reset | Description     |
|-------|----------------|------|-------|-----------------|
| 31:16 | Reserved       | RO   | 0x0   | Reserved        |
| 15:8  | DEBUG_MA<br>SK | RW   | 0x00  | Debug<br>mask   |
| 7:0   | ADDR_MAS<br>K  | RW   | 0xFF  | Address<br>mask |

### **Read Engine AXI Monitor Configuration (0x260 - 0x27F)**

#### *RDMON\_ENABLE (0x260)*

Read engine AXI master monitor enable controls.

| Bits | Field          | Type | Reset | Description                    |
|------|----------------|------|-------|--------------------------------|
| 31:5 | Reserved       | RO   | 0x0   | Reserved                       |
| 4    | PERF_EN        | RW   | 0     | Performance packet enable      |
| 3    | TIMEOUT_E<br>N | RW   | 1     | Timeout detection enable       |
| 2    | COMPL_EN       | RW   | 0     | Completion packet enable       |
| 1    | ERR_EN         | RW   | 1     | Error detection enable         |
| 0    | MON_EN         | RW   | 1     | Master enable for read monitor |

#### *RDMON\_TIMEOUT (0x264)*

Read engine AXI monitor timeout threshold.

| Bits | Field              | Type | Reset | Description             |
|------|--------------------|------|-------|-------------------------|
| 31:0 | TIMEOUT_CYCL<br>ES | RW   | 10000 | Timeout in clock cycles |

#### *RDMON\_LATENCY\_THRESH (0x268)*

Read engine AXI monitor latency threshold.

| Bits | Field              | Type | Reset | Description                       |
|------|--------------------|------|-------|-----------------------------------|
| 31:0 | LATENCY_THR<br>ESH | RW   | 5000  | Latency threshold in clock cycles |

#### *RDMON\_PKT\_MASK (0x26C)*

Read engine AXI monitor packet type filtering.

| Bits  | Field        | Type | Reset    | Description                            |
|-------|--------------|------|----------|----------------------------------------|
| 31:16 | Reserved     | RO   | 0x0      | Reserved                               |
| 15:0  | PKT_MAS<br>K | RW   | 0xFFFF F | Packet type mask (1=enable, 0=disable) |

#### *RDMON\_ERR\_CFG (0x270)*

Read engine AXI monitor error selection and filtering.

| Bits  | Field          | Type | Reset | Description               |
|-------|----------------|------|-------|---------------------------|
| 31:16 | Reserved       | RO   | 0x0   | Reserved                  |
| 15:8  | ERR_MASK       | RW   | 0xFF  | Error type filtering mask |
| 7:4   | Reserved       | RO   | 0x0   | Reserved                  |
| 3:0   | ERR_SELE<br>CT | RW   | 0x0   | Error type selection      |

#### *RDMON\_MASK1 (0x274)*

Read engine AXI monitor timeout and completion masks.

| Bits  | Field            | Type | Reset | Description     |
|-------|------------------|------|-------|-----------------|
| 31:16 | Reserved         | RO   | 0x0   | Reserved        |
| 15:8  | COMPL_MASK       | RW   | 0x00  | Completion mask |
| 7:0   | TIMEOUT_MA<br>SK | RW   | 0xFF  | Timeout mask    |

#### *RDMON\_MASK2 (0x278)*

Read engine AXI monitor threshold and performance masks.

| Bits  | Field           | Type | Reset | Description      |
|-------|-----------------|------|-------|------------------|
| 31:16 | Reserved        | RO   | 0x0   | Reserved         |
| 15:8  | PERF_MAS<br>K   | RW   | 0x00  | Performance mask |
| 7:0   | THRESH_M<br>ASK | RW   | 0xFF  | Threshold mask   |

### [RDMON\\_MASK3 \(0x27C\)](#)

Read engine AXI monitor address and debug masks.

| Bits  | Field          | Type | Reset | Description  |
|-------|----------------|------|-------|--------------|
| 31:16 | Reserved       | RO   | 0x0   | Reserved     |
| 15:8  | DEBUG_MA<br>SK | RW   | 0x00  | Debug mask   |
| 7:0   | ADDR_MAS<br>K  | RW   | 0xFF  | Address mask |

### [Write Engine AXI Monitor Configuration \(0x280 - 0x29F\)](#)

#### [WRMON\\_ENABLE \(0x280\)](#)

Write engine AXI master monitor enable controls.

| Bits | Field          | Type | Reset | Description                     |
|------|----------------|------|-------|---------------------------------|
| 31:5 | Reserved       | RO   | 0x0   | Reserved                        |
| 4    | PERF_EN        | RW   | 0     | Performance packet enable       |
| 3    | TIMEOUT_E<br>N | RW   | 1     | Timeout detection enable        |
| 2    | COMPL_EN       | RW   | 0     | Completion packet enable        |
| 1    | ERR_EN         | RW   | 1     | Error detection enable          |
| 0    | MON_EN         | RW   | 1     | Master enable for write monitor |

#### [WRMON\\_TIMEOUT \(0x284\)](#)

Write engine AXI monitor timeout threshold.

| Bits | Field              | Typ<br>e | Reset | Description             |
|------|--------------------|----------|-------|-------------------------|
| 31:0 | TIMEOUT_CYCL<br>ES | RW       | 10000 | Timeout in clock cycles |

#### [\*WRMON\\_LATENCY\\_THRESH \(0x288\)\*](#)

Write engine AXI monitor latency threshold.

| Bits | Field              | Typ<br>e | Reset | Description                       |
|------|--------------------|----------|-------|-----------------------------------|
| 31:0 | LATENCY_THR<br>ESH | RW       | 5000  | Latency threshold in clock cycles |

#### [\*WRMON\\_PKT\\_MASK \(0x28C\)\*](#)

Write engine AXI monitor packet type filtering.

| Bits  | Field        | Typ<br>e | Reset       | Description                            |
|-------|--------------|----------|-------------|----------------------------------------|
| 31:16 | Reserved     | RO       | 0x0         | Reserved                               |
| 15:0  | PKT_MAS<br>K | RW       | 0xFFFF<br>F | Packet type mask (1=enable, 0=disable) |

#### [\*WRMON\\_ERR\\_CFG \(0x290\)\*](#)

Write engine AXI monitor error selection and filtering.

| Bits  | Field          | Typ<br>e | Reset | Description               |
|-------|----------------|----------|-------|---------------------------|
| 31:16 | Reserved       | RO       | 0x0   | Reserved                  |
| 15:8  | ERR_MASK       | RW       | 0xFF  | Error type filtering mask |
| 7:4   | Reserved       | RO       | 0x0   | Reserved                  |
| 3:0   | ERR_SELE<br>CT | RW       | 0x0   | Error type selection      |

#### [\*WRMON\\_MASK1 \(0x294\)\*](#)

Write engine AXI monitor timeout and completion masks.

| Bits  | Field            | Type | Reset | Description     |
|-------|------------------|------|-------|-----------------|
| 31:16 | Reserved         | RO   | 0x0   | Reserved        |
| 15:8  | COMPL_MASK       | RW   | 0x00  | Completion mask |
| 7:0   | TIMEOUT_MA<br>SK | RW   | 0xFF  | Timeout mask    |

#### [WRMON\\_MASK2 \(0x298\)](#)

Write engine AXI monitor threshold and performance masks.

| Bits  | Field           | Type | Reset | Description      |
|-------|-----------------|------|-------|------------------|
| 31:16 | Reserved        | RO   | 0x0   | Reserved         |
| 15:8  | PERF_MAS<br>K   | RW   | 0x00  | Performance mask |
| 7:0   | THRESH_M<br>ASK | RW   | 0xFF  | Threshold mask   |

#### [WRMON\\_MASK3 \(0x29C\)](#)

Write engine AXI monitor address and debug masks.

| Bits  | Field          | Type | Reset | Description  |
|-------|----------------|------|-------|--------------|
| 31:16 | Reserved       | RO   | 0x0   | Reserved     |
| 15:8  | DEBUG_MA<br>SK | RW   | 0x00  | Debug mask   |
| 7:0   | ADDR_MAS<br>K  | RW   | 0xFF  | Address mask |

### [AXI Transfer Configuration \(0x2A0 - 0x2AF\)](#)

#### [AXI\\_XFER\\_CONFIG \(0x2A0\)](#)

AXI read and write transfer burst sizes.

| Bits  | Field             | Typ | Rese | Description                                     |
|-------|-------------------|-----|------|-------------------------------------------------|
|       |                   | e   | t    |                                                 |
| 31:16 | Reserved          | RO  | 0x0  | Reserved                                        |
| 15:8  | WR_XFER_B<br>EATS | RW  | 15   | AXI write transfer beats (AWLEN: 0-255 = 1-256) |

| Bits | Field             | Typ | Rese |                                                |
|------|-------------------|-----|------|------------------------------------------------|
|      |                   | e   | t    | Description                                    |
| 7:0  | RD_XFER_BE<br>ATS | RW  | 15   | AXI read transfer beats (ARLEN: 0-255 = 1-256) |

**Usage:**

```
// Configure for 16-beat bursts (default)
write32(BASE + AXI_XFER_CONFIG, 0x0F0F);
```

```
// Configure for 64-beat bursts
write32(BASE + AXI_XFER_CONFIG, 0x3F3F);
```

```
// Configure for maximum 256-beat bursts
write32(BASE + AXI_XFER_CONFIG, 0xFFFF);
```

---

## Performance Profiler Configuration (0x2B0 - 0x2BF)

### PERF\_CONFIG (0x2B0)

Performance profiler enable and mode controls.

| Bits | Field          | Typ |       |                                         |
|------|----------------|-----|-------|-----------------------------------------|
|      |                | e   | Reset | Description                             |
| 31:3 | Reserved       | RO  | 0x0   | Reserved                                |
| 2    | PERF_CLEA<br>R | RW  | 0     | Clear counters (write 1, self-clearing) |
| 1    | PERF_MOD<br>E  | RW  | 0     | Mode: 0=count, 1=histogram              |
| 0    | PERF_EN        | RW  | 0     | Performance profiler enable             |

**Usage:**

```
// Enable performance profiler in count mode
write32(BASE + PERF_CONFIG, 0x01);
```

```
// Enable in histogram mode
write32(BASE + PERF_CONFIG, 0x03);
```

```
// Clear counters
write32(BASE + PERF_CONFIG, 0x05); // EN + CLEAR
// PERF_CLEAR self-clears after one cycle
```

---

## Typical Usage Flow

### Initialization

```
// 1. Global enable
write32(BASE + GLOBAL_CTRL, 0x1);

// 2. Configure scheduler
write32(BASE + SCHED_TIMEOUT_CYCLES, 10000);
write32(BASE + SCHED_CONFIG, 0x1F); // All features enabled

// 3. Configure descriptor engine
write32(BASE + DESCENG_CONFIG, 0x01); // Enable, no prefetch
write32(BASE + DESCENG_ADDR0_BASE, 0x8000_0000);
write32(BASE + DESCENG_ADDR0_LIMIT, 0x8FFF_FFFF);

// 4. Configure monitors (minimal reporting)
write32(BASE + DAXMON_CONFIG, 0x05); // Error + timeout only
write32(BASE + RDMON_CONFIG, 0x05);
write32(BASE + WRMON_CONFIG, 0x05);

// 5. Enable desired channels
write32(BASE + CHANNEL_ENABLE, 0xFF); // All 8 channels
```

### Start Transfer

```
// Write 64-bit descriptor address to channel kick-off registers
// Descriptor at address 0x0000_0000_8000_0100 (64-bit)
write32(BASE + CH0_CTRL_LOW, 0x8000_0100); // Lower 32 bits
(buffered)
write32(BASE + CH0_CTRL_HIGH, 0x0000_0000); // Upper 32 bits
(triggers kick-off)
// Transfer starts immediately (blocks until descriptor engine ready)
```

### Poll for Completion

```
// Check channel 0 idle status
while (!(read32(BASE + CHANNEL_IDLE) & 0x01)) {
    // Wait for channel 0 to become idle
}

// Or check scheduler state (one-hot encoding)
while ((read32(BASE + CH0_STATE) & 0x7F) != 0x01) {
    // Wait for CH_IDLE state (bit 0 = 0x01)
}
```

### Error Handling

```
// Check all channel states for errors (one-hot encoding)
for (int ch = 0; ch < 8; ch++) {
```

```

    uint32_t state = read32(BASE + CH0_STATE + (ch * 4)) & 0x7F;
    if (state & 0x20) { // CH_ERROR (bit 5)
        // Reset channel
        write32(BASE + CHANNEL_RESET, 1 << ch);
    }
}

```

---

## Register Summary Table

| Offset Range | Description                            | Count | Type          |
|--------------|----------------------------------------|-------|---------------|
| 0x000-0x03F  | Channel kick-off registers (LOW/HIGH)  | 16    | Write-routing |
| 0x100-0x11F  | Global control and status              | 3     | RW/RO         |
| 0x120-0x13F  | Per-channel control                    | 2     | RW            |
| 0x140-0x16F  | Per-channel status                     | 11    | RO            |
| 0x170-0x17F  | Engine completion and error status     | 3     | RO            |
| 0x180-0x1FF  | Monitor FIFO status                    | 2     | RO            |
| 0x200-0x21F  | Scheduler configuration                | 2     | RW            |
| 0x220-0x23F  | Descriptor engine configuration        | 5     | RW            |
| 0x240-0x25F  | Descriptor AXI monitor configuration   | 8     | RW            |
| 0x260-0x27F  | Read engine AXI monitor configuration  | 8     | RW            |
| 0x280-0x29F  | Write engine AXI monitor configuration | 8     | RW            |
| 0x2A0-0x2AF  | AXI transfer configuration             | 1     | RW            |
| 0x2B0-0x2BF  | Performance profiler configuration     | 1     | RW            |

**Total:** 70 registers (16 kick-off + 54 config/status)

---

## PeakRDL Generation

To generate SystemVerilog from the register definition:

```
cd projects/components/stream/rtl/stream_macro/
peakrdl regblock stream_regs.rdl -o generated/
```

This generates: - stream\_regs\_pkg.sv - Register definitions package - stream\_regs.sv - APB slave register interface

---

## Revision History:

| Version | Date       | Author        | Description                                                                                                                                                                                                      |
|---------|------------|---------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 1.0     | 2025-10-20 | sean galloway | Initial creation                                                                                                                                                                                                 |
| 1.1     | 2025-12-01 | sean galloway | Added complete monitor registers from RDL<br>Added engine completion/error status (0x170)<br>Added monitor FIFO status (0x180)<br>Added AXI transfer config (0x2A0)<br>Added performance profiler config (0x2B0) |

## Chapter 5: Programming Models

This chapter provides software developer guidance for using the STREAM DMA engine.

