

Table of Contents

Stream Index

Generated: 2025-11-23

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
Total	-	-	~9,500 LUTs + 64KB

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

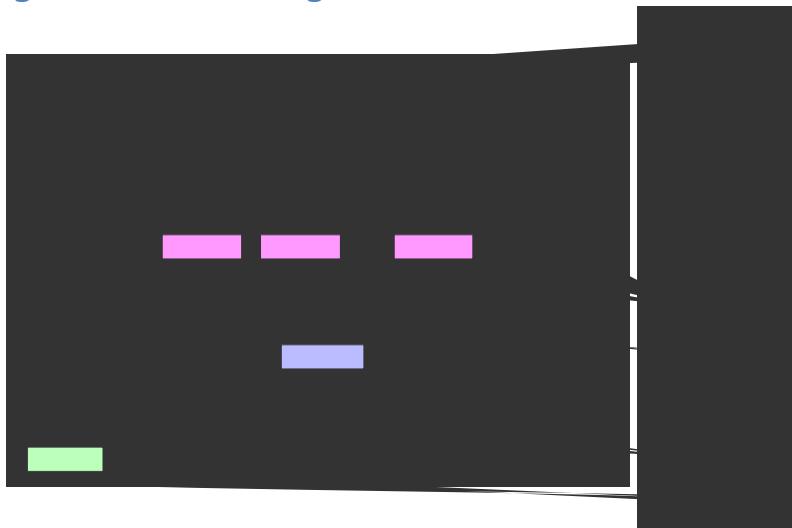
STREAM is a high-performance, multi-channel descriptor-based DMA engine designed for memory-to-memory scatter-gather transfers. It provides an educational yet production-capable architecture demonstrating key DMA concepts while maintaining professional design practices.

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

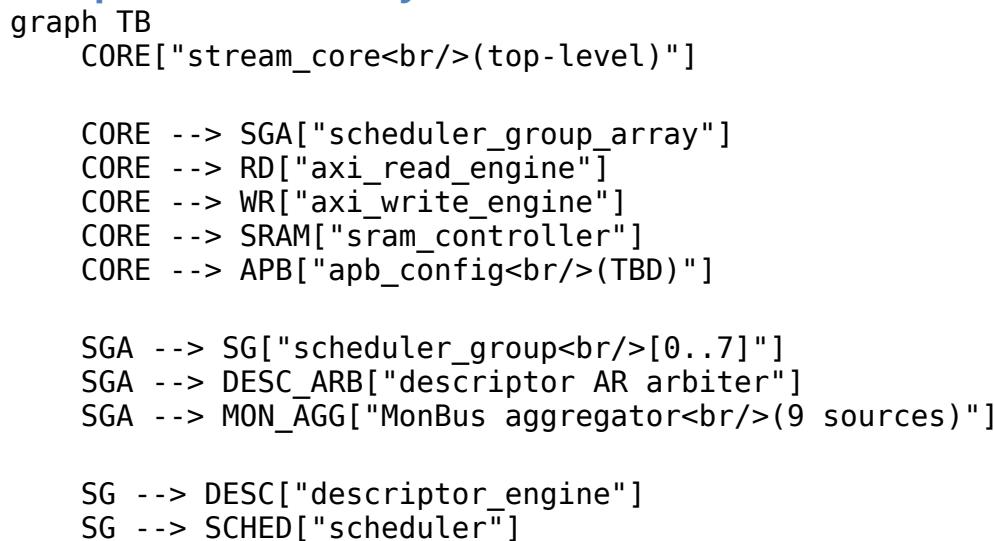
SCHED->>SCHED: Both counters reach 0
SCHED->>DESC: Check next_descriptor_ptr
alt More descriptors
 DESC->>MEM: Fetch next descriptor
else Chain complete
 SCHED->>APB: Transfer complete IRQ
end

```

→

---

## Component Hierarchy



```

RD --> AR_MGR["AR channel manager"]
RD --> R_RX["R channel receiver"]
RD --> RD_ARB["Space-aware arbiter"]

WR --> AW_MGR["AW channel manager"]
WR --> W_STR["W channel streaming"]
WR --> B_TRK["B response tracker"]

SRAM --> SCU["sram_controller_unit
[0..7]"]

SCU --> ALLOC["stream_alloc_ctrl"]
SCU --> FIFO["gaxi_fifo_sync"]
SCU --> BRIDGE["stream_latency_bridge"]
SCU --> DRAIN["stream_drain_ctrl"]

style CORE fill:#e1f5ff,stroke:#333,stroke-width:3px
style SGA fill:#ffe1e1,stroke:#333,stroke-width:2px
style RD fill:#e1ffe1,stroke:#333,stroke-width:2px
style WR fill:#fffe1,stroke:#333,stroke-width:2px
style SRAM fill:#ffffe1,stroke:#333,stroke-width:2px
style APB fill:#f0f0f0,stroke:#333,stroke-width:1px,stroke-
dasharray: 5 5

```

---

## Interface Summary

### External Interfaces

| Interface      | Type   | Width   | Purpose                        |
|----------------|--------|---------|--------------------------------|
| APB            | Slave  | 32-bit  | Configuration, control, status |
| 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         |
| MonBus         | Output | 64-bit  | Debug/trace event stream       |
| IRQ            | Output | 1-bit   | Transfer completion            |

| Interface | Type | Width | Purpose    |
|-----------|------|-------|------------|
|           |      |       | interrupts |

### Configuration Registers (APB)

**Global Registers:** - STREAM\_CTRL - Global enable/disable - STREAM\_STATUS - Overall status - STREAM\_IRQ\_STATUS - Interrupt status (per-channel) - STREAM\_IRQ\_ENABLE - Interrupt enable mask

**Per-Channel Registers (CH0-CH7):** - CHx\_CTRL - Channel enable, kickoff descriptor address - CHx\_STATUS - Channel state, error flags - CHx\_CONFIG - Burst sizes, timeouts, priority - CHx\_CURRENT\_DESC - Current descriptor address (RO) - CHx\_BEATS\_READ - Read beat counter (RO) - CHx\_BEATS\_WRITE - Write beat counter (RO)

---

## 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 × 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:** 2025-11-22 **Document Version:** 0.90

## STREAM Top-Level Port List

**Module:** stream\_core.sv **Location:** projects/components/stream/rtl/macro/

**Last Updated:** 2025-11-22

---

### 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.

**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                              |
|--------|-----------|-------|------------------------------------------|
| clk    | input     | 1     | System clock<br>(100-500 MHz<br>typical) |
| rst_n  | 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                                     |
|---------------|-----------|------------------------------------|-------------------------------------------------|
| apb_valid[ch] | input     | NUM_CHAN<br>NELS                   | Channel descriptor address valid                |
| apb_ready[ch] | output    | NUM_CHAN<br>NELS                   | Channel ready to accept descriptor address      |
| apb_addr[ch]  | input     | NUM_CHAN<br>NELS × ADDR_WIDTH<br>H | Descriptor address per 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 (0=disabled, 1=enabled) |
| cfg_channel_reset[ch]  | input     | NUM_CHAN<br>NELS | Soft reset channel (FSM → 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 cycles)   |
| cfg_sched_timeout_enable        | input     | 1     | Enable timeout detection           |
| cfg_sched_error_enable          | input     | 1     | Enable error event reporting       |
| cfg_sched_completion_enable     | input     | 1     | Enable completion event reporting  |
| cfg_sched_performance_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               |

| Signal Suffix | Width | Description            |
|---------------|-------|------------------------|
| _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_x_fer_beats | input     | 8     | Read burst size (beats, 1-256)  |
| cfg_axi_wr_x_fer_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 complete for channel          |
| axi_wr_all_complete[ch]    | output    | NUM_CHAN<br>NELS     | All write transactions complete for channel         |

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