### Contents

#### [01\\_initialization.md](#)

- Power-on initialization sequence
- Register configuration
- Channel setup
- Configuration presets (minimal, high-performance)

#### [02\\_single\\_transfer.md](#)

- Descriptor format (256-bit)
- Kick-off address map and write sequence
- Simple single-descriptor transfer
- C code examples with complete workflow

#### [03\\_chained\\_transfers.md](#)

- Multi-descriptor chains
- Descriptor linking via next\_descriptor\_ptr
- Chain termination methods
- Scatter-gather operations

- Prefetch mode for performance

## [04\\_multi\\_channel.md](#)

- 8-channel concurrent operations
- Priority-based scheduling
- Resource sharing strategies
- Channel pooling and load balancing
- Performance considerations

## [05\\_error\\_handling.md](#)

- Error types (AXI, timeout, internal)
- Error detection registers
- Recovery procedures
- Interrupt-based error handling
- Debug monitoring

## **Planned (Future)**

### [06\\_performance\\_tuning.md](#)

- Burst size selection
- Priority tuning
- SRAM depth considerations
- Maximizing throughput

### [07\\_software\\_examples.md](#)

- Complete working examples
  - Linux driver skeleton
  - Bare-metal usage
  - Common use cases
- 

**Status:** Core programming guides complete (01-05)

**Target Audience:** - Software engineers integrating STREAM - Driver developers - System architects - Application developers

---

**Last Updated:** 2025-12-01

# Chapter 6: Configuration Reference

**Version:** 0.90 **Last Updated:** 2025-11-22 **Purpose:** Complete reference for all STREAM configuration signals

---

## Overview

STREAM provides comprehensive runtime configuration through APB registers and compile-time parameters. This chapter documents all configuration signals, their valid ranges, default values, and recommended settings for different use cases.

### Configuration Categories

| Category                 | Signals                         | Purpose                          |
|--------------------------|---------------------------------|----------------------------------|
| <b>Channel Control</b>   | 2                               | Enable/reset individual channels |
| <b>Scheduler</b>         | 6                               | Transfer scheduling and timeouts |
| <b>Descriptor Engine</b> | 7                               | Descriptor fetch behavior        |
| <b>AXI Monitors</b>      | 45 (15 per monitor × 3)         | Debug/trace filtering            |
| <b>AXI Transfer</b>      | 2                               | Burst configuration              |
| <b>Performance</b>       | 3                               | Profiling and metrics            |
| <b>Total</b>             | <b>65 configuration signals</b> | Full system control              |

## 1. Channel Control Configuration

### `cfg_channel_enable[NUM_CHANNELS-1:0]`

**Type:** Per-channel enable **Width:** 1 bit × NUM\_CHANNELS (default 8) **Default:** 8'hFF (all enabled) **Register:** CHANNEL\_ENABLE @ 0x120

**Description:** Enables or disables individual DMA channels. When disabled, the channel:  
- Ignores descriptor kick-off requests  
- Completes current transfer if in progress  
- Enters idle state  
- Remains in idle until re-enabled

**Valid Values:** - 1'b1: Channel enabled (can accept transfers) - 1'b0: Channel disabled (ignores new transfers)

**Use Cases:**

```
// Enable only channels 0, 2, 4 (even channels)
cfg_channel_enable = 8'b01010101;
```

```
// Disable all channels (emergency stop)
cfg_channel_enable = 8'b00000000;
```

```
// Enable all channels (normal operation)
cfg_channel_enable = 8'b11111111;
```

**Interaction with Global Enable:** The final channel enable is:  
cfg\_channel\_enable[i] & reg\_global\_ctrl\_global\_en

---

## cfg\_channel\_reset[NUM\_CHANNELS-1:0]

**Type:** Per-channel reset **Width:** 1 bit × NUM\_CHANNELS (default 8) **Default:** 8'h00 (no resets active) **Register:** CHANNEL\_RESET @ 0x124

**Description:** Asserts reset for individual channels without affecting other channels or global state. When asserted: - Channel FSM returns to IDLE - Pending transfers aborted - SRAM buffers flushed - Descriptor engine reset

**Valid Values:** - 1'b1: Channel in reset (clears state) - 1'b0: Channel operating normally

**Use Cases:**

```
// Reset channel 3 after error
cfg_channel_reset = 8'b00001000;
wait_cycles(10);
cfg_channel_reset = 8'b00000000; // Release reset
```

```
// Reset all channels (soft reset)
cfg_channel_reset = 8'hFF;
wait_cycles(10);
cfg_channel_reset = 8'h00;
```

**IMPORTANT:** Assert reset for minimum 10 clock cycles to ensure complete FSM reset.

---

## 2. Scheduler Configuration

### cfg\_sched\_timeout\_cycles[15:0]

**Type:** Timeout threshold **Width:** 16 bits **Default:** 16'd1000 (1,000 cycles) **Register:** SCHED\_TIMEOUT\_CYCLES @ 0x200[15:0]

**Description:** Number of clock cycles before a channel operation times out.  
Applies to: - Descriptor fetch latency - AXI read/write response latency - Scheduler state transitions

**Valid Range:** 100 to 65535 cycles **Typical Values:** - Fast SRAM: 100-500 cycles - DDR3/DDR4: 1000-10000 cycles - High-latency: 10000-65535 cycles

#### Calculation:

$$\text{Timeout cycles} = (\text{Expected latency} \times 10) + \text{margin}$$

Example for DDR4 @ 200 MHz:

- Expected read latency: 100 cycles (500 ns)
  - Safety margin: 10x
  - Timeout =  $100 \times 10 = 1000$  cycles
- 

### cfg\_sched\_enable

**Type:** Global scheduler enable **Width:** 1 bit **Default:** 1'b1 (enabled) **Register:** SCHED\_CONFIG @ 0x204[0]

**Description:** Master enable for all schedulers. When disabled, all channels stop scheduling new operations.

---

### cfg\_sched\_timeout\_enable

**Type:** Timeout detection enable **Width:** 1 bit **Default:** 1'b1 (enabled) **Register:** SCHED\_CONFIG @ 0x204[1]

**Description:** Enables timeout detection for scheduler operations. Disable for simulation or known slow memory.

---

### **cfg\_sched\_err\_enable**

**Type:** Error detection enable **Width:** 1 bit **Default:** 1'b1 (enabled) **Register:** SCHED\_CONFIG @ 0x204[2]

**Description:** Enables error packet generation for scheduler errors (AXI SLVERR, DECERR, timeouts).

---

### **cfg\_sched\_compl\_enable**

**Type:** Completion event enable **Width:** 1 bit **Default:** 1'b1 (enabled by default) **Register:** SCHED\_CONFIG @ 0x204[3]

**Description:** Enables MonBus packets for transfer completion events. Generate high traffic, use sparingly.

---

### **cfg\_sched\_perf\_enable**

**Type:** Performance monitoring enable **Width:** 1 bit **Default:** 1'b0 (disabled by default) **Register:** SCHED\_CONFIG @ 0x204[4]

**Description:** Enables performance profiling packets (latency, throughput metrics).

---

## **3. Descriptor Engine Configuration**

### **cfg\_desceng\_enable**

**Type:** Descriptor engine global enable **Width:** 1 bit **Default:** 1'b1 (enabled) **Register:** DESCENG\_CTRL @ 0x220[0]

**Description:** Global enable for all descriptor engines. When disabled, descriptor fetch stops.

---

### **cfg\_desceng\_prefetch**

**Type:** Prefetch enable **Width:** 1 bit **Default:** 1'b0 (disabled) **Register:** DESCENG\_CTRL @ 0x220[1]

**Description:** Enables descriptor prefetching to hide fetch latency.

**Prefetch Behavior:** - Enabled: Fetch next descriptor while current transfer executes - Disabled: Wait for current transfer completion before fetching next

**Performance Impact:** - Prefetch ON: +15-30% throughput for chained descriptors - Prefetch OFF: Simpler logic, lower area

---

### [\*\*cfg\\_desceng\\_fifo\\_thresh\[3:0\]\*\*](#)

**Type:** FIFO threshold **Width:** 4 bits **Default:** 4'h8 (8 entries) **Register:** DESCENG\_CTRL @ 0x220[7:4]

**Description:** Number of entries in descriptor FIFO before asserting backpressure.

**Valid Range:** 1-15 entries **Typical Values:** - Low latency: 2-4 entries - Balanced: 8 entries (default) - High throughput: 12-15 entries

---

### [\*\*cfg\\_desceng\\_addr0\\_base\[31:0\]\*\*](#)

**Type:** Base address for descriptor region 0 **Width:** 32 bits **Default:** 32'h0000\_0000 **Register:** DESCENG\_ADDR0\_BASE @ 0x224

**Description:** Base address of first descriptor memory region. Descriptors fetched from this region if within range.

**Alignment:** Must be aligned to descriptor size (256 bits = 32 bytes)

---

### [\*\*cfg\\_desceng\\_addr0\\_limit\[31:0\]\*\*](#)

**Type:** Limit address for descriptor region 0 **Width:** 32 bits **Default:** 32'hFFFF\_FFFF (no limit) **Register:** DESCENG\_ADDR0\_LIMIT @ 0x228

**Description:** Upper limit of first descriptor region. Descriptors beyond this address use region 1.

---

### ***cfg\_desceng\_addr1\_base[31:0]***

**Type:** Base address for descriptor region 1 **Width:** 32 bits **Default:** 32'h0000\_0000  
**Register:** DESCENG\_ADDR1\_BASE @ 0x22C

**Description:** Base address of second descriptor memory region (optional).

---

### ***cfg\_desceng\_addr1\_limit[31:0]***

**Type:** Limit address for descriptor region 1 **Width:** 32 bits **Default:** 32'hFFFF\_FFFF **Register:** DESCENG\_ADDR1\_LIMIT @ 0x230

**Description:** Upper limit of second descriptor region.

---

## **4. AXI Monitor Configuration**

STREAM includes three independent AXI monitors with identical configuration sets:

1. **Descriptor AXI Monitor (cfg\_desc\_mon\_\*)** - Monitors descriptor fetch AXI master
2. **Read Engine Monitor (cfg\_rdeng\_mon\_\*)** - Monitors data read AXI master
3. **Write Engine Monitor (cfg\_wreng\_mon\_\*)** - Monitors data write AXI master

Each monitor has 15 configuration signals with the same structure.

---

### **4.1 Descriptor AXI Monitor (cfg\_desc\_mon\_\*)**

**Register Base:** 0x240-0x25F

#### ***cfg\_desc\_mon\_enable***

**Type:** Monitor master enable **Width:** 1 bit **Default:** 1'b1 (enabled) **Register:** DAXMON\_ENABLE @ 0x240[0]

**Description:** Master enable for descriptor AXI monitor. All monitor packets disabled when this is 0.

---

### *cfg\_desc\_mon\_err\_enable*

**Type:** Error packet enable **Width:** 1 bit **Default:** 1'b1 (enabled) **Register:** DAXMON\_ENABLE @ 0x240[1]

**Description:** Enables error packet generation (SLVERR, DECERR, protocol violations).

---

### *cfg\_desc\_mon\_perf\_enable*

**Type:** Performance packet enable **Width:** 1 bit **Default:** 1'b0 (disabled) **Register:** DAXMON\_ENABLE @ 0x240[2]

**Description:** Enables performance monitoring packets (latency, bandwidth).

**WARNING:** High packet rate - use only during debug.

---

### *cfg\_desc\_mon\_timeout\_enable*

**Type:** Timeout detection enable **Width:** 1 bit **Default:** 1'b1 (enabled) **Register:** DAXMON\_ENABLE @ 0x240[3]

**Description:** Enables timeout packet generation when transactions exceed threshold.

---

### *cfg\_desc\_mon\_timeout\_cycles[31:0]*

**Type:** Timeout threshold **Width:** 32 bits **Default:** 32'd10000 **Register:** DAXMON\_TIMEOUT @ 0x244

**Description:** Number of cycles before transaction times out.

---

### *cfg\_desc\_mon\_latency\_thresh[31:0]*

**Type:** Latency threshold **Width:** 32 bits **Default:** 32'd1000 **Register:** DAXMON\_LATENCY @ 0x248

**Description:** Latency threshold for performance warnings.

---

## *cfg\_desc\_mon\_pkt\_mask[15:0]*

**Type:** Packet type filter **Width:** 16 bits (1 bit per packet type) **Default:** 16'h00FF  
(errors + completions) **Register:** DAXMON\_PKT\_MASK @ 0x24C[15:0]

**Description:** Bit mask to filter packet types. Only packet types with corresponding bit set are generated.

### Packet Type Mapping:

| Bit  | Packet Type  |
|------|--------------|
| 0    | AR command   |
| 1    | AW command   |
| 2    | R completion |
| 3    | W data       |
| 4    | B response   |
| 5    | Error        |
| 6    | Timeout      |
| 7    | Completion   |
| 8-15 | Reserved     |

### Examples:

```
// Only errors
cfg_desc_mon_pkt_mask = 16'h0020; // Bit 5
```

```
// Errors + timeouts
cfg_desc_mon_pkt_mask = 16'h0060; // Bits 5-6
```

```
// All packets
cfg_desc_mon_pkt_mask = 16'hFFFF;
```

---

## *cfg\_desc\_mon\_err\_select[3:0]*

**Type:** Error type selector **Width:** 4 bits **Default:** 4'hF (all errors) **Register:** DAXMON\_ERR\_SELECT @ 0x24C[19:16]

**Description:** Selects which error types to monitor.

### Error Type Bits:

| Bit | Error Type |
|-----|------------|
| 0   | SLVERR     |
| 1   | DECERR     |

---

|   |                    |
|---|--------------------|
| 2 | Protocol violation |
| 3 | Reserved           |

---

#### *cfg\_desc\_mon\_err\_mask[7:0]*

**Type:** Error event filter **Width:** 8 bits **Default:** 8'hFF (all errors) **Register:** DAXMON\_MASK1 @ 0x250[7:0]

**Description:** Bit mask for error packet generation (channel-specific filtering).

---

#### *cfg\_desc\_mon\_timeout\_mask[7:0]*

**Type:** Timeout channel filter **Width:** 8 bits **Default:** 8'hFF (all channels) **Register:** DAXMON\_MASK1 @ 0x250[15:8]

**Description:** Channel mask for timeout packets. Set bit enables timeout detection for corresponding channel.

---

#### *cfg\_desc\_mon\_compl\_mask[7:0]*

**Type:** Completion channel filter **Width:** 8 bits **Default:** 8'h00 (no channels)  
**Register:** DAXMON\_MASK1 @ 0x250[23:16]

**Description:** Channel mask for completion packets.

**WARNING:** Completion packets are high volume - enable only for specific channels.

---

#### *cfg\_desc\_mon\_thresh\_mask[7:0]*

**Type:** Threshold event filter **Width:** 8 bits **Default:** 8'hFF (all channels) **Register:** DAXMON\_MASK2 @ 0x254[7:0]

**Description:** Channel mask for latency threshold exceedance packets.

---

#### *cfg\_desc\_mon\_perf\_mask[7:0]*

**Type:** Performance packet filter **Width:** 8 bits **Default:** 8'h00 (no channels)  
**Register:** DAXMON\_MASK2 @ 0x254[15:8]

**Description:** Channel mask for performance monitoring packets.

---

#### *cfg\_desc\_mon\_addr\_mask[7:0]*

**Type:** Address-based filter **Width:** 8 bits **Default:** 8'hFF (all addresses) **Register:** DAXMON\_MASK2 @ 0x254[23:16]

**Description:** Channel mask for address-range-based packet filtering.

---

#### *cfg\_desc\_mon\_debug\_mask[7:0]*

**Type:** Debug event filter **Width:** 8 bits **Default:** 8'h00 (no debug packets)  
**Register:** DAXMON\_MASK2 @ 0x254[31:24]

**Description:** Channel mask for debug-level packets (verbose trace).

---

### 4.2 Read Engine Monitor (cfg\_rdeng\_mon\_\*)

**Register Base:** 0x260-0x27F

All signals identical to Descriptor Monitor (Section 4.1), but applied to data read AXI master.

**Key Differences:** - Monitors data transfers (not descriptors) - Higher throughput  
→ more packets - Recommended: Keep cfg\_rdeng\_mon\_compl\_enable = 0 unless debugging

---

### 4.3 Write Engine Monitor (cfg\_wreng\_mon\_\*)

**Register Base:** 0x280-0x29F

All signals identical to Descriptor Monitor (Section 4.1), but applied to data write AXI master.

**Key Differences:** - Monitors write transactions (AW/W/B channels) - B response timing critical for performance - Recommended: Enable only error packets by default

---

## 5. AXI Transfer Configuration

### cfg\_axi\_rd\_xfer\_beats[7:0]

**Type:** Read transfer size **Width:** 8 bits **Default:** 8'd16 (16 beats) **Register:** AXI\_RD\_XFER @ 0x2A0[7:0]

**Description:** Default number of beats per AXI read burst. Actual burst size may be less to respect 4KB boundaries.

**Valid Range:** 1-256 beats (AXI4 standard) **Typical Values:** - Small transfers: 4-8 beats - Balanced: 16 beats (default) - Large transfers: 32-64 beats - Maximum throughput: 128-256 beats

#### Calculation:

Transfer size (bytes) = beats × (DATA\_WIDTH / 8)

Example for DATA\_WIDTH = 512 bits:

- 16 beats =  $16 \times 64 = 1024$  bytes (1 KB)
  - 64 beats =  $64 \times 64 = 4096$  bytes (4 KB)
- 

### cfg\_axi\_wr\_xfer\_beats[7:0]

**Type:** Write transfer size **Width:** 8 bits **Default:** 8'd16 (16 beats) **Register:** AXI\_WR\_XFER @ 0x2A0[15:8]

**Description:** Default number of beats per AXI write burst.

**Same constraints as cfg\_axi\_rd\_xfer\_beats**

---

## 6. Performance Profiler Configuration

### cfg\_perf\_enable

**Type:** Profiler enable **Width:** 1 bit **Default:** 1'b0 (disabled) **Register:** PERF\_CTRL @ 0x2B0[0]

**Description:** Enables performance profiling for all channels. When enabled, profiler captures:  
- Transfer start/end timestamps  
- Latency per channel  
- Throughput measurements  
- Channel utilization

---

### cfg\_perf\_mode

**Type:** Profiling mode **Width:** 1 bit **Default:** 1'b0 (timestamp mode) **Register:** PERF\_CTRL @ 0x2B0[1]

**Description:** Selects profiling mode:  
- 1'b0: Timestamp mode - Record absolute timestamps  
- 1'b1: Elapsed time mode - Record delta times

**Use Cases:** - Timestamp: Correlate events across multiple blocks  
- Elapsed: Measure operation latencies

---

### cfg\_perf\_clear

**Type:** Clear profiler state **Width:** 1 bit (write-only) **Default:** 1'b0 **Register:** PERF\_CTRL @ 0x2B0[2]

**Description:** Write 1'b1 to clear profiler FIFOs and counters. Self-clearing (automatically returns to 0).

---

## 7. Configuration Presets

### 7.1 Minimal Configuration (Tutorial/Embedded)

**Use Case:** Educational, minimal logic, single-channel operation

```
// Channel control
cfg_channel_enable = 8'b00000001; // Only channel 0

// Scheduler
cfg_sched_enable = 1'b1;
cfg_sched_timeout_cycles = 16'd500;           // Short timeout (SRAM)
cfg_sched_timeout_enable = 1'b1;
cfg_sched_err_enable = 1'b1;
cfg_sched_compl_enable = 1'b0;                // Disable completion packets
cfg_sched_perf_enable = 1'b0;                // Disable performance
packets
```

```

// Descriptor engine
cfg_desceng_enable = 1'b1;
cfg_desceng_prefetch = 1'b0;           // No prefetch (simpler)
cfg_desceng_fifo_thresh = 4'h4;        // Small FIFO

// All monitors DISABLED (reduce logic)
cfg_desc_mon_enable = 1'b0;
cfg_rdeng_mon_enable = 1'b0;
cfg_wreng_mon_enable = 1'b0;

// AXI transfer
cfg_axi_rd_xfer_beats = 8'd8;         // Small bursts
cfg_axi_wr_xfer_beats = 8'd8;

// Performance profiler
cfg_perf_enable = 1'b0;               // Disabled

```

---

## 7.2 Balanced Configuration (Typical FPGA)

**Use Case:** General-purpose DMA, moderate channels, balanced performance/area

```

// Channel control
cfg_channel_enable = 8'hFF;           // All 8 channels

// Scheduler
cfg_sched_enable = 1'b1;
cfg_sched_timeout_cycles = 16'd5000;   // DDR4 timeout
cfg_sched_timeout_enable = 1'b1;
cfg_sched_err_enable = 1'b1;
cfg_sched_compl_enable = 1'b0;         // Errors only
cfg_sched_perf_enable = 1'b0;

// Descriptor engine
cfg_desceng_enable = 1'b1;
cfg_desceng_prefetch = 1'b1;           // Enable prefetch
cfg_desceng_fifo_thresh = 4'h8;        // Balanced FIFO

// Descriptor monitor (errors only)
cfg_desc_mon_enable = 1'b1;
cfg_desc_mon_err_enable = 1'b1;
cfg_desc_mon_perf_enable = 1'b0;
cfg_desc_mon_timeout_enable = 1'b1;
cfg_desc_mon_timeout_cycles = 32'd10000;
cfg_desc_mon_pkt_mask = 16'h0060;      // Errors + timeouts

// Read/write monitors (errors only)

```

```

cfg_rdeng_mon_enable = 1'b1;
cfg_rdeng_mon_err_enable = 1'b1;
cfg_rdeng_mon_perf_enable = 1'b0;

cfg_wreng_mon_enable = 1'b1;
cfg_wreng_mon_err_enable = 1'b1;
cfg_wreng_mon_perf_enable = 1'b0;

// AXI transfer
cfg_axi_rd_xfer_beats = 8'd32;           // Moderate bursts
cfg_axi_wr_xfer_beats = 8'd32;

// Performance profiler
cfg_perf_enable = 1'b1;                  // Enable profiling
cfg_perf_mode = 1'b1;                   // Elapsed time mode

```

---

### 7.3 High-Performance Configuration (ASIC/Datacenter)

**Use Case:** Maximum throughput, all channels active, full monitoring

```

// Channel control
cfg_channel_enable = 8'hFF;             // All channels

// Scheduler
cfg_sched_enable = 1'b1;
cfg_sched_timeout_cycles = 16'd20000;    // High latency tolerance
cfg_sched_timeout_enable = 1'b1;
cfg_sched_err_enable = 1'b1;
cfg_sched_compl_enable = 1'b1;          // Full monitoring
cfg_sched_perf_enable = 1'b1;

// Descriptor engine
cfg_desceng_enable = 1'b1;
cfg_desceng_prefetch = 1'b1;
cfg_desceng_fifo_thresh = 4'hF;         // Max FIFO depth

// All monitors ENABLED with full profiling
cfg_desc_mon_enable = 1'b1;
cfg_desc_mon_err_enable = 1'b1;
cfg_desc_mon_perf_enable = 1'b1;        // Performance monitoring
cfg_desc_mon_timeout_enable = 1'b1;
cfg_desc_mon_timeout_cycles = 32'd20000;
cfg_desc_mon_pkt_mask = 16'hFFFF;       // All packet types

// Same for read/write monitors
cfg_rdeng_mon_enable = 1'b1;

```

```

cfg_rdeng_mon_err_enable = 1'b1;
cfg_rdeng_mon_perf_enable = 1'b1;

cfg_wreng_mon_enable = 1'b1;
cfg_wreng_mon_err_enable = 1'b1;
cfg_wreng_mon_perf_enable = 1'b1;

// AXI transfer
cfg_axi_rd_xfer_beats = 8'd128;           // Large bursts
cfg_axi_wr_xfer_beats = 8'd128;

// Performance profiler
cfg_perf_enable = 1'b1;
cfg_perf_mode = 1'b0;                      // Timestamp mode
(correlation)

```

---

## 7.4 Debug Configuration (Verbose Monitoring)

**Use Case:** Debugging integration issues, detailed trace analysis

```

// Enable specific channel for debug
cfg_channel_enable = 8'b00000001;          // Channel 0 only

// Scheduler
cfg_sched_enable = 1'b1;
cfg_sched_timeout_cycles = 16'd65535;      // Long timeout for debug
cfg_sched_timeout_enable = 1'b1;
cfg_sched_err_enable = 1'b1;
cfg_sched_compl_enable = 1'b1;             // All events
cfg_sched_perf_enable = 1'b1;

// Descriptor engine
cfg_desceng_enable = 1'b1;
cfg_desceng_prefetch = 1'b0;                // Simpler for debug

// All monitors ENABLED with verbose trace
cfg_desc_mon_enable = 1'b1;
cfg_desc_mon_err_enable = 1'b1;
cfg_desc_mon_perf_enable = 1'b1;
cfg_desc_mon_timeout_enable = 1'b1;
cfg_desc_mon_pkt_mask = 16'hFFFF;           // ALL packets
cfg_desc_mon_compl_mask = 8'h01;            // Channel 0 completions
cfg_desc_mon_debug_mask = 8'h01;             // Channel 0 debug packets

// Same for read/write monitors
cfg_rdeng_mon_enable = 1'b1;

```

```

cfg_rdeng_mon_pkt_mask = 16'hFFFF;           // Verbose
cfg_rdeng_mon_compl_mask = 8'h01;

cfg_wreng_mon_enable = 1'b1;
cfg_wreng_mon_pkt_mask = 16'hFFFF;
cfg_wreng_mon_compl_mask = 8'h01;

// AXI transfer (small for debug)
cfg_axi_rd_xfer_beats = 8'd4;
cfg_axi_wr_xfer_beats = 8'd4;

// Performance profiler
cfg_perf_enable = 1'b1;
cfg_perf_mode = 1'b0;                         // Timestamps

```

---

## 8. Configuration Best Practices

### 8.1 Monitor Configuration Guidelines

#### General Rules:

1. **Start with errors only:** Enable only `cfg_*_mon_err_enable` initially. Add other packets as needed.
  2. **Completion packets are expensive:** Only enable `cfg_*_mon_compl_enable` for specific channels during debug.
  3. **Performance packets flood MonBus:** Enable `cfg_*_mon_perf_enable` sparingly (1-2 channels maximum).
  4. **Use masks aggressively:** Set channel masks to enable monitoring only on channels of interest.
- 

### 8.2 Timeout Configuration

#### Calculation Method:

Recommended timeout = (Expected latency × Safety factor) + Margin

#### Safety factor:

- SRAM: 2-5x
- DDR3/DDR4: 5-10x
- High-latency: 10-20x

Margin: +100 cycles minimum

### Examples:

SRAM @ 200 MHz:

- Expected: 20 cycles (100 ns)
- Safety: 5x
- Timeout:  $20 \times 5 + 100 = 200$  cycles

DDR4 @ 200 MHz:

- Expected: 100 cycles (500 ns)
  - Safety: 10x
  - Timeout:  $100 \times 10 + 100 = 1100$  cycles
- 

## 8.3 Prefetch Configuration

**Enable prefetch when:** - Descriptor chains > 2 descriptors - Memory latency > 50 cycles - Throughput is priority

**Disable prefetch when:** - Area is constrained - Single descriptors only - Simplicity is priority

---

## 8.4 Burst Size Selection

### Read Burst Size:

```
Optimal burst size = min(  
    Memory controller page size,  
    4KB (AXI limit),  
    SRAM FIFO depth / 2  
)
```

Example for DDR4 (8KB page), FIFO depth 512 entries:

- Page size: 8192 bytes = 128 beats (512-bit)
- AXI limit: 4096 bytes = 64 beats
- FIFO limit: 512/2 = 256 beats
- Optimal:  $\min(128, 64, 256) = 64$  beats