```
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 descriptor
7'b0100000 // CH_ERROR - Error occurred
7'b1000000 // (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 (0-256)             |
| perf_fifo_rd        | input     | 1     | Read profiler entry (pop FIFO)              |
| perf_fifo_data_low  | output    | 32    | Profiler data [31:0] (timestamp or elapsed) |
| perf_fifo_data_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                                  |
| m_axi_desc_ar_addr   | output    | ADDR_WIDTH   | Read address                                    |
| m_axi_desc_ar_len    | output    | 8            | Burst length - 1<br>(AXI encoding)              |
| m_axi_desc_ar_size   | output    | 3            | Burst size =<br>$\log_2(\text{bytes per beat})$ |
| m_axi_desc_ar_burst  | output    | 2            | Burst type<br>(01=INCR)                         |
| m_axi_desc_ar_lock   | output    | 1            | Lock type<br>(0=normal)                         |
| m_axi_desc_ar_cache  | output    | 4            | Cache<br>attributes                             |
| m_axi_desc_ar_prot   | output    | 3            | Protection<br>attributes                        |
| m_axi_desc_ar_qos    | output    | 4            | QoS value                                       |
| m_axi_desc_ar_region | output    | 4            | Region<br>identifier                            |
| m_axi_desc_ar_user   | output    | CHAN_WIDTH   | User signal<br>(channel ID)                     |
| m_axi_desc_ar_valid  | output    | 1            | Address valid                                   |
| m_axi_desc_ar_ready  | input     | 1            | Address ready                                   |

### R Channel (Read Data)

| Signal            | Direction | Width            | Description                                 |
|-------------------|-----------|------------------|---------------------------------------------|
| m_axi_desc_r_id   | input     | AXI_ID_WID<br>TH | Transaction ID                              |
| m_axi_desc_r_data | input     | 256              | Read data (FIXED 256-bit descriptor)        |
| m_axi_desc_r_resp | input     | 2                | Response (00=OKAY,<br>01=EXOKAY, 10=SLVERR, |

| Signal                        | Direction | Width      | Description              |
|-------------------------------|-----------|------------|--------------------------|
|                               |           |            | 11=DECERR)               |
| m_axi_desc_r <sub>last</sub>  | input     | 1          | Last beat of burst       |
| m_axi_desc_r <sub>user</sub>  | input     | CHAN_WIDTH | User signal (channel ID) |
| m_axi_desc_r <sub>valid</sub> | input     | 1          | Read data valid          |
| m_axi_desc_r <sub>ready</sub> | 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 <sub>dr</sub>  | output    | ADDR_WIDTH   | Read address                      |
| m_axi_rd_arle <sub>n</sub>   | output    | 8            | Burst length - 1                  |
| m_axi_rd_arsi <sub>ze</sub>  | output    | 3            | Burst size = log2(bytes per beat) |
| m_axi_rd_arbu <sub>rst</sub> | output    | 2            | Burst type (01=INCR)              |
| m_axi_rd_arlo <sub>ck</sub>  | output    | 1            | Lock type (0=normal)              |
| m_axi_rd_arca <sub>che</sub> | output    | 4            | Cache attributes                  |
| m_axi_rd_arpr <sub>ot</sub>  | output    | 3            | Protection attributes             |

| Signal            | Direction | Width      | Description              |
|-------------------|-----------|------------|--------------------------|
| m_axi_rd_arqs     | output    | 4          | QoS value                |
| m_axi_rd_arregion | output    | 4          | Region 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_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             |

**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 (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: 2025-11-22 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