**Write Burst Size:** - Usually same as read burst size - May be smaller if write FIFO depth is limited

---

## 9. Configuration Register Map Summary

| Address     | Register Name        | Fields                                                    | Section |
|-------------|----------------------|-----------------------------------------------------------|---------|
| 0x100       | GLOBAL_CTRL          | global_en,<br>global_RST                                  | -       |
| 0x120       | CHANNEL_ENABLE       | ch_en[7:0]                                                | 1       |
| 0x124       | CHANNEL_RESET        | ch_RST[7:0]                                               | 1       |
| 0x200       | SCHED_TIMEOUT_CYCLES | timeout_cycle<br>s[15:0]                                  | 2       |
| 0x204       | SCHED_CONFIG         | enable,<br>timeout_en,<br>err_en,<br>compl_en,<br>perf_en | 2       |
| 0x220       | DESCENG_CONFIG       | enable,<br>prefetch,<br>fifo_thresh                       | 3       |
| 0x224       | DESCENG_ADDR0_BASE   | addr0_base[3<br>1:0]                                      | 3       |
| 0x228       | DESCENG_ADDR0_LIMIT  | addr0_limit[3<br>1:0]                                     | 3       |
| 0x22C       | DESCENG_ADDR1_BASE   | addr1_base[3<br>1:0]                                      | 3       |
| 0x230       | DESCENG_ADDR1_LIMIT  | addr1_limit[3<br>1:0]                                     | 3       |
| 0x240-0x25F | DAXMON_*             | Descriptor<br>monitor (15<br>signals)                     | 4.1     |
| 0x260-0x27F | RDMON_*              | Read engine<br>monitor (15<br>signals)                    | 4.2     |
| 0x280-0x29F | WRMON_*              | Write engine<br>monitor (15<br>signals)                   | 4.3     |
| 0x2A0       | AXI_XFER_CFG         | rd_xfer_beats<br>,                                        | 5       |
|             |                      | wr_xfer_beat                                              |         |

| Address | Register Name | Fields                                   | Section |
|---------|---------------|------------------------------------------|---------|
|         |               | S                                        |         |
| 0x2B0   | PERF_CTRL     | perf_enable,<br>perf_mode,<br>perf_clear | 6       |

**Total Address Space:** 0x000-0x3FF (1KB)

---

## 10. Software Configuration Examples

### 10.1 C/C++ Initialization (Minimal)

```
// Minimal configuration for single-channel operation
void stream_init_minimal(volatile uint32_t *base_addr) {
    // Global enable
    base_addr[0x100/4] = 0x00000001; // GLOBAL_CTRL.global_en

    // Enable channel 0 only
    base_addr[0x120/4] = 0x00000001; // CHANNEL_ENABLE.ch_en

    // Scheduler config
    base_addr[0x200/4] = 500;           // SCHED_TIMEOUT_CYCLES (SRAM)
    base_addr[0x204/4] = 0x00000007; // SCHED_CONFIG: enable |
    timeout_en | err_en

    // Descriptor engine
    base_addr[0x220/4] = 0x00000041; // enable | fifo_thresh=4

    // Disable all monitors (minimal)
    base_addr[0x240/4] = 0x00000000; // DAXMON_ENABLE
    base_addr[0x260/4] = 0x00000000; // RDMON_ENABLE
    base_addr[0x280/4] = 0x00000000; // WRMON_ENABLE

    // AXI transfer config
    base_addr[0x2A0/4] = 0x00000808; // 8 beats read + write
}
```

---

### 10.2 C/C++ Initialization (Balanced)

```
// Balanced configuration for typical FPGA
void stream_init_balanced(volatile uint32_t *base_addr) {
    // Global enable
    base_addr[0x100/4] = 0x00000001;
```

```

// Enable all 8 channels
base_addr[0x120/4] = 0x000000FF;

// Scheduler config
base_addr[0x200/4] = 5000;           // SCHED_TIMEOUT_CYCLES (DDR4)
base_addr[0x204/4] = 0x00000007;    // SCHED_CONFIG: enable |
timeout_en | err_en

// Descriptor engine
base_addr[0x220/4] = 0x00000083;   // enable | prefetch |
fifo_thresh=8

// Descriptor AXI monitor (errors only)
base_addr[0x240/4] = 0x0000000B;   // enable | err_en | timeout_en
base_addr[0x244/4] = 10000;        // timeout_cycles
base_addr[0x24C/4] = 0x00000060;   // pkt_mask: errors + timeouts

// Read/write monitors (errors only)
base_addr[0x260/4] = 0x0000000B;   // RDMON: enable | err_en |
timeout_en
base_addr[0x280/4] = 0x0000000B;   // WRMON: enable | err_en |
timeout_en

// AXI transfer config
base_addr[0x2A0/4] = 0x00002020;   // 32 beats read + write

// Enable performance profiler
base_addr[0x2B0/4] = 0x00000003;   // enable | elapsed mode
}

```

---

## 11. Troubleshooting Configuration Issues

### Problem: No transfers occurring

**Check:** 1. cfg\_channel\_enable[n] set for channel n? 2. cfg\_sched\_enable = 1? 3. cfg\_desceng\_enable = 1? 4. Global enable set?

---

### Problem: Timeout errors

**Check:** 1. cfg\_sched\_timeout\_cycles too small? 2. Memory latency higher than expected? 3. AXI backpressure not handled?

**Solution:** - Increase timeout: `cfg_sched_timeout_cycles = 20000` - Disable temporarily: `cfg_sched_timeout_enable = 0`

---

### Problem: MonBus overflow

**Check:** 1. Too many completion packets enabled? 2. Performance packets enabled on all channels? 3. Debug packets enabled?

**Solution:** - Disable completion: `cfg_*_mon_compl_enable = 0` - Use channel masks: `cfg_*_mon_compl_mask = 8'h01` (channel 0 only) - Reduce packet types: `cfg_*_mon_pkt_mask = 16'h0060` (errors + timeouts)

---

### Problem: Low throughput

**Check:** 1. Prefetch disabled? 2. Burst size too small? 3. FIFO threshold too conservative?

**Solution:** - Enable prefetch: `cfg_desceng_prefetch = 1` - Increase bursts: `cfg_axi_rd_xfer_beats = 64` - Increase FIFO: `cfg_desceng_fifo_thresh = 12`

---

### Related Documentation

- [Register Map](#) - Complete APB register specification
  - [Clocks and Reset](#) - Timing requirements
  - [Programming Guide](#) - Software API examples
- 

**Last Updated:** 2025-12-01 **Maintained By:** STREAM Architecture Team

## Product Requirements Document (PRD)

### STREAM - Scatter-gather Transfer Rapid Engine for AXI Memory

**Version:** 1.0 **Date:** 2025-10-17 **Status:** Nearly Complete - Final Integration  
**Pending Owner:** RTL Design Sherpa Project **Parent Document:** /PRD.md

---

## 1. Executive Summary

The **STREAM** (Scatter-gather Transfer Rapid Engine for AXI Memory) is a simplified DMA-style accelerator designed for efficient memory-to-memory data movement with descriptor-based scatter-gather support. STREAM serves as a beginner-friendly tutorial demonstrating descriptor-based DMA engine design patterns while maintaining production-quality RTL practices.

### 1.1 Quick Stats

- **Modules:** ~8-10 SystemVerilog files (estimated)
- **Channels:** Maximum 8 independent channels
- **Interfaces:** APB (config), AXI4 (descriptor fetch + data read/write), MonBus (monitoring)
- **Architecture:** Simplified from RAPIDS - pure memory-to-memory
- **Tutorial Focus:** Aligned addresses only, straightforward data flow
- **Status:** Nearly complete - config interface and top-level wrapper pending

### 1.2 Project Goals

- **Primary:** Educational DMA engine demonstrating scatter-gather descriptor chains
  - **Secondary:** Production-quality RTL suitable for FPGA/ASIC implementation
  - **Tertiary:** Foundation for understanding more complex DMA architectures (e.g., RAPIDS)
- 

## 2. Key Design Principles

### 2.1 Simplifications from RAPIDS

STREAM is intentionally simplified for tutorial purposes:

| Feature                   | RAPIDS                     | STREAM                          |
|---------------------------|----------------------------|---------------------------------|
| <b>Network Interfaces</b> | Yes (Network master/slave) | ✗ No (pure memory-to-memory)    |
| <b>Address Alignment</b>  | Complex fixup logic        | ✓ <b>Aligned addresses only</b> |
| <b>Credit Management</b>  | Exponential encoding       | ✓ Simple transaction limits     |
| <b>Control Engines</b>    | Control read/write         | ✗ No (direct APB config)        |

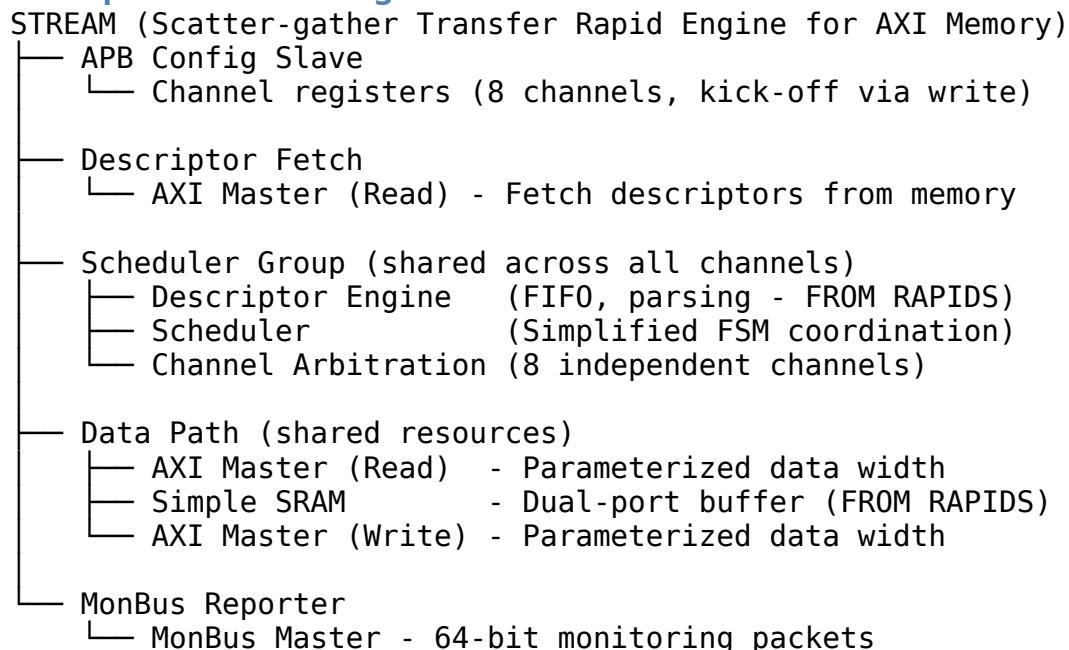
| Feature                  | RAPIDS                   | STREAM                       |
|--------------------------|--------------------------|------------------------------|
|                          | engines                  |                              |
| <b>Descriptor Length</b> | Chunks (4-byte)          | ✓ <b>Beats</b> (data width)  |
| <b>Program Engine</b>    | Complex alignment<br>FSM | ✓ Simplified<br>coordination |

## 2.2 Tutorial-Friendly Features

- **Aligned Addresses:** Source and destination addresses must be aligned to data width
- **Length in Beats:** Descriptor length specified in data beats (not bytes/chunks)
- **Single APB Write:** One APB register write kicks off entire descriptor chain
- **No Circular Buffers:** Chained descriptors with explicit termination
- **Parameterized Engines:** Multiple AXI engine versions (compile-time selection)

## 3. Architecture Overview

### 3.1 Top-Level Block Diagram



### 3.2 Data Flow

#### Descriptor-Based Transfer Sequence:

1. Software writes to APB channel register  
↓
2. Descriptor fetch via AXI descriptor master  
↓
3. Descriptor Engine parses descriptor fields  
↓
4. Scheduler coordinates data transfer:
  - a. AXI Read Engine fetches source data → SRAM buffer
  - b. AXI Write Engine writes SRAM → destination
 ↓
5. Check for chained descriptor (`next_descriptor_ptr != 0`)
  - ↓ (if chained)
6. Fetch next descriptor, repeat from step 3
  - ↓ (if last)
7. Generate MonBus completion packet

**Channel Independence:** - 8 channels operate independently - All channels share: SRAM, AXI data masters, descriptor fetch master - Arbitration required for shared resources

---

## 4. Interfaces

### 4.1 External Interfaces

| Interface                      | Type   | Width           | Purpose                         | Notes                                     |
|--------------------------------|--------|-----------------|---------------------------------|-------------------------------------------|
| <b>APB Slave</b>               | Slave  | 32-bit          | Configuration, channel kick-off | Write to channel register starts transfer |
| <b>AXI Master (Descriptor)</b> | Master | 256-bit         | Fetch descriptors from memory   | Dedicated descriptor fetch path           |
| <b>AXI Master (Data Read)</b>  | Master | Parameterizable | Read source data                | Multiple engine versions (compile-time)   |
| <b>AXI Master (Data Write)</b> | Master | Parameterizable | Write destination data          | Multiple engine versions (compile-        |

| Interface            | Type   | Width  | Purpose               | Notes                         |
|----------------------|--------|--------|-----------------------|-------------------------------|
| <b>MonBus Master</b> | Master | 64-bit | Monitor packet output | Standard AMBA format<br>time) |

## 4.2 Descriptor Format

### 256-bit Descriptor Structure:

| Bits      | Field               | Description                                                 |
|-----------|---------------------|-------------------------------------------------------------|
| [63:0]    | src_addr            | Source address (64-bit, must be aligned to data width)      |
| [127:64]  | dst_addr            | Destination address (64-bit, must be aligned to data width) |
| [159:128] | length              | Transfer length in <b>BEATS</b> (not bytes!)                |
| [191:160] | next_descriptor_ptr | Address of next descriptor (0 = last in chain)              |
| [192]     | valid               | Descriptor is valid                                         |
| [193]     | interrupt           | Generate interrupt on completion                            |
| [194]     | last                | Last descriptor in chain (explicit flag)                    |
| [195]     | error               | Error status (used for reporting)                           |
| [199:196] | channel_id          | Channel ID (0-7)                                            |
| [207:200] | priority            | Transfer priority (for arbitration)                         |
| [255:208] | reserved            | Reserved for future use                                     |

**Key Descriptor Features:** - ✓ **Chained descriptors:** next\_descriptor\_ptr links to next descriptor - ✗ **No circular buffers:** Explicit termination (last flag or ptr=0) - ✓ **Length in beats:** Simplified for tutorial (no byte/chunk conversion) - ✓ **Aligned addresses:** Tutorial constraint (performance hidden for now)

## 5. Key Components

### 5.1 Descriptor Engine (APB-Only for STREAM)

**Source:** Adapted from RAPIDS descriptor\_engine.sv

**Purpose:** - Autonomous descriptor fetch and chaining - APB interface for initial descriptor address - AXI read interface for descriptor memory fetches - Descriptor FIFO storage and distribution

**Key Features:** - ✓ **Autonomous chaining:** Automatically fetches next descriptor if `next_descriptor_ptr != 0` AND `last == 0` - ✓ **Address validation:** Validates next descriptor addresses against `cfg_addr0/1_base/limit` - ✓ **APB blocking:** APB blocked until `channel_idle == 1` (channel fully idle) - ✓ **Error handling:** AXI errors stop chaining, set `descriptor_error`, block `descriptor_valid`

**Adaptations from RAPIDS:** - ✗ **RDA removed:** STREAM is memory-to-memory only (no network interfaces) - ✓ **APB-only:** Single APB write kicks off entire descriptor chain - ✓ **Descriptor Read Address FIFO:** 2-deep FIFO stores addresses for AXI fetch (APB + chaining) - ✓ **Chaining logic:** Descriptor engine autonomously manages `next_descriptor_ptr` chaining

**Idle Signal:** - `descriptor_engine_idle` asserted when: - FSM in RD\_IDLE state - No pending descriptor fetches (address FIFO empty) - No active AXI transactions

## 5.2 Scheduler Group (Integration Wrapper)

**Purpose:** Wraps descriptor engine and scheduler into a single channel processing unit

**Architecture:**

```
scheduler_group (
    // APB interface (from APB config slave)
    .apb_valid      (apb_valid),
    .apb_ready      (apb_ready),   // Blocked when channel not idle
    .apb_addr       (descriptor_addr),

    // Channel idle signal composition (CRITICAL!)
    .channel_idle   (channel_idle),

    // Descriptor → Scheduler flow
    .descriptor_valid (desc_valid),
    .descriptor_ready (desc_ready),
    .descriptor_packet (desc_packet),

    // Data engine interfaces
    .datard_*        (datard_*),    // Read engine
    .datawr_*        (datawr_*),    // Write engine

    // Status
    .scheduler_idle  (sched_idle),
```

```

    .descriptor_idle  (desc_idle)
);

```

### Channel Idle Signal Composition:

```
// Channel is idle ONLY when BOTH sub-blocks are idle
assign channel_idle = scheduler_idle && descriptor_engine_idle;
```

### Why Both Signals Matter:

| Signal                     | Indicates                                           | Used For                                        |
|----------------------------|-----------------------------------------------------|-------------------------------------------------|
| scheduler_idle             | No active data transfers, all descriptors processed | Prevents new APB request during active transfer |
| descriptor_engine _idle    | No pending descriptor fetches (FIFO empty)          | Prevents new APB request during chaining        |
| channel_idle (AND of both) | Channel fully quiescent                             | <b>Gates APB interface</b>                      |

### APB Blocking Logic:

```
// Descriptor engine blocks APB when channel not idle
assign apb_ready = apb_skid_ready_in &&
                  !r_channel_reset_active &&
                  w_desc_addr_fifo_empty &&
                  channel_idle; // No pending fetches
                           // Scheduler +
descriptor_idle
```

### Example Scenario:

1. Software writes APB → descriptor\_addr = 0x1000
  - channel\_idle = 1 (both idle)
  - APB accepted
2. Descriptor engine fetches descriptor @ 0x1000
  - descriptor\_engine\_idle = 0 (fetch in progress)
  - channel\_idle = 0
  - APB BLOCKED
3. Descriptor pushed to scheduler
  - descriptor\_engine\_idle = 1 (fetch complete)
  - scheduler\_idle = 0 (transfer starting)
  - channel\_idle = 0
  - APB BLOCKED
4. Scheduler completes data transfer
  - Descriptor has next\_descriptor\_ptr = 0x1100 (chained!)
  - Descriptor engine autonomously fetches @ 0x1100

- descriptor\_engine\_idle = 0 (autonomous fetch)
  - channel\_idle = 0
  - APB BLOCKED
5. Final descriptor completes (last = 1 OR next\_ptr = 0)
- scheduler\_idle = 1 (transfer done)
  - descriptor\_engine\_idle = 1 (no more fetches)
  - channel\_idle = 1
  - APB UNBLOCKED (ready for next transfer!)

**Key Insight:** The AND gate ensures software cannot interrupt a descriptor chain in progress!

### 5.3 Scheduler (Simplified from RAPIDS)

**Purpose:** - Coordinate descriptor-to-data-transfer flow - Manage 8 independent channels - Arbitrate shared resources (SRAM, AXI masters)

**FSM States:**

```
typedef enum logic [7:0] {
    SCHED_IDLE          = 8'b00000001, // Idle, waiting for
    channel activation
    SCHED_FETCH_DESCRIPTOR = 8'b00000010, // Fetch descriptor via
    AXI master
    SCHED_PARSE_DESCRIPTOR = 8'b00000100, // Parse descriptor fields
    SCHED_READ_PHASE     = 8'b00001000, // Coordinate read engine
    SCHED_WRITE_PHASE    = 8'b00010000, // Coordinate write engine
    SCHED_CHAIN_CHECK    = 8'b00100000, // Check for next
    descriptor
    SCHED_COMPLETE       = 8'b01000000, // Transfer complete,
    report status
    SCHED_ERROR          = 8'b10000000 // Error state
} scheduler_state_t;
```

**Key Differences from RAPIDS:** - ✗ No credit management (just simple transaction limits) - ✗ No program engine coordination (no alignment fixup) - ✓ Simplified FSM (no control read/write phases)

### 5.3 AXI Read Engine (Streaming Pipeline - NO FSM)

**Purpose:** High-performance streaming reads from memory to SRAM buffer

**Architecture:** Pipelined streaming design (NO FSM for performance)

**Key Insight:** FSMs are horrible for performance! Instead, use: - **Arbiter** selects which channel's datard\_\* interface gets access - **Streaming pipeline**

continuously moves data when granted - **Data interface** (datard\_valid, datard\_ready, datard\_beats\_remaining) controls flow

### Data Read Interface (per channel):

```
// Channel requests read access
 datard_valid;           // Channel has read request
 datard_ready;          // Engine ready for request
 [63:0] datard_addr;    // Source address (aligned)
 [31:0] datard_beats_remaining; // Beats left to read
 [7:0] datard_burst_len;   // Preferred burst length
 [3:0] datard_channel_id; // Channel ID for tracking
```

**Multiple Versions (Compile-Time Selection):** 1. **Version 1 - Basic:** Single outstanding read, fixed burst length  
2. **Version 2 - Pipelined:** Multiple outstanding reads, configurable bursts  
3. **Version 3 - Adaptive:** Dynamic burst sizing based on remaining beats

### Operation:

1. Arbiter grants channel access based on priority
2. Engine accepts datard\_\* request (continuous streaming)
3. AXI AR channel issues read burst
4. AXI R channel streams data → SRAM (no FSM stalls!)
5. Engine updates beats\_remaining, accepts next request
6. Different channels can have different burst lengths (e.g., CH0: 8 beats, CH1: 16 beats)

### Example: Different Burst Lengths per Channel

```
// Channel 0 prefers 8-beat bursts
datard_burst_len[0] = 8'd8;

// Channel 1 prefers 16-beat bursts
datard_burst_len[1] = 8'd16;

// Engine adapts to requested burst length (within MAX_BURST_LEN)
```

## 5.4 AXI Write Engine (Streaming Pipeline - NO FSM)

**Purpose:** High-performance streaming writes from SRAM buffer to memory

**Architecture:** Pipelined streaming design (NO FSM for performance)

**Key Insight:** Same as read engine - no FSMs! Use streaming pipeline with arbiter.

### Data Write Interface (per channel):

```

// Channel requests write access
 // Channel has write
request
 // Engine ready for
request
 // Destination address
(aligned)
 // Beats left to write
 // Preferred burst length
 // Channel ID for tracking

```

**Multiple Versions (Compile-Time Selection):** 1. **Version 1 - Basic:** Single outstanding write, fixed burst length  
 2. **Version 2 - Pipelined:** Multiple outstanding writes, configurable bursts  
 3. **Version 3 - Adaptive:** Dynamic burst sizing based on remaining beats

### Operation:

1. Arbiter grants channel access based on priority
2. Engine accepts datawr\_\* request (continuous streaming)
3. Engine reads data from SRAM
4. AXI AW channel issues write address
5. AXI W channel streams data (no FSM stalls!)
6. AXI B channel receives response, updates beats\_remaining
7. Different channels can have different burst lengths (e.g., CH0: 8 beats, CH2: 32 beats)

### Read/Write Asymmetry Example:

```

// Channel can use different burst lengths for read vs write
// Example: Read in small bursts, write in large bursts
datard_burst_len[0] = 8'd8; // Read: 8 beats
datawr_burst_len[0] = 8'd16; // Write: 16 beats

// Engine handles asymmetry via SRAM buffering

```

## 5.5 Simple SRAM

**Source:** Direct copy from RAPIDS simple\_sram.sv

**Purpose:** - Dual-port SRAM buffer - Decouples read and write engines - Shared across all channels (arbitration required)

**Why Reuse:** - Standard dual-port SRAM design - Proven in RAPIDS integration tests - Parameterizable depth and width

---

## 6. Configuration and Control

### 6.1 APB Register Map

| Offset | Register       | Access | Description                                             |
|--------|----------------|--------|---------------------------------------------------------|
| 0x0000 | GLOBAL_CTRL    | RW     | Global enable, reset                                    |
| 0x0004 | GLOBAL_STATUS  | RO     | Global status, error flags                              |
| 0x0100 | CH0_CTRL       | WO     | <b>Channel 0 kick-off</b><br>(write descriptor address) |
| 0x0104 | CH0_STATUS     | RO     | Channel 0 status                                        |
| 0x0108 | CH0_DESC_ADDR  | RO     | Channel 0 current descriptor address                    |
| 0x010C | CH0_BYTES_XFER | RO     | Channel 0 bytes transferred                             |
| ...    | ...            | ...    | ... (repeat for channels 1-7)                           |
| 0x0200 | CH1_CTRL       | WO     | Channel 1 kick-off                                      |
| ...    |                |        |                                                         |
| 0x0700 | CH7_CTRL       | WO     | Channel 7 kick-off                                      |

**Kick-Off Sequence:** 1. Software writes descriptor address to CHx\_CTRL register 2. STREAM fetches descriptor from memory 3. Transfer begins automatically 4. If chained, STREAM follows next\_descriptor\_ptr automatically 5. Completion reported via MonBus packet

### 6.2 Channel Configuration

Each channel independently configurable:  
- **Descriptor start address:** Written to CHx\_CTRL  
- **Priority:** Encoded in descriptor (arbitration)  
- **Interrupt enable:** Per-descriptor flag  
- **Status monitoring:** Read CHx\_STATUS

---

## 7. Resource Sharing and Arbitration

### 7.1 Shared Resources

**All channels share:** 1. **Descriptor Fetch AXI Master** - Fetches descriptors for all channels 2. **Data Read AXI Master** - Reads source data for all channels 3. **Data Write AXI Master** - Writes destination data for all channels 4. **SRAM Buffer** -

Shared buffer (dual-port, but still arbitrated) 5. **MonBus Reporter** - Single monitor output

## 7.2 Arbitration Strategy

**Priority-Based Round-Robin:** - Channels have priority field in descriptor - Higher priority = serviced first - Within same priority: round-robin - Prevents starvation with timeout

### Example Arbitration:

Channel 0: Priority 7 (highest)  
Channel 1: Priority 5  
Channel 2: Priority 5  
Channel 3: Priority 3

Service order: CH0 → CH1 → CH2 (round-robin) → CH0 → CH1 → CH2 → CH3 ...

---

## 8. Error Detection and Recovery

### 8.1 Error Types

| Error Type                | Detection           | Response                     |
|---------------------------|---------------------|------------------------------|
| <b>Invalid descriptor</b> | Valid bit = 0       | Skip, move to next           |
| <b>Alignment error</b>    | Address not aligned | Set error flag, halt channel |
| <b>AXI SLVERR</b>         | AXI response        | Set error flag, halt channel |
| <b>AXI DECERR</b>         | AXI response        | Set error flag, halt channel |
| <b>Timeout</b>            | Transaction timeout | Set error flag, halt channel |
| <b>SRAM overflow</b>      | Buffer full         | Backpressure, wait           |

### 8.2 Error Recovery

**Per-Channel Error Handling:** - Error sets channel to CH\_ERROR state - Channel halts, does not affect other channels - Software must: 1. Read CHx\_STATUS to identify error 2. Clear error condition 3. Re-kick channel with new descriptor

**No Automatic Retry:** - Tutorial design keeps error handling simple - Software responsible for retry logic

---

## 9. MonBus Integration

### 9.1 Standard MonBus Format

Uses standard 64-bit MonBus packet format: - [63:60] Packet Type (0=ERROR, 1=COMPL, etc.) - [59:57] Protocol (custom STREAM protocol) - [56:53] Event Code (STREAM-specific events) - [52:47] Channel ID (0-7) - [46:43] Unit ID (unused for STREAM) - [42:35] Agent ID (unused for STREAM) - [34:0] Event Data (address, byte count, etc.)