---

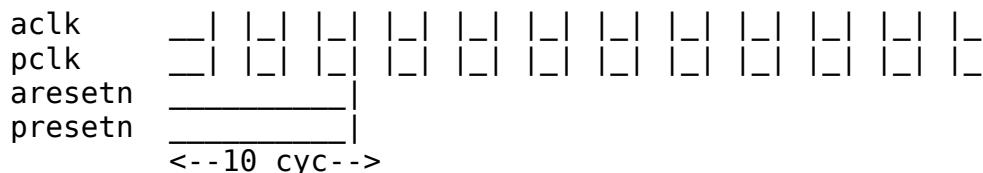
## Reset Sequencing

### 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:** 2025-11-21

---

## Overview

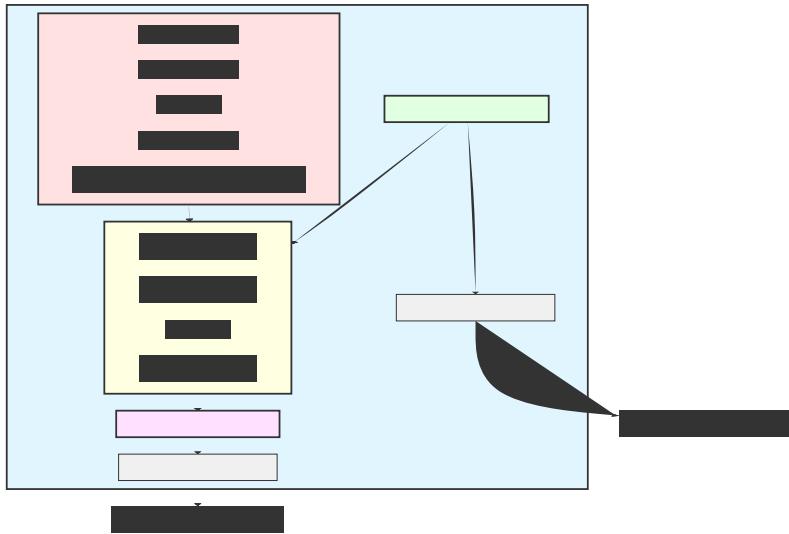
The STREAM Core is the top-level integration module that combines all STREAM components into a complete scatter-gather DMA engine. It provides 8 independent channels with descriptor-based memory-to-memory transfers.

## 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                       |

### 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 closure  |
| SKID_DEPTH_AW | 2       | 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<br>NELS                      | Channel ready to accept descriptor address |
| apb_addr[ch]  | input     | NUM_CHAN<br>NELS ×<br>ADDR_WIDTH<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_enable      | input     | 1     | Enable error event reporting       |
| cfg_sched_completion_enable | input     | 1     | Enable completion event reporting  |
| cfg_sched_perf_enable       | 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_WIDTH_H | Base address limit 0        |

| Signal                  | Direction | Width          | Description           |
|-------------------------|-----------|----------------|-----------------------|
| cfg_desceng_addr0_limit | input     | ADDR_WIDT<br>H | Limit address limit 0 |
| cfg_desceng_addr1_base  | input     | ADDR_WIDT<br>H | Base address limit 1  |
| cfg_desceng_addr1_limit | input     | ADDR_WIDT<br>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 |
| _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           |

| Signal Suffix | Width | Description            |
|---------------|-------|------------------------|
| _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

#### Per-Channel Status:

| Signal                     | Direction | Width         | Description             |
|----------------------------|-----------|---------------|-------------------------|
| descriptor_engine_idle[ch] | output    | NUM_CHAN NELS | Descriptor engine idle  |
| scheduler_idle[ch]         | output    | NUM_CHAN NELS | Scheduler in IDLE state |
| scheduler_st               | output    | NUM_CHAN      | Scheduler FSM state     |

| Signal                   | Direction | Width            | Description                     |
|--------------------------|-----------|------------------|---------------------------------|
| ate[ch]                  |           | NELS × 7         | (ONE-HOT)                       |
| sched_error[ch]          | output    | NUM_CHAN<br>NELS | Scheduler error (sticky)        |
| axi_rd_all_c_omplete[ch] | output    | NUM_CHAN<br>NELS | All read transactions complete  |
| axi_wr_all_c_omplete[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_high | output    | 32    | Profiler data [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_type | output    | 2            | Burst type              |

| Signal               | Direction | Width      | Description              |
|----------------------|-----------|------------|--------------------------|
| burst                |           |            | (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                       |
| 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_WIDT H   | 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<br>dr   | output    | ADDR_WIDTH   | Address                  |
| m_axi_rd_arle<br>n    | output    | 8            | Burst length - 1         |
| m_axi_rd_arsi<br>ze   | output    | 3            | Burst size (log2 bytes)  |
| m_axi_rd_arbu<br>rst  | output    | 2            | Burst type (INCR)        |
| m_axi_rd_arlo<br>ck   | output    | 1            | Lock type                |
| m_axi_rd_arca<br>che  | output    | 4            | Cache attributes         |
| m_axi_rd_arpr<br>ot   | output    | 3            | Protection attributes    |
| m_axi_rd_arqo<br>s    | output    | 4            | QoS value                |
| m_axi_rd_arre<br>gion | output    | 4            | Region identifier        |
| m_axi_rd_arus<br>er   | output    | CHAN_WIDTH   | User signal (channel ID) |
| m_axi_rd_arva<br>lid  | output    | 1            | Address valid            |
| m_axi_rd_arre<br>ady  | input     | 1            | Address ready            |

### R Channel:

| Signal             | Direction | Width        | Description                 |
|--------------------|-----------|--------------|-----------------------------|
| m_axi_rd_rid       | input     | AXI_ID_WIDTH | Transaction ID              |
| m_axi_rd_rdat<br>a | input     | DATA_WIDTH   | Read data (default 512-bit) |
| m_axi_rd_rres<br>p | input     | 2            | Response                    |

| Signal           | Direction | Width      | Description              |
|------------------|-----------|------------|--------------------------|
| m_axi_rd_rlas_t  | input     | 1          | Last beat of burst       |
| m_axi_rd_ruse_r  | input     | CHAN_WIDTH | User signal (channel ID) |
| m_axi_rd_rval_id | input     | 1          | Read data valid          |
| m_axi_rd_rrea_dy | 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_awad_dr    | output    | ADDR_WIDTH   | Address                  |
| m_axi_wr_awle_n     | output    | 8            | Burst length - 1         |
| m_axi_wr_awsiz_e    | output    | 3            | Burst size (log2 bytes)  |
| m_axi_wr_awbu_rst   | output    | 2            | Burst type (INCR)        |
| m_axi_wr_awlo_ck    | output    | 1            | Lock type                |
| m_axi_wr_awca_ch_e  | output    | 4            | Cache attributes         |
| m_axi_wr_awpr_ot    | output    | 3            | Protection attributes    |
| m_axi_wr_awqos      | output    | 4            | QoS value                |
| m_axi_wr_awregion   | output    | 4            | Region identifier        |
| m_axi_wr_awuser     | output    | CHAN_WIDTH   | User signal (channel ID) |
| m_axi_wr_awvalid_id | output    | 1            | Address valid            |

| Signal                       | Direction | Width | Description   |
|------------------------------|-----------|-------|---------------|
| m_axi_wr_awre <sub>ady</sub> | input     | 1     | Address ready |

### W Channel:

| Signal                      | Direction | Width        | Description                  |
|-----------------------------|-----------|--------------|------------------------------|
| m_axi_wr_wda <sub>ta</sub>  | output    | DATA_WIDTH   | Write data (default 512-bit) |
| m_axi_wr_wst <sub>rb</sub>  | output    | DATA_WIDTH/8 | Write strobes (byte enables) |
| m_axi_wr_wla <sub>st</sub>  | output    | 1            | Last beat of burst           |
| m_axi_wr_wus <sub>er</sub>  | output    | CHAN_WIDTH   | User signal (channel ID)     |
| m_axi_wr_wva <sub>lid</sub> | output    | 1            | Write data valid             |
| m_axi_wr_wre <sub>ady</sub> | 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 <sub>p</sub>  | input     | 2            | Response                 |
| m_axi_wr_buse <sub>r</sub>  | input     | CHAN_WIDTH   | User signal (channel ID) |
| m_axi_wr_bval <sub>id</sub> | input     | 1            | Response valid           |
| m_axi_wr_brea <sub>dy</sub> | output    | 1            | Response ready           |

### Status/Debug Outputs

#### Descriptor AXI Monitor:

| Signal                               | Direction | Width | Description              |
|--------------------------------------|-----------|-------|--------------------------|
| cfg_sts_desc <sub>_mon_busy</sub>    | output    | 1     | Monitor busy             |
| cfg_sts_desc <sub>_mon_active_</sub> | output    | 8     | Active transaction count |

| Signal                          | Direction | Width | Description             |
|---------------------------------|-----------|-------|-------------------------|
| txns                            |           |       |                         |
| cfg_sts_desc_mon_error_count    | output    | 16    | Error count             |
| cfg_sts_desc_mon_txn_count      | output    | 32    | Total transaction count |
| cfg_sts_desc_mon_conflict_error | output    | 1     | ID conflict detected    |

#### Read Engine AXI Monitor:

| Signal                            | Direction | Width | Description              |
|-----------------------------------|-----------|-------|--------------------------|
| cfg_sts_rden_g_skid_busy          | output    | 1     | Skid buffer busy         |
| cfg_sts_rden_g_mon_active_txns    | output    | 8     | Active transaction count |
| cfg_sts_rden_g_mon_error_count    | output    | 16    | 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     |

#### Write Engine AXI Monitor:

| Signal                            | Direction | Width | Description              |
|-----------------------------------|-----------|-------|--------------------------|
| cfg_sts_wren_g_skid_busy          | output    | 1     | Skid buffer busy         |
| cfg_sts_wren_g_mon_active_txns    | output    | 8     | Active transaction count |
| cfg_sts_wren_g_mon_error_count    | output    | 16    | 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     |

| Signal               | Direction | Width | Description |
|----------------------|-----------|-------|-------------|
| 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/macro/

### 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:** 12\_scheduler\_group\_array.md - Multi-channel scheduler integration
  - **Scheduler Group:** 11\_scheduler\_group.md - Single channel scheduler + descriptor engine
  - **SRAM Controller:** sram\_controller.md - Per-channel buffering
  - **AXI Read Engine:** 08\_axi\_read\_engine.md - Read datapath
  - **AXI Write Engine:** 10\_axi\_write\_engine.md - Write datapath
  - **Performance Profiler:** 05\_perf\_profiler.md - Performance monitoring
- 

**Last Updated:** 2025-11-21 (verified against RTL implementation)

## Scheduler Group Array Specification

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

projects/components/stream/rtl/macro/ **Status:** Implemented

---

## Overview

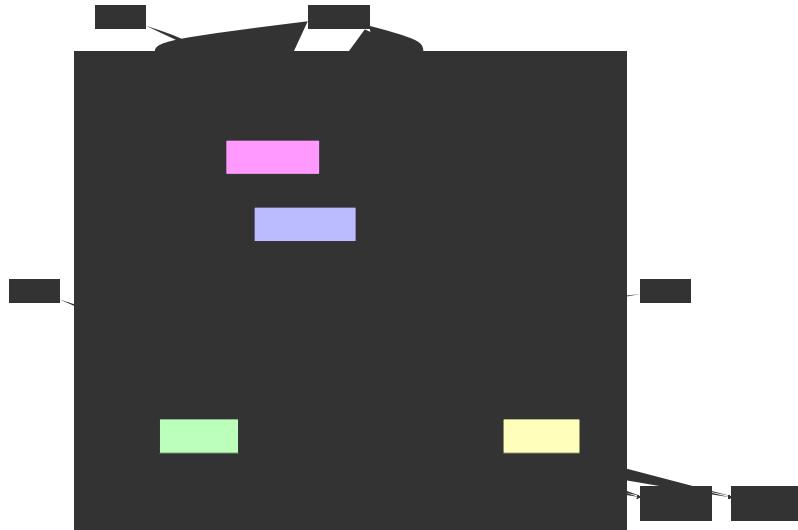
The Scheduler Group Array is the top-level multi-channel DMA control structure, instantiating 8 independent scheduler\_group instances with shared resource management. It provides descriptor fetch arbitration and unified MonBus aggregation.

## Key Features

- **8 Independent Channels:** Each with descriptor engine + scheduler
  - **Shared Descriptor AXI Master:** Round-robin AR arbitration across channels
  - **AXI Master Monitor:** Monitors descriptor fetch transactions for debug/performance
  - **Direct Data Paths:** Read/write interfaces pass through (engines arbitrate, not array)
  - **9-Source MonBus Aggregation:** 8 channels + descriptor AXI monitor
  - **Global Configuration:** Common config distributed to all channels
- 