### 9.2 STREAM Event Codes

| Code | Event          | Description              |
|------|----------------|--------------------------|
| 0x0  | DESC_START     | Descriptor started       |
| 0x1  | DESC_COMPLETE  | Descriptor completed     |
| 0x2  | READ_START     | Read phase started       |
| 0x3  | READ_COMPLETE  | Read phase completed     |
| 0x4  | WRITE_START    | Write phase started      |
| 0x5  | WRITE_COMPLETE | Write phase completed    |
| 0x6  | CHAIN_FETCH    | Chained descriptor fetch |
| 0xF  | ERROR          | Error occurred           |

### 9.3 Default Configuration

**Tutorial-Friendly Defaults:** - **Errors only:** `cfg_error_enable = 1`, all others = 0  
- **Interrupts:** Descriptor flag controls per-transfer interrupt - **Reduces MonBus traffic** for beginner understanding

---

## 10. Design Constraints

### 10.1 Tutorial Constraints (Intentional Simplifications)

| Constraint             | Rationale                          |
|------------------------|------------------------------------|
| Aligned addresses only | Simplify data path, hide alignment |

| Constraint                  | Rationale                                 |
|-----------------------------|-------------------------------------------|
| <b>Length in beats</b>      | complexity                                |
| <b>No circular buffers</b>  | Avoid byte/chunk conversion math          |
| <b>Single APB kick-off</b>  | Explicit termination easier to understand |
| <b>No credit management</b> | Simple software model                     |
|                             | Avoid exponential encoding complexity     |

## 10.2 Implementation Constraints

| Parameter        | Value           | Notes                      |
|------------------|-----------------|----------------------------|
| Max channels     | 8               | Compile-time parameter     |
| Max burst length | 256             | AXI4 spec limit            |
| Descriptor size  | 256 bits        | 4 × 64-bit words           |
| SRAM depth       | Parameterizable | Typical: 1024-4096 entries |
| Data width       | Parameterizable | Typical: 512-bit           |

## 11. Verification Strategy

### 11.1 Test Organization

```
projects/components/stream/dv/tests/
└── fub_tests/                      # Functional Unit Block tests
    └── descriptor_engine/          # Copy from RAPIDS (adapt imports)
        └── scheduler/              # Simplified scheduler tests
        └── axi_engines/            # Read/write engine tests
        └── sram/                   # SRAM tests

└── integration_tests/               # Multi-block scenarios
    └── single_channel/           # Single channel transfers
    └── multi_channel/            # 8-channel concurrent
    └── chained_descriptors/     # Descriptor chain tests
    └── error_handling/          # Error recovery tests
```

### 11.2 Test Levels

**Basic (Quick Smoke Tests):** - Single descriptor, single channel - Aligned addresses, simple transfers - ~30 seconds runtime

**Medium (Typical Scenarios):** - Multiple descriptors, 2-4 channels - Chained descriptors (2-3 deep) - ~90 seconds runtime

**Full (Comprehensive Validation):** - All 8 channels concurrent - Long descriptor chains (10+ deep) - Error injection, stress testing - ~180 seconds runtime

---

## 12. Performance Characteristics

### 12.1 Throughput by Engine Version

**V1 (Low Performance - Tutorial Mode):** - **Throughput:** 0.14 beats/cycle (DDR4), 0.40 beats/cycle (SRAM) - **Architecture:** Single outstanding transaction, blocks on completion - **Use Case:** Tutorial examples, embedded systems, low-latency SRAM

**V2 (Medium Performance - Command Pipelined):** - **Throughput:** 0.94 beats/cycle (DDR4), 0.85 beats/cycle (SRAM) - **Improvement:** 6.7x over V1 (DDR4), 2.1x over V1 (SRAM) - **Architecture:** Command queue (4-8 deep), hides memory latency - **Use Case:** General-purpose FPGA, DDR3/DDR4, best area efficiency

**V3 (High Performance - Out-of-Order):** - **Throughput:** 0.98 beats/cycle (DDR4), 0.92 beats/cycle (SRAM) - **Improvement:** 7.0x over V1 (DDR4), 2.3x over V1 (SRAM) - **Architecture:** OOO command selection, maximizes memory controller efficiency - **Use Case:** Datacenter, ASIC, HBM2, high-performance memory

**Key Insight:** Benefit scales with memory latency. V1 throughput degrades from 40% (SRAM) to 14% (DDR4), while V2/V3 maintain 94-98% throughput regardless of latency.

**Configuration Parameters:** - `ENABLE_CMD_PIPELINE = 0`: V1 (default, tutorial mode) - `ENABLE_CMD_PIPELINE = 1`: V2 (command pipelined) - `ENABLE_CMD_PIPELINE = 1, ENABLE_000_DRAIN = 1` (write) or `ENABLE_000_READ = 1` (read): V3

**Factors Affecting Throughput:** - Memory latency (V2/V3 hide latency via pipelining) - SRAM buffer size (limits burst pipelining) - Channel arbitration overhead - Descriptor fetch latency - Engine version (V1/V2/V3 configuration)

### 12.2 Latency

**Transfer Latency Breakdown:** - Descriptor fetch: ~10-50 cycles (depends on memory) - Read phase:  $(\text{length} / \text{burst\_len}) \times \text{burst\_latency}$  - Write phase:

$(\text{length} / \text{burst\_len}) \times \text{burst\_latency}$  - End-to-end: Typically <200 cycles for small transfers (V1), <100 cycles (V2/V3 pipelined)

**V2/V3 Latency Hiding:** - Command pipelining overlaps descriptor fetch, read, write operations - Multiple outstanding transactions hide memory latency - OOO completion (V3) reduces head-of-line blocking

## 12.3 Resource Utilization by Engine Version

**V1 Configuration (Tutorial - Minimum Area):** - Total: ~9,500 LUTs + 64 KB SRAM - Descriptor Engine (8x): ~2,400 LUTs - Scheduler (8x): ~3,200 LUTs - AXI Read Engine: ~1,250 LUTs - AXI Write Engine: ~1,250 LUTs - SRAM Controller: ~1,600 LUTs - APB Config: ~350 LUTs - MonBus AXIL Group: ~1,000 LUTs

**V2 Configuration (Balanced - Best Area Efficiency):** - Total: ~11,000 LUTs + 64 KB SRAM (1.16x area) - AXI Read Engine: ~2,000 LUTs (1.6x increase, 6.7x throughput) - AXI Write Engine: ~2,500 LUTs (2.0x increase, 6.7x throughput) - Other blocks: Same as V1

**V3 Configuration (Maximum Performance):** - Total: ~14,000 LUTs + 64 KB SRAM (1.47x area) - AXI Read Engine: ~3,500 LUTs (2.8x increase, 7.0x throughput) - AXI Write Engine: ~4,000 LUTs (3.2x increase, 7.0x throughput) - Other blocks: Same as V1

**Area Efficiency Comparison:** - V1: 1.00 throughput / 1.00 area = 1.00 - V2: 6.70 throughput / 1.16 area = 5.78 (best efficiency) - V3: 7.00 throughput / 1.47 area = 4.76

**Recommendation:** V2 provides best area efficiency for most use cases. V3 justified only for high-performance memory controllers that support OOO responses.

---

## 13. Development Roadmap

### 13.1 Phase 1: Foundation (Current)

- ✓ Directory structure
- ✓ Package definitions (`stream_pkg.sv`)
- ✓ Imports header (`stream_imports.svh`)
- ⏱ Documentation (this PRD)

## 13.2 Phase 2: Core Blocks

- Adapt descriptor engine from RAPIDS
- Design simplified scheduler
- Create APB config interface
- Copy simple SRAM from RAPIDS

## 13.3 Phase 3: Data Path

- AXI read engine (version 1 - basic)
- AXI write engine (version 1 - basic)
- SRAM integration
- Channel arbitration

## 13.4 Phase 4: Integration

- Top-level module
- MonBus reporter
- Integration testbench
- Single-channel validation

## 13.5 Phase 5: Multi-Channel

- 8-channel support
- Arbiter implementation
- Multi-channel tests
- Performance tuning

## 13.6 Phase 6: Advanced Engines (Future - V2/V3)

**Goal:** Add parameterized high-performance engine variants

**V2 - Command Pipelined (Medium Performance):** - Command queue implementation (4-8 deep) - W drain FSM for write engine - B response scoreboard (write) or in-order R reception (read) - Per-command SRAM pointer tracking - Parameter: `ENABLE_CMD_PIPELINE = 1` - Expected: 6.7x throughput improvement over V1

**V3 - Out-of-Order Completion (High Performance):** - OOO command selection logic - Transaction ID matching (AXI ID to queue entry) - SRAM data availability checking (write engine) - R beat matching to queue entry (read engine) - Parameters: `ENABLE_CMD_PIPELINE = 1`, `ENABLE_000_DRAIN = 1` (write) or `ENABLE_000_READ = 1` (read) - Expected: 7.0x throughput improvement over V1

**Deliverables:** - Updated RTL with parameterization - Performance comparison tests (V1 vs V2 vs V3) - Tutorial documentation explaining trade-offs - Area/throughput measurements on target FPGA

---

## 14. Educational Value

### 14.1 Learning Objectives

**STREAM teaches:** 1. **Descriptor-based DMA design patterns** - Descriptor structure and parsing - Chained descriptors (scatter-gather) - Descriptor fetch via AXI

#### 2. **AXI4 Memory Interface Usage**

- Burst transactions
- Address/data/response channels
- Outstanding transactions

#### 3. **Resource Sharing and Arbitration**

- Multiple channels sharing AXI masters
- Priority-based arbitration
- Conflict resolution

#### 4. **FSM Design and Coordination**

- Multiple interconnected FSMs
- State machine composition
- Error handling

#### 5. **Buffer Management**

- SRAM-based buffering
- Rate matching
- Flow control

### 14.2 Progression Path

**Learning Sequence:** 1. **STREAM (this project):** Memory-to-memory DMA, aligned addresses 2. **STREAM Extended:** Add alignment fixup, more complex scenarios 3. **RAPIDS:** Add network interfaces, credit management, full complexity

---

## 15. Success Criteria

### 15.1 Functional

- ✓ Single descriptor transfer working
- ✓ Chained descriptors (2-3 deep)
- ✓ Multi-channel operation (8 channels concurrent)
- ✓ Error detection and reporting
- ✓ MonBus packet generation
- ✓ >90% functional test coverage

### 15.2 Quality

- ✓ Verilator compiles with 0 warnings
- ✓ All tests passing (100% success rate)
- ✓ Comprehensive documentation
- ✓ Tutorial-quality code comments

### 15.3 Performance

- ✓ Achieves >80% of theoretical AXI bandwidth
  - ✓ <5K LUT utilization (excluding SRAM)
  - ✓ <200 cycle end-to-end latency for small transfers
- 

## 16. Open Questions (For Review)

### 16.1 Descriptor Engine Adaptation

- **Q:** Should descriptor engine use APB-only, RDA-only, or mixed mode?
- **A (pending):** TBD - depends on software use case preference

### 16.2 AXI Descriptor Master

- **Q:** Fixed 256-bit width, or parameterizable?
- **A (pending):** Propose fixed 256-bit for simplicity

### 16.3 Channel Arbitration

- **Q:** Fixed priority, or dynamic priority based on age/fairness?
- **A (pending):** Propose fixed priority with round-robin for tutorial simplicity

### 16.4 SRAM Partitioning

- **Q:** Single shared SRAM, or per-channel SRAMs?

- **A (pending):** Propose single shared SRAM with arbitration (matches RAPIDS pattern)
- 

## 16. Attribution and Contribution Guidelines

### 16.1 Git Commit Attribution

When creating git commits for STREAM documentation or implementation:

**Use:**

Documentation and implementation support by Claude.

**Do NOT use:**

Co-Authored-By: Claude <noreply@anthropic.com>

**Rationale:** STREAM documentation and organization receives AI assistance for structure and clarity, while design concepts and architectural decisions remain human-authored.

---

## 16.2 PDF Generation Location

**IMPORTANT: PDF files should be generated in the docs directory:**

/mnt/data/github/rtldesignsherpa/projects/components/stream/docs/

**Quick Command:** Use the provided shell script:

```
cd /mnt/data/github/rtldesignsherpa/projects/components/stream/docs  
./generate_pdf.sh
```

The shell script will automatically: 1. Use the md\_to\_docx.py tool from bin/ 2. Process the stream\_spec index file 3. Generate both DOCX and PDF files in the docs/ directory 4. Create table of contents and title page

 **See:** bin/md\_to\_docx.py for complete implementation details

---

## 17. References

### 17.1 Internal Documentation

- **RAPIDS PRD:** projects/components/rapids/PRD.md - Parent architecture

- **RAPIDS Descriptor Engine:** [projects/components/rapids/rtl/rapids\\_fub/descriptor\\_engine.sv](projects/components/rapids/rtl/rapids_fub/descriptor_engine.sv)
- **RAPIDS Simple SRAM:** [projects/components/rapids/rtl/rapids\\_fub/simple\\_sram.sv](projects/components/rapids/rtl/rapids_fub/simple_sram.sv)
- **AMBA PRD:** <rtl/amba/PRD.md> - MonBus integration
- **Repository Guide:** </CLAUDE.md> - Design patterns and conventions

## 17.2 External References

- **AXI4 Specification:** ARM IHI0022E
  - **APB Specification:** ARM IHI0024C
  - CocoTB Documentation: <https://docs.cocotb.org/>
  - Verilator Manual: <https://verilator.org/guide/latest/>
- 

**Document Version:** 1.0 **Last Updated:** 2025-10-17 **Review Cycle:** Weekly during initial design **Next Review:** TBD (after user feedback) **Owner:** RTL Design Sherpa Project

---

## Navigation

- ← **Back to Root:** </PRD.md>
- **Parent Architecture:** <projects/components/rapids/PRD.md>
- **AI Guidance:** <CLAUDE.md> (to be created)
- **Quick Start:** <README.md> (to be created)

## Claude Code Guide: STREAM Subsystem

**Version:** 1.0 **Last Updated:** 2025-10-17 **Purpose:** AI-specific guidance for working with STREAM subsystem

---

## Quick Context

**What:** STREAM - Scatter-gather Transfer Rapid Engine for AXI Memory

**Status:** 🟡 Initial design - tutorial-focused DMA engine **Your Role:** Help users build a beginner-friendly descriptor-based DMA engine

📖 **Complete Specification:** <projects/components/stream/PRD.md> ← **Always reference this for technical details**

---

## Global Requirements Reference

### **IMPORTANT:** Review `/GLOBAL_REQUIREMENTS.md` for all mandatory requirements

All mandatory requirements are consolidated in the global requirements document: - **See:** `/GLOBAL_REQUIREMENTS.md` - Repository-wide mandatory requirements - **STREAM-Specific:** Attribution format, tutorial focus (intentional simplifications) - **Universal:** TB location, three methods, TBBBase inheritance, 100% success

This CLAUDE.md provides STREAM-specific guidance. Also review: - Root `/CLAUDE.md` - Repository-wide patterns - `projects/components/CLAUDE.md` - Project area standards (reset macros, FPGA attributes) - `bin/CocoTBFramework/CLAUDE.md` - Framework usage patterns

---

## Critical Rules for This Subsystem

### **Rule #0: Attribution Format for Git Commits**

**IMPORTANT:** When creating git commit messages for STREAM documentation or code:

**Use:**

Documentation and implementation support by Claude.