## Architecture

### Block Diagram



*Diagram*

→

---

## Parameters

```
parameter int NUM_CHANNELS = 8; // Number of channels
(fixed at 8)
parameter int CHAN_WIDTH = $clog2(NUM_CHANNELS); // Channel ID width
(3 bits)
parameter int ADDR_WIDTH = 64; // Address bus width
parameter int DATA_WIDTH = 512; // Data bus width
parameter int AXI_ID_WIDTH = 8; // AXI ID width
parameter int TIMEOUT_CYCLES = 1000; // Scheduler timeout
threshold

// Monitor Bus Base IDs
parameter DESC_MON_BASE_AGENT_ID = 16; // 0x10 - Descriptor
Engines (16-23)
parameter SCHED_MON_BASE_AGENT_ID = 48; // 0x30 - Schedulers
(48-55)
parameter DESC_AXI_MON_AGENT_ID = 8; // 0x08 - Descriptor
AXI Master Monitor
parameter MON_UNIT_ID = 1; // 0x1
```

## MonBus Agent ID Mapping:

Channel 0: Descriptor Engine = 16 (0x10), Scheduler = 48 (0x30)  
Channel 1: Descriptor Engine = 17 (0x11), Scheduler = 49 (0x31)  
...  
Channel 7: Descriptor Engine = 23 (0x17), Scheduler = 55 (0x37)  
Desc AXI Monitor: 8 (0x08)

---

## Port List

### Clock and Reset

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

### APB Programming Interface (Per-Channel)

| Signal    | Direction | Width            | Description           |
|-----------|-----------|------------------|-----------------------|
| apb_valid | input     | NUM_CHAN<br>NELS | Per-channel APB valid |
| apb_ready | output    | NUM_CHAN<br>NELS | Per-channel APB ready |

| Signal   | Direction | Width                                | Description                              |
|----------|-----------|--------------------------------------|------------------------------------------|
| apb_addr | input     | NUM_CHAN<br>NELS ×<br>ADDR_WIDT<br>H | Initial descriptor addresses per channel |

## Configuration Interface

### Per-Channel Control:

| Signal             | Direction | Width        | Description            |
|--------------------|-----------|--------------|------------------------|
| cfg_channel_enable | input     | NUM_CHANNELS | Per-channel enable     |
| cfg_channel_reset  | input     | NUM_CHANNELS | Per-channel soft reset |

### Global Scheduler Configuration:

| Signal                        | Direction | Width | Description                 |
|-------------------------------|-----------|-------|-----------------------------|
| cfg_sched_enable              | input     | 1     | Master scheduler enable     |
| cfg_sched_timeout_eout_cycles | input     | 16    | Timeout threshold (future)  |
| cfg_sched_timeout_eout_enable | input     | 1     | Timeout enable (future)     |
| cfg_sched_error_enable        | input     | 1     | Error reporting (future)    |
| cfg_sched_completion_enable   | input     | 1     | Completion events (future)  |
| cfg_sched_performance_enable  | input     | 1     | Performance events (future) |

### Global Descriptor Engine Configuration:

| Signal             | Direction | Width | Description              |
|--------------------|-----------|-------|--------------------------|
| cfg_desceng_enable | input     | 1     | Master descriptor engine |

| Signal                  | Direction | Width          | Description           |
|-------------------------|-----------|----------------|-----------------------|
|                         |           |                | enable                |
| cfg_desceng_prefetch    | input     | 1              | Enable prefetch       |
| cfg_desceng_fifo_thresh | input     | 4              | FIFO threshold        |
| cfg_desceng_addr0_base  | input     | ADDR_WIDT<br>H | Address range 0 base  |
| cfg_desceng_addr0_limit | input     | ADDR_WIDT<br>H | Address range 0 limit |
| cfg_desceng_addr1_base  | input     | ADDR_WIDT<br>H | Address range 1 base  |
| cfg_desceng_addr1_limit | input     | ADDR_WIDT<br>H | Address range 1 limit |

### Descriptor AXI Monitor Configuration:

| Signal                     | Direction | Width | Description                |
|----------------------------|-----------|-------|----------------------------|
| cfg_daxmon_er_r_enable     | input     | 1     | Enable error detection     |
| cfg_daxmon_co_mpl_enable   | input     | 1     | Enable completion packets  |
| cfg_daxmon_ti_meout_enable | input     | 1     | Enable timeout detection   |
| cfg_daxmon_pe_rf_enable    | input     | 1     | Enable performance packets |
| cfg_daxmon_de_bug_enable   | input     | 1     | Enable debug packets       |

### Status Interface

#### Per-Channel Status:

| Signal                 | Direction | Width            | Description                        |
|------------------------|-----------|------------------|------------------------------------|
| descriptor_engine_idle | output    | NUM_CHAN<br>NELS | Per-channel descriptor engine idle |

| Signal          | Direction | Width             | Description                           |
|-----------------|-----------|-------------------|---------------------------------------|
| scheduler_id_le | output    | NUM_CHAN_NELS     | Per-channel scheduler idle            |
| scheduler_state | output    | NUM_CHAN_NELS × 7 | Per-channel scheduler state (ONE-HOT) |

### Descriptor AXI Monitor Status:

| Signal                      | Direction | Width | Description                 |
|-----------------------------|-----------|-------|-----------------------------|
| desc_axi_mon_busy           | output    | 1     | Descriptor AXI monitor busy |
| desc_axi_mon_active_txns    | output    | 8     | Active transaction count    |
| desc_axi_mon_error_count    | output    | 16    | Error count                 |
| desc_axi_mon_txn_count      | output    | 32    | Transaction count           |
| desc_axi_mon_conflict_error | output    | 1     | Conflict error flag         |

### Shared Descriptor AXI4 Master Read Interface

#### AR Channel (Arbitrated Output):

| Signal           | Direction | Width      | Description                       |
|------------------|-----------|------------|-----------------------------------|
| desc_axi_arvalid | output    | 1          | Address read valid                |
| desc_axi_arready | input     | 1          | Address read ready                |
| desc_axi_araddr  | output    | ADDR_WIDTH | Read address                      |
| desc_axi_arlen   | output    | 8          | Burst length - 1                  |
| desc_axi_arsize  | output    | 3          | Burst size (log2(bytes per beat)) |
| desc_axi_arburst | output    | 2          | Burst type (2'b01 = INCR)         |

| Signal             | Direction | Width        | Description           |
|--------------------|-----------|--------------|-----------------------|
| desc_axi_arid      | output    | AXI_ID_WIDTH | Transaction ID        |
| desc_axi_arlo ck   | output    | 1            | Lock signal           |
| desc_axi_arca che  | output    | 4            | Cache attributes      |
| desc_axi_arpr ot   | output    | 3            | Protection attributes |
| desc_axi_arqs      | output    | 4            | Quality of Service    |
| desc_axi_arre gion | output    | 4            | Region identifier     |