**Do NOT use:**

Co-Authored-By: Claude <noreply@anthropic.com>

**Rationale:** STREAM receives AI assistance for structure and clarity, while design concepts and architectural decisions remain human-authored.

### **Rule #0.1: TUTORIAL FOCUS - Intentional Simplifications**

**⚠ STREAM is INTENTIONALLY SIMPLIFIED for educational purposes ⚠**

**Key Simplifications (DO NOT “fix” these):** 1. ✓ Aligned addresses only - No alignment fixup logic (kept for RAPIDS) 2. ✓ Length in beats - Not bytes or chunks (simplifies math) 3. ✓ No circular buffers - Explicit chain termination

only 4. ✓ **No credit management** - Simple transaction limits 5. ✓ **Pure memory-to-memory** - No network interfaces

**When users ask “Can we add alignment fixup?”:** - ✓ **Correct answer:** “STREAM intentionally keeps addresses aligned for tutorial simplicity. For complex alignment, see RAPIDS.” - ✗ **Wrong answer:** “Sure, let me add alignment logic...” (defeats tutorial purpose!)

## Rule #0.1: Testbench Location and Test Structure (MANDATORY)

📖 See: /GLOBAL\_REQUIREMENTS.md Section 2.1 for complete requirement

### STREAM-Specific Directory Structure:

```
projects/components/stream/dv/
  └── tbclasses/                                # ★ STREAM TB classes (project
    ┌─────────────────────────────────────────────────┐
    │   scheduler_tb.py      # Scheduler testbench
    │   descriptor_engine_tb.py  # Descriptor engine testbench
    │   axi_engine_tb.py      # AXI engine testbenches
    └── tests/fub_tests/      # Test runners import from
      └── tbclasses/
```

### STREAM Import Pattern:

```
# Import STREAM TB from project area
from projects.components.stream.dv.tbclasses.scheduler_tb import StreamSchedulerTB

# Shared utilities from framework
from CocoTBFramework.tbclasses.shared.tbbase import TBBBase
```

📖 **Complete Pattern:** projects/components/rapids/CLAUDE.md Rule #0.1

## Rule #0.2: Three Mandatory TB Methods (MANDATORY)

📖 See: /GLOBAL\_REQUIREMENTS.md Section 2.2 for complete requirement

### STREAM-Specific Context:

STREAM has simpler config requirements than RAPIDS - most config can be set after reset.

### Standard STREAM Pattern:

```
class StreamSchedulerTB(TBBBase):
    async def setup_clocks_and_reset(self):
        """Standard STREAM initialization"""
        await self.start_clock('clk', freq=10, units='ns')
```

```

await self.assert_reset()
await self.wait_clocks('clk', 10)
await self.deassert_reset()
await self.wait_clocks('clk', 5)

async def assert_reset(self):
    self.dut.rst_n.value = 0

async def deassert_reset(self):
    self.dut.rst_n.value = 1

```

**Note:** Unlike RAPIDS, STREAM typically doesn't need config set before reset.

### Rule #1: REUSE from RAPIDS Where Appropriate

**Direct Reuse (No Modification):** - ✓ descriptor\_engine.sv - Works with STREAM descriptors - ✓ simple\_sram.sv - Standard dual-port SRAM - ✓ Descriptor engine tests - Adapt imports only

**Adapt from RAPIDS:** - ⚠ scheduler.sv - **Simplify FSM** (no credit management, no control engines) - ⚠ AXI engines - **Create simplified versions** (no alignment fixup)

**Create New for STREAM:** - 📋 APB config interface - PeakRDL-generated (like HPET), 8 channels, kick-off registers - 📋 Top-level integration - Different interface set

**Always Ask Yourself:** “Can I reuse from RAPIDS instead of creating new?”

### Rule #2: Descriptor Format is DIFFERENT from RAPIDS

**STREAM Descriptor (256-bit):** - src\_addr (64-bit), dst\_addr (64-bit), length (beats), next\_descriptor\_ptr (32-bit) - **Length is in BEATS, not chunks!**

**RAPIDS Descriptor:** - Uses chunks (4-byte units) - Has alignment metadata

**When comparing/referencing RAPIDS:** - ✓ “RAPIDS uses chunks, STREAM uses beats for tutorial simplicity” - ✗ Don’t assume RAPIDS descriptor format applies to STREAM

### Rule #3: Know the Shared Resources

**All 8 channels share:** 1. Descriptor fetch AXI master 2. Data read AXI master 3. Data write AXI master 4. SRAM buffer 5. MonBus reporter

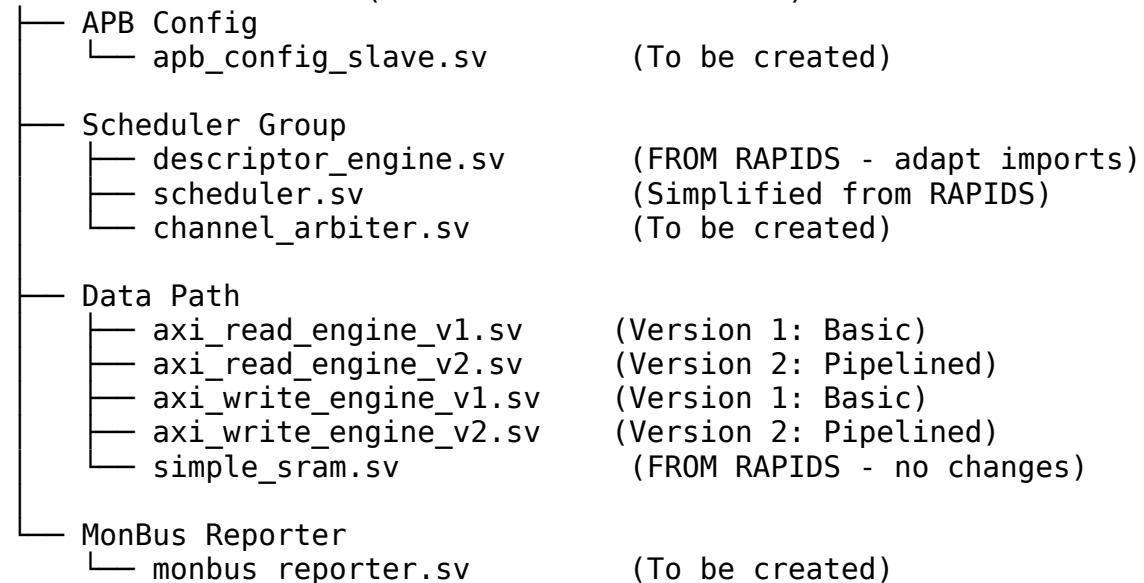
**Arbitration is required!** - Never assume exclusive access - All shared resources need arbiter logic - Priority-based round-robin (descriptor priority field)

---

## Architecture Quick Reference

### Block Organization

STREAM Architecture (Estimated: 8-10 modules)



### Module Status

| Module                | Source            | Status    | Notes                               |
|-----------------------|-------------------|-----------|-------------------------------------|
| descriptor_engine.sv  | RAPIDS (copy)     | ✓ Copied  | Adapt #include only                 |
| simple_sram.sv        | RAPIDS (copy)     | ✓ Copied  | No changes needed                   |
| stream_pkg.sv         | New               | ✓ Created | Descriptor format defined           |
| stream_imports.svh    | New               | ✓ Created | Package imports                     |
| scheduler.sv          | RAPIDS (simplify) | ⌚ Pending | Remove credit mgmt, control engines |
| apb_config_slave.sv   | New               | ⌚ Pending | 8 channel registers                 |
| axi_read_engine_v1.sv | New               | ⌚ Pending | Basic version                       |

| Module                 | Source | Status  | Notes                      |
|------------------------|--------|---------|----------------------------|
| axi_write_engine_v1.sv | New    | Pending | Basic version              |
| channel_arbiter.sv     | New    | Pending | Priority-based round-robin |
| monbus_reporter.sv     | New    | Pending | STREAM event codes         |
| stream_top.sv          | New    | Pending | Top-level integration      |

## Common User Questions and Responses

**Q: "How is STREAM different from RAPIDS?"**

**A: STREAM is intentionally simplified for tutorial purposes:**

**Simplifications:** | Feature | RAPIDS | STREAM | | ----- | ----- | ----- | |  
**Interfaces** | APB + AXI + Network | APB + AXI only | | **Address Alignment** |  
Complex fixup | Aligned only | | **Credit Management** | Exponential encoding |  
Simple limits | | **Descriptor Length** | Chunks (4-byte) | Beats (data width) | |  
**Control Engines** | Control read/write | None (direct APB) |

**Learning Path:** 1. **STREAM** - Basic DMA with scatter-gather 2. **STREAM Extended** - Add alignment fixup 3. **RAPIDS** - Full complexity with network + credit management

 **See:** PRD.md Section 2.1 for complete comparison table

**Q: "How do I kick off a transfer?"**

**A: Single APB write starts descriptor chain:**

```
// Software writes descriptor address to channel register
write_apb(ADDR_CH0_CTRL, 0x1000_0000); // Start address of descriptor

// STREAM automatically:
// 1. Fetches descriptor from 0x1000_0000
// 2. Parses src_addr, dst_addr, length
// 3. Reads source data → SRAM
// 4. Writes SRAM → destination
// 5. Checks next_descriptor_ptr
//     - If != 0: Fetch next descriptor, repeat
//     - If == 0 or last flag set: Complete
```

## **Chained Descriptors:**

```
Descriptor 0 @ 0x1000_0000:  
    src_addr = 0x2000_0000  
    dst_addr = 0x3000_0000  
    length = 64 beats  
    next_descriptor_ptr = 0x1000_0100 ← Points to next descriptor  
  
Descriptor 1 @ 0x1000_0100:  
    src_addr = 0x2000_1000  
    dst_addr = 0x3000_1000  
    length = 32 beats  
    next_descriptor_ptr = 0x0000_0000 ← Last descriptor (0 = stop)
```

 See: PRD.md Section 3.2 for complete data flow

**Q: "How many channels can I use?"**

**A: Maximum 8 independent channels:**

**Channel Operation:** - Each channel has own FSM state - Each channel can have separate descriptor chain - All channels share: AXI masters, SRAM, MonBus

**Arbitration:** - Priority-based (descriptor priority field) - Round-robin within same priority - Prevents starvation with timeout

**Example:**

```
// Kick off 3 channels concurrently  
write_apb(ADDR_CH0_CTRL, desc0_addr); // Channel 0: Priority 7  
write_apb(ADDR_CH1_CTRL, desc1_addr); // Channel 1: Priority 5  
write_apb(ADDR_CH2_CTRL, desc2_addr); // Channel 2: Priority 5  
  
// Service order: CH0 → CH1 → CH2 → CH0 → CH1 → CH2 ...
```

 See: PRD.md Section 7 for arbitration details

**Q: "What's the descriptor format?"**

**A: 256-bit descriptor (4 × 64-bit words):**

```
typedef struct packed {  
    logic [63:0] reserved;           // [255:192] Reserved  
    logic [7:0] priority;           // [207:200] Priority  
    logic [3:0] channel_id;         // [199:196] Channel ID  
    logic error;                   // [195] Error flag  
    logic last;                    // [194] Last in chain  
    logic interrupt;               // [193] Generate interrupt  
    logic valid;                   // [192] Valid descriptor
```

```

    logic [31:0] next_descriptor_ptr; // [191:160] Next descriptor
address
    logic [31:0] length; // [159:128] Length in BEATS
★
    logic [63:0] dst_addr; // [127:64] Destination
address
    logic [63:0] src_addr; // [63:0] Source address
} descriptor_t;

```

★ **CRITICAL:** length is in **BEATS**, not bytes or chunks!

### Example:

Data width = 512 bits = 64 bytes  
Length = 16 beats  
Total transfer =  $16 \times 64 = 1024$  bytes

 See: PRD.md Section 4.2 for complete descriptor specification

### Q: "How do I run STREAM tests?"

#### A: Multi-layered test approach (same as RAPIDS):

```

# 1. FUB (Functional Unit Block) Tests - Individual blocks
pytest projects/components/stream/dv/tests/fub_tests/scheduler/ -v
pytest
projects/components/stream/dv/tests/fub_tests(descriptor_engine/ -v
pytest projects/components/stream/dv/tests/fub_tests/ -v # All FUB
tests

# 2. Integration Tests - Multi-block scenarios
pytest projects/components/stream/dv/tests/integration_tests/ -v

# Run with waveforms for debugging
pytest projects/components/stream/dv/tests/fub_tests/scheduler/ --
vcd=debug.vcd
gtkwave debug.vcd

```

**Test Organization:** - **FUB tests:** Focus on individual block functionality -  
**Integration tests:** Verify block-to-block interfaces and complete data flows

---

## Integration Patterns

### Pattern 1: Basic STREAM Instantiation

```

stream_top #(
    .NUM_CHANNELS(8),
    .DATA_WIDTH(512),

```

```

    .ADDR_WIDTH(64),
    .SRAM_DEPTH(4096)
) u_stream (
    // Clock and Reset
    .aclk                (system_clk),
    .aresetn             (system_rst_n),

    // APB Configuration Interface
    .s_apb_paddr         (apb_paddr),
    .s_apb_psel          (apb_psel),
    .s_apb_penable       (apb_penable),
    .s_apb_pwrite        (apb_pwrite),
    .s_apb_pwdata        (apb_pwdata),
    .s_apb_pready        (apb_pready),
    .s_apb_prdata        (apb_prdata),
    .s_apb_pslverr       (apb_pslverr),

    // AXI Master - Descriptor Fetch (256-bit)
    .m_axi_desc_araddr   (desc_araddr),
    .m_axi_desc_arlen    (desc_arlen),
    .m_axi_desc_arsize   (desc_arsize),
    .m_axi_desc_arvalid  (desc_arvalid),
    .m_axi_desc_arready  (desc_arready),
    .m_axi_desc_rdata    (desc_rdata),
    .m_axi_desc_rresp    (desc_rresp),
    .m_axi_desc_rlast    (desc_rlast),
    .m_axi_desc_rvalid   (desc_rvalid),
    .m_axi_desc_rready   (desc_rready),

    // AXI Master - Data Read (parameterizable width)
    .m_axi_rd_araddr    (rd_araddr),
    // ... (full AXI4 AR + R channels)

    // AXI Master - Data Write (parameterizable width)
    .m_axi_wr_awaddr    (wr_awaddr),
    // ... (full AXI4 AW + W + B channels)

    // MonBus Output
    .monbus_pkt_valid    (stream_mon_valid),
    .monbus_pkt_ready    (stream_mon_ready),
    .monbus_pkt_data     (stream_mon_data)
);

```

## Pattern 2: Descriptor Creation (Software Model)

```

// C/C++ software model for descriptor creation
typedef struct {
    uint64_t src_addr;           // Source address (must be aligned!)
    uint64_t dst_addr;           // Destination address (must be
aligned!)

```

```

        uint32_t length;           // Transfer length in BEATS
        uint32_t next_descriptor_ptr; // Next descriptor address (0 =
last)
        uint8_t control;          // valid | interrupt | last | error |
channel_id | priority
} stream_descriptor_t;

// Create descriptor chain
stream_descriptor_t desc[2];

// Descriptor 0
desc[0].src_addr = 0x80000000ULL; // Aligned to 64B (512-bit data)
desc[0].dst_addr = 0x90000000ULL;
desc[0].length = 64; // 64 beats × 64 bytes = 4KB transfer
desc[0].next_descriptor_ptr = (uint32_t)&desc[1]; // Chain to next
desc[0].control = 0x01; // valid = 1

// Descriptor 1 (last)
desc[1].src_addr = 0x80001000ULL;
desc[1].dst_addr = 0x90001000ULL;
desc[1].length = 32; // 32 beats × 64 bytes = 2KB transfer
desc[1].next_descriptor_ptr = 0; // Last descriptor
desc[1].control = 0x45; // valid | last | interrupt

// Kick off transfer
write_apb_reg(CH0_CTRL, (uint32_t)&desc[0]);

```

### Pattern 3: MonBus Integration

```

// Always add downstream FIFO for MonBus
gaxi_fifo_sync #(
    .DATA_WIDTH(64),
    .DEPTH(256)
) u_stream_mon_fifo (
    .i_clk      (aclk),
    .i_rst_n    (aresetn),
    .i_data     (monbus_pkt_data),
    .i_valid    (monbus_pkt_valid),
    .o_ready    (monbus_pkt_ready),
    .o_data     (fifo_mon_data),
    .o_valid    (fifo_mon_valid),
    .i_ready    (consumer_ready)
);

```

---

## Anti-Patterns to Catch

### X Anti-Pattern 1: Adding Alignment Fixup

x WRONG:

```
"Let me add alignment fixup logic to handle unaligned addresses..."
```

✓ CORRECTED:

```
"STREAM intentionally requires aligned addresses for tutorial simplicity.
```

```
If you need unaligned transfers, that's a great learning exercise for extending STREAM, or use RAPIDS which has full alignment support."
```

### X Anti-Pattern 2: Using Length in Bytes

x WRONG:

```
descriptor.length = 4096; // Thinking this is 4096 bytes
```

✓ CORRECTED:

```
// Length is in BEATS, not bytes!
```

```
// For 512-bit data width (64 bytes per beat):
```

```
descriptor.length = 4096 / 64; // = 64 beats for 4096 bytes
```

### X Anti-Pattern 3: Circular Buffer Descriptors

x WRONG:

```
// Descriptor chain that loops back
```

```
desc[9].next_descriptor_ptr = &desc[0]; // Circular!
```

✓ CORRECTED:

```
"STREAM does not support circular buffers (no stop condition).
```

```
Always terminate chains explicitly:
```

```
desc[last].next_descriptor_ptr = 0; // Stop
```

```
desc[last].last = 1; // Explicit last flag
```

### X Anti-Pattern 4: Assuming Exclusive Channel Access

x WRONG:

```
// Assume channel has exclusive AXI master access
```

```
assign m_axi_arvalid = channel_request; // No arbitration!
```

✓ CORRECTED:

```
// All channels share AXI masters - arbitration required
```

```
channel_arbiter u_arbiter (
    .ch_requests (channel_requests[7:0]),
    .ch_grant   (channel_grant[7:0]),
    .axi_master_available (axi_master_idle)
);
```

---

## Debugging Workflow

### Issue: Descriptor Not Fetched

**Check in order:** 1. ✓ APB write to CHx\_CTRL register successful? 2. ✓ Descriptor address valid? 3. ✓ AXI descriptor master not stalled? 4. ✓ Descriptor memory accessible? 5. ✓ Arbiter granting descriptor fetch?

#### Debug commands:

```
pytest projects/components/stream/dv/tests/fub_tests/scheduler/ -v -s  
# Verbose test  
pytest projects/components/stream/dv/tests/fub_tests/scheduler/ --  
vcd=debug.vcd  
gtkwave debug.vcd # Inspect FSM state transitions
```

### Issue: Data Transfer Stalls

**Check in order:** 1. ✓ SRAM buffer depth sufficient? 2. ✓ Source/destination addresses aligned? 3. ✓ AXI read/write engines getting grants? 4. ✓ Downstream AXI ready signals asserted? 5. ✓ Channel not in error state?

**Waveform Analysis:** - Check SRAM read/write pointers - Verify AXI AR/AW/R/W/B handshakes - Inspect scheduler FSM state - Check arbiter grant signals

### Issue: Chained Descriptors Not Following

**Check in order:** 1. ✓ next\_descriptor\_ptr != 0? 2. ✓ last flag not set prematurely? 3. ✓ Descriptor fetch completing successfully? 4. ✓ Scheduler transitioning to CHAIN\_CHECK state? 5. ✓ MonBus showing CHAIN\_FETCH event?

---

## Testing Guidance

### Test Organization

```
projects/components/stream/dv/tests/  
└── fub_tests/                      # Individual block tests  
    ├── descriptor_engine/            # Adapt from RAPIDS tests  
    ├── scheduler/                   # Simplified scheduler tests  
    ├── axi_engines/                 # Read/write engine tests  
    └── sram/                        # SRAM tests (from RAPIDS)  
  
└── integration_tests/               # Multi-block scenarios  
    ├── single_channel/              # Single channel transfers  
    └── multi_channel/               # 8-channel concurrent
```

```
└── chained_descriptors/      # Descriptor chain tests  
  └── error_handling/        # Error recovery tests
```

## Test Levels

**Basic (Quick Smoke Tests):** - Single descriptor, single channel - Aligned addresses, simple transfers - ~30 seconds runtime

**Medium (Typical Scenarios):** - Multiple descriptors, 2-4 channels - Chained descriptors (2-3 deep) - ~90 seconds runtime

**Full (Comprehensive Validation):** - All 8 channels concurrent - Long descriptor chains (10+ deep) - Error injection, stress testing - ~180 seconds runtime

---

## Key Documentation Links

### Always Reference These

**This Subsystem:** - projects/components/stream/PRD.md - **Complete specification** - projects/components/stream/README.md - Quick start guide (to be created) - projects/components/stream/known\_issues/ - Bug tracking

**Related:** - projects/components/rapids/PRD.md - Parent architecture (for comparison) - rtl/amba/PRD.md - MonBus integration - /PRD.md - Master requirements - /CLAUDE.md - Repository guide

---

## Quick Commands

# View complete specification

```
cat projects/components/stream/PRD.md
```

# Check package definition

```
cat projects/components/stream/rtl/includes/stream_pkg.sv
```

# Run tests (once created)

```
pytest projects/components/stream/dv/tests/fub_tests/ -v
```

```
pytest projects/components/stream/dv/tests/integration_tests/ -v
```

# Lint (once RTL created)

```
verilator --lint-only
```

```
projects/components/stream/rtl/stream_fub/scheduler.sv
```

# Search for modules

```
find projects/components/stream/rtl/ -name "*.sv" -exec grep -H  
"^module" {} \;
```

---

## Remember

1. **Tutorial focus** - Intentional simplifications for learning
  2. **Reuse from RAPIDS** - Descriptor engine, SRAM, patterns
  3. **Length in beats** - Not bytes or chunks!
  4. **Aligned addresses** - No fixup logic
  5. **Chained descriptors** - No circular buffers
  6. **8 channels** - Shared resources, arbitration required
  7. **MonBus standard** - Same format as AMBA/RAPIDS
  8. **Testbench reuse** - Always create TB classes in  
bin/CocoTBFramework/tbclasses/stream/
- 

## PDF Generation Location

**IMPORTANT: PDF files should be generated in the docs directory:**

```
/mnt/data/github/rtldesignsherpa/projects/components/stream/docs/
```

**Quick Command:** Use the provided shell script:

```
cd /mnt/data/github/rtldesignsherpa/projects/components/stream/docs  
../generate_pdf.sh
```

The shell script will automatically: 1. Use the md\_to\_docx.py tool from bin/ 2. Process the stream\_spec index file 3. Generate both DOCX and PDF files in the docs/ directory 4. Create table of contents and title page

**See:** bin/md\_to\_docx.py for complete implementation details

---

**Version:** 1.0 **Last Updated:** 2025-10-17 **Maintained By:** RTL Design Sherpa Project

## STREAM Register Definitions

**Purpose:** PeakRDL register files and generated RTL for STREAM configuration

---

## Directory Structure

```
regs/
└── README.md                      # This file
    └── stream_regs.rdl            # Register definition (to be created)
        └── generated/             # PeakRDL-generated outputs (to be
            created)
            └── rtl/
                └── stream_regs.sv
                    └── stream_regs_pkg.sv
            └── docs/
                └── stream_regs.html
```

---

## Register Generation Workflow

### Phase 1: Define Register Map (Future)

Create `stream_regs.rdl` following the same pattern as `apb_hpet`.

### Phase 2: PeakRDL-Generated Registers (Future)

Following the same pattern as `projects/components/apb_hpet/`:

#### 1. Define Register Map (`stream_regs.rdl`)

```
addrmap stream_regs {
    name = "STREAM Configuration Registers";

    // Global control
    reg {
        field { sw=rw; hw=r; } enable;
        field { sw=rw; hw=r; } reset;
    } GLOBAL_CTRL @ 0x00;

    // Channel registers (8 channels × 16 bytes)
    regfile channel_regs {
        reg { ... } CH_CTRL;
        reg { ... } CH_STATUS;
        reg { ... } CH_RD_BURST;
        reg { ... } CH_WR_BURST;
    };

    channel_regs CH[8] @ 0x10 += 0x10;
};
```

#### 2. Generate RTL

```

cd projects/components/stream/regs
../../bin/peakrdl_generate.py stream_regs.rdl --copy-rtl
./rtl/stream_macro

```

### 3. Create APB Config Wrapper

- Instantiate generated register block (similar to apb\_hpet.sv)
  - Add apb\_slave\_cdc wrapper if clock domain crossing needed
  - Wire register outputs to STREAM control signals
- 

## Register Map (Planned)

### Global Registers

| Address | Name        | Access   | Description                             |
|---------|-------------|----------|-----------------------------------------|
| 0x00    | GLOBAL_CTRL | RW       | Global enable,<br>reset                 |
| 0x04    | GLOBAL_STAT | RO<br>US | Global status,<br>channel<br>idle/error |
| 0x08    | GLOBAL_CONF | RW<br>IG | Global<br>configuration                 |
| 0x0C    | Reserved    | -        | -                                       |

### Channel Registers (8 × 0x10 bytes)

Each channel (0-7) has a 16-byte register block starting at  $0x10 + (\text{channel\_id} \times 0x10)$ :

| Offset | Name        | Access  | Description                                  |
|--------|-------------|---------|----------------------------------------------|
| +0x00  | CHx_CTRL    | WO      | Descriptor<br>address (write<br>to kick off) |
| +0x04  | CHx_STATUS  | RO      | Channel status,<br>idle, error               |
| +0x08  | CHx_RD_BURS | RW<br>T | Read burst<br>length (for<br>engine config)  |
| +0x0C  | CHx_WR_BURS | RW<br>T | Write burst<br>length (for                   |

| Offset | Name | Access | Description    |
|--------|------|--------|----------------|
|        |      |        | engine config) |

**Example:** - Channel 0 registers: 0x10, 0x14, 0x18, 0x1C - Channel 1 registers: 0x20, 0x24, 0x28, 0x2C - Channel 7 registers: 0x80, 0x84, 0x88, 0x8C

---

## Integration Pattern

### PeakRDL-Generated Implementation (Future):

```
// rtl/stream_macro/app_config.sv
module app_config (
    // APB interface
    input logic [ADDR_WIDTH-1:0] paddr,
    // ...
);
    // Instantiate PeakRDL-generated registers
    stream_regs u_regs (
        .pclk      (pclk),
        .presetn   (presetn),
        .paddr     (paddr),
        .psel      (psel),
        // ... APB signals

        // Register field outputs
        .global_ctrl_enable (ch_enable_internal),
        .ch0_ctrl_desc_addr (ch_desc_addr[0]),
        .ch0_rd_burst       (ch_read_burst_len[0]),
        // ... generated field outputs
    );
    // Optional: Add CDC wrapper if crossing clock domains
    // (like HPET's apb_slave_cdc)
endmodule
```

---

## Reference Examples

### HPET PeakRDL Implementation: -

projects/components/apb\_hpét/peakrdl/hpet\_regs.rdl - Register definition -  
 projects/components/apb\_hpét/peakrdl/generated/ - Generated outputs -  
 projects/components/apb\_hpét/rtl/apb\_hpét.sv - Wrapper with CDC

### HPET Generation Command:

```
.../bin/peakrdl_generate.py hpet_regs.rdl --copy-rtl ..
```

---

## Status

-  **Phase 1:** Manual apb\_config.sv (placeholder)
-  **Phase 2:** Create stream\_regs.rdl (deferred to after Phase 2 RTL implementation)
-  **Phase 3:** Generate register RTL with PeakRDL
-  **Phase 4:** Update apb\_config.sv wrapper to use generated registers

**Note:** Per user feedback, “The apb config will be the last thing done. They will have configs done along the lines of how they are done for the hpet.”

---

## Last Updated: 2025-10-17 Related Documentation: -

[.../docs/ARCHITECTURAL\\_NOTES.md](#) Section 6 - APB Configuration (Deferred) -

[.../PRD.md](#) Section 3.1 - APB Configuration Interface -

[.../apb\\_hpet/peakrdl/README.md](#) - HPET PeakRDL example