### R Channel (Broadcast to All Channels):

| Signal           | Direction | Width        | Description                     |
|------------------|-----------|--------------|---------------------------------|
| desc_axi_rval id | input     | 1            | Read data valid                 |
| desc_axi_rrea dy | output    | 1            | Read data ready                 |
| desc_axi_rdat a  | input     | 256          | Descriptor data (FIXED 256-bit) |
| desc_axi_rres p  | input     | 2            | Read response (2'b00 = OKAY)    |
| desc_axi_rlas t  | input     | 1            | Last beat in burst              |
| desc_axi_rid     | input     | AXI_ID_WIDTH | Transaction ID                  |

### Shared Data Read Interface (to AXI Read Engine)

#### Per-Channel Arrays (Direct Passthrough):

| Signal       | Direction | Width         | Description                    |
|--------------|-----------|---------------|--------------------------------|
| datard_valid | output    | NUM_CHAN NELS | Per-channel read request valid |
| datard_ready | input     | NUM_CHAN NELS | Per-channel read engine ready  |

| Signal                  | Direction | Width                                | Description                  |
|-------------------------|-----------|--------------------------------------|------------------------------|
| datard_addr             | output    | NUM_CHAN<br>NELS ×<br>ADDR_WIDT<br>H | Per-channel source addresses |
| datard_beats _remaining | output    | NUM_CHAN<br>NELS × 32                | Per-channel beats remaining  |
| datard_done_ strobe     | input     | NUM_CHAN<br>NELS                     | Per-channel read burst done  |
| datard_beats _done      | input     | NUM_CHAN<br>NELS × 32                | Per-channel beats completed  |
| datard_error            | input     | NUM_CHAN<br>NELS                     | Per-channel read error       |

## Shared Data Write Interface (to AXI Write Engine)

### Per-Channel Arrays (Direct Passthrough):

| Signal                  | Direction | Width                                | Description                       |
|-------------------------|-----------|--------------------------------------|-----------------------------------|
| datawr_valid            | output    | NUM_CHAN<br>NELS                     | Per-channel write request valid   |
| datawr_ready            | input     | NUM_CHAN<br>NELS                     | Per-channel write engine ready    |
| datawr_addr             | output    | NUM_CHAN<br>NELS ×<br>ADDR_WIDT<br>H | Per-channel destination addresses |
| datawr_beats _remaining | output    | NUM_CHAN<br>NELS × 32                | Per-channel beats remaining       |
| datawr_done_ strobe     | input     | NUM_CHAN<br>NELS                     | Per-channel write burst done      |
| datawr_beats _done      | input     | NUM_CHAN<br>NELS × 32                | Per-channel beats completed       |
| datawr_error            | input     | NUM_CHAN<br>NELS                     | Per-channel write error           |

## Unified 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
```

## APB Programming Interface (Per-Channel)

### Descriptor Kickoff:

```
input logic [NUM_CHANNELS-1:0] apb_valid;
output logic [NUM_CHANNELS-1:0] apb_ready;
input logic [NUM_CHANNELS-1:0][ADDR_WIDTH-1:0] apb_addr; // Initial
descriptor addresses
```

**Per-Channel Independent:** - Each channel has separate APB interface - Software writes descriptor address for channel N - Channel N descriptor engine begins fetching

## Configuration Interface

### Per-Channel Control:

```
input logic [NUM_CHANNELS-1:0] cfg_channel_enable;
// Per-channel enable
input logic [NUM_CHANNELS-1:0] cfg_channel_reset;
// Per-channel soft reset
```

### Global Scheduler Configuration:

```
input logic cfg_sched_enable;
// Master enable
input logic [15:0] cfg_sched_timeout_cycles;
// Timeout (future)
input logic cfg_sched_timeout_enable;
```

```

// Timeout enable (future)

// Error reporting (future)

// Completion events (future)

// Performance events (future)

cfg_sched_err_enable;
cfg_sched_compl_enable;
cfg_sched_perf_enable;

```

### Global Descriptor Engine Configuration:

```


// Master enable

// Enable prefetch
 [3:0]
// FIFO threshold
 [ADDR_WIDTH-1:0]
// Address range 0 base
 [ADDR_WIDTH-1:0]
// Address range 0 limit
 [ADDR_WIDTH-1:0]
// Address range 1 base
 [ADDR_WIDTH-1:0]
// Address range 1 limit

cfg_desceng_enable;
cfg_desceng_prefetch;
cfg_desceng_fifo_thresh;
cfg_desceng_addr0_base;
cfg_desceng_addr0_limit;
cfg_desceng_addr1_base;
cfg_desceng_addr1_limit;

```

### Descriptor AXI Monitor Configuration:

```


// Enable error detection

// Enable completion packets

cfg_daxmon_timeout_enable; // Enable timeout detection

// Enable performance packets

// Enable debug packets

cfg_daxmon_err_enable;
cfg_daxmon_compl_enable;
cfg_daxmon_perf_enable;
cfg_daxmon_debug_enable;

```

**Global vs. Per-Channel:** - cfg\_channel\_enable/reset: **Per-channel** (independent control) - cfg\_sched\_, cfg\_desceng\_: **Global** (broadcast to all channels) - cfg\_daxmon\_\*: **Global** (single monitor instance)

### Status Interface

#### Per-Channel Status:

```

output logic [NUM_CHANNELS-1:0]
output logic [NUM_CHANNELS-1:0]
output logic [NUM_CHANNELS-1:0][6:0] descriptor_engine_idle;
 scheduler_idle;
 scheduler_state; // ONE-
HOT encoding
 // ONE-

```

### Descriptor AXI Monitor Status:

```
output logic desc_axi_mon_busy;
output logic [7:0] desc_axi_mon_active_txns;
output logic [15:0] desc_axi_mon_error_count;
output logic [31:0] desc_axi_mon_txn_count;
output logic desc_axi_mon_conflict_error;
```

### Shared Descriptor AXI4 Master Read Interface

#### AR Channel (Arbitrated Output):

```
output logic desc_axi_arvalid;
input logic desc_axi_arready;
output logic [ADDR_WIDTH-1:0] desc_axi_araddr;
output logic [7:0] desc_axi_arlen;
output logic [2:0] desc_axi_arsize;
output logic [1:0] desc_axi_arburst;
output logic [AXI_ID_WIDTH-1:0] desc_axi_arid;
output logic desc_axi_arlock;
output logic [3:0] desc_axi_arcache;
output logic [2:0] desc_axi_arprot;
output logic [3:0] desc_axi_arqos;
output logic [3:0] desc_axi_arregion;
```

#### R Channel (Broadcast to All Channels):

```
input logic desc_axi_rvalid;
output logic desc_axi_rready;
input logic [255:0] desc_axi_rdata; // FIXED
256-bit descriptor
input logic [1:0] desc_axi_rrresp;
input logic desc_axi_rlast;
input logic [AXI_ID_WIDTH-1:0] desc_axi_rid;
```

**Arbitration:** - AR channel: Round-robin arbiter selects one channel at a time - R channel: Broadcast to all channels, each checks RID to claim data

### Shared Data Read Interface (to AXI Read Engine)

#### Per-Channel Arrays (Direct Passthrough):

```
output logic [NUM_CHANNELS-1:0] datard_valid;
input logic [NUM_CHANNELS-1:0] datard_ready;
output logic [NUM_CHANNELS-1:0][ADDR_WIDTH-1:0] datard_addr;
output logic [NUM_CHANNELS-1:0][31:0] datard_beats_remaining;
```

**NO ARBITRATION:** - All 8 channels connect directly to read engine - Read engine performs arbitration internally

### Shared Data Write Interface (to AXI Write Engine)

#### Per-Channel Arrays (Direct Passthrough):

```
output logic [NUM_CHANNELS-1:0] datawr_valid;
input logic [NUM_CHANNELS-1:0] datawr_ready;
output logic [NUM_CHANNELS-1:0][ADDR_WIDTH-1:0] datawr_addr;
output logic [NUM_CHANNELS-1:0][31:0] datawr_beats_remaining;
```

**NO ARBITRATION:** - All 8 channels connect directly to write engine - Write engine performs arbitration internally

### Data Path Completion Strobes

#### Per-Channel Feedback (Direct Passthrough):

```
input logic [NUM_CHANNELS-1:0] datard_done_strobe;
input logic [NUM_CHANNELS-1:0][31:0] datard_beats_done;
input logic [NUM_CHANNELS-1:0] datawr_done_strobe;
input logic [NUM_CHANNELS-1:0][31:0] datawr_beats_done;
```

### Error Signals

#### Per-Channel Errors (Direct Passthrough):

```
input logic [NUM_CHANNELS-1:0] datard_error;
input logic [NUM_CHANNELS-1:0] datawr_error;
```

### Unified Monitor Bus Interface

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

**Aggregation:** - 9 sources:  $8 \times \text{scheduler\_group} + 1 \times \text{desc\_axi\_monitor}$  - Round-robin arbitration - Input skid buffers (depth 2) - Output skid buffer (depth 2)

---

## Internal Architecture

### Scheduler Group Instantiation

#### 8 Independent Instances:

```

generate
 for (genvar ch = 0; ch < NUM_CHANNELS; ch++) begin :
gen_scheduler_groups
 scheduler_group #(
 .CHANNEL_ID (ch),
 .NUM_CHANNELS (NUM_CHANNELS),
 .DESC_MON_AGENT_ID (8'(DESC_MON_BASE_AGENT_ID + ch)),
// 16-23
 .SCHED_MON_AGENT_ID (8'(SCHED_MON_BASE_AGENT_ID +
ch)), // 48-55
 .MON_CHANNEL_ID (6'(ch))
) u_scheduler_group (
 // ... per-channel connections ...
);
end
endgenerate

```

**Per-Channel Unique:** - CHANNEL\_ID: 0-7 - DESC\_MON\_AGENT\_ID: 16-23 (BASE + ch) - SCHED\_MON\_AGENT\_ID: 48-55 (BASE + ch) - MON\_CHANNEL\_ID: 0-7

## Descriptor AXI AR Arbitration

### Round-Robin Arbiter:

```

arbiter_round_robin #(
 .CLIENTS (NUM_CHANNELS),
 .WAIT_GNT_ACK (1) // Wait for AR handshake
) u_desc_ar_arbiter (
 .request (desc_ar_valid), // 8 channels
requesting
 .grant_ack (desc_ar_grant_ack), // Ack when AR
accepted
 .grant_valid (desc_ar_grant_valid),
 .grant (desc_ar_grant), // One-hot grant
 .grant_id (desc_ar_grant_id) // Binary grant
ID
);

```

### AR Mux:

```

// Select AR signals from granted channel
assign desc_axi_int_araddr = desc_ar_addr[desc_ar_grant_id];
assign desc_axi_int_arlen = desc_ar_len[desc_ar_grant_id];
// ... other AR signals ...

```

### AR Ready Demux:

```

// Broadcast ready to all channels (only granted channel uses it)
for (genvar ch = 0; ch < NUM_CHANNELS; ch++) begin
 assign desc_ar_ready[ch] = (desc_ar_grant[ch] &&

```

```
desc_axi_int_arready);
end
```

## Descriptor AXI R Broadcast

### R Valid Broadcast:

```
// All channels see R valid
assign desc_r_valid = {NUM_CHANNELS{desc_axi_int_rvalid}};
```

### R Data Broadcast:

```
// All channels see same R data
for (genvar ch = 0; ch < NUM_CHANNELS; ch++) begin
 assign desc_r_data[ch] = desc_axi_int_rdata;
 assign desc_r_id[ch] = desc_axi_int_rid;
 assign desc_r_last[ch] = desc_axi_int_rlast;
 assign desc_r_resp[ch] = desc_axi_int_rresp;
end
```

### R Ready OR:

```
// R ready if ANY channel ready (each checks RID to claim)
assign desc_axi_int_rready = |desc_r_ready;
```

**ID-Based Claiming:** - Each descriptor engine checks desc\_r\_id against expected ID - Only matching channel asserts desc\_r\_ready - Allows out-of-order responses (different channels)

## AXI Master Read Monitor

### Monitor Insertion:

```
axi4_master_read_monitor #(
 .ADDR_WIDTH(ADDR_WIDTH),
 .DATA_WIDTH(256), // FIXED for descriptor width
 .ID_WIDTH(AXI_ID_WIDTH)
) u_desc_axi_monitor (
 // AR channel (tap)
 .m_axi_araddr (desc_axi_int_araddr),
 .m_axi_arvalid (desc_axi_int_arvalid),
 .m_axi_arready (desc_axi_int_arready),
 // ... other AR signals ...

 // R channel (tap)
 .m_axi_rdata (desc_axi_int_rdata),
 .m_axi_rvalid (desc_axi_int_rvalid),
 .m_axi_rready (desc_axi_int_rready),
 // ... other R signals ...
```

```

// MonBus output
.mon_valid (desc_axi_mon_valid),
.mon_ready (desc_axi_mon_ready),
.mon_packet (desc_axi_mon_packet),

// Status outputs
.busy (desc_axi_mon_busy),
.active_txns (desc_axi_mon_active_txns),
.error_count (desc_axi_mon_error_count),
.txn_count (desc_axi_mon_txn_count)
);

```

**Pass-Through:** - Monitor does NOT modify AXI signals - Observes transactions and generates MonBus events - Provides visibility into descriptor fetch performance

## MonBus Aggregation

### 9-Source Arbiter:

```

localparam int MONBUS_SOURCES = NUM_CHANNELS + 1; // 8 channels + 1
monitor

monbus_arbiter #(
 .CLIENTS (MONBUS_SOURCES), // 9 sources
 .INPUT_SKID_ENABLE (1),
 .OUTPUT_SKID_ENABLE (1),
 .INPUT_SKID_DEPTH (2),
 .OUTPUT_SKID_DEPTH (2)
) u_monbus_aggregator (
 .monbus_valid_in (monbus_valid_all), // [9] array
 .monbus_ready_in (monbus_ready_all),
 .monbus_packet_in (monbus_packet_all),
 .monbus_valid (mon_valid),
 .monbus_ready (mon_ready),
 .monbus_packet (mon_packet)
);

```

### Source Mapping:

```

// Channels 0-7
for (genvar ch = 0; ch < NUM_CHANNELS; ch++) begin
 assign monbus_valid_all[ch] = mon_valid_ch[ch];
 assign monbus_packet_all[ch] = mon_packet_ch[ch];
 assign mon_ready_ch[ch] = monbus_ready_all[ch];
end

// Descriptor AXI monitor (source 8)
assign monbus_valid_all[NUM_CHANNELS] = desc_axi_mon_valid;

```

```
assign monbus_packet_all[NUM_CHANNELS] = desc_axi_mon_packet;
assign desc_axi_mon_ready =
monbus_ready_all[NUM_CHANNELS];
```

---

## Operation Flows

### Multi-Channel Concurrent Operation

**Scenario:** Channels 0, 2, 5 active simultaneously

#### Descriptor Fetch:

Cycle N: CH0 requests AR (desc\_ar\_valid[0] = 1)  
CH2 requests AR (desc\_ar\_valid[2] = 1)  
Arbiter grants CH0 → desc\_axi\_arvalid = 1, araddr = ch0\_addr

Cycle N+1: AR accepted → CH0 waits for R data  
Arbiter grants CH2 → desc\_axi\_arvalid = 1, araddr = ch2\_addr

Cycle N+5: R data arrives, RID = 0 → CH0 claims (desc\_r\_ready[0] = 1)

Cycle N+7: R data arrives, RID = 2 → CH2 claims (desc\_r\_ready[2] = 1)

#### Data Transfers (Concurrent):

All 3 channels:

- datard\_valid[0,2,5] = 1 (requesting read engine)
- datawr\_valid[0,2,5] = 1 (requesting write engine)
- Read engine arbitrates AR channel
- Write engine arbitrates AW channel
- SRAM controller isolates channel data (per-channel FIFOs)

### MonBus Event Aggregation (9 Sources)

#### Concurrent Events:

Cycle N:

CH0 descriptor engine: Fetch complete → mon\_valid\_ch[0] = 1  
CH2 scheduler: State transition → mon\_valid\_ch[2] = 1  
Desc AXI monitor: Transaction complete → desc\_axi\_mon\_valid = 1

#### MonBus Arbiter:

Cycle N: Grant source 0 (CH0 desc eng) → mon\_packet = CH0 event  
Cycle N+1: Grant source 2 (CH2 sched) → mon\_packet = CH2 event  
Cycle N+2: Grant source 8 (AXI mon) → mon\_packet = AXI event

---

## Resource Sharing

### Shared: Descriptor AXI Master

**Why Shared:** - Descriptor fetch bandwidth: ~256 bits per descriptor - Infrequent access (once per descriptor, not per beat) - Total bandwidth << 1% of AXI bus capacity - 8 separate AXI masters wasteful

**Arbitration:** - Round-robin ensures fairness - ACK mode prevents head-of-line blocking - Out-of-order R responses (ID-based claiming)

### NOT Shared: Data Read/Write

**Why NOT Shared (Direct Passthrough):** - Data path bandwidth: Continuous streaming (many GB/s) - Need concurrent operation (all channels transferring simultaneously) - Engines perform internal arbitration: - Read engine: Round-robin AR arbiter - Write engine: Round-robin AW arbiter - Array just passes through (no arbitration overhead)

**Benefit:** - Scheduler\_group\_array remains simple - Arbitration complexity in engines (where it belongs) - Engines can optimize (space-aware, pipeline-aware)

---

## Configuration Management

### Global Configuration Distribution

**Single Config → All Channels:**

```
// All scheduler_groups receive same global config
for (genvar ch = 0; ch < NUM_CHANNELS; ch++) begin
 u_scheduler_group[ch].cfg_desceng_prefetch = cfg_desceng_prefetch;
 u_scheduler_group[ch].cfg_desceng_addr0_base =
cfg_desceng_addr0_base;
 // ... all other cfg_desceng_*, cfg_sched_*
end
```

**Per-Channel Override:** - cfg\_channel\_enable[ch]: Enable/disable individual channels - cfg\_channel\_reset[ch]: Reset individual channels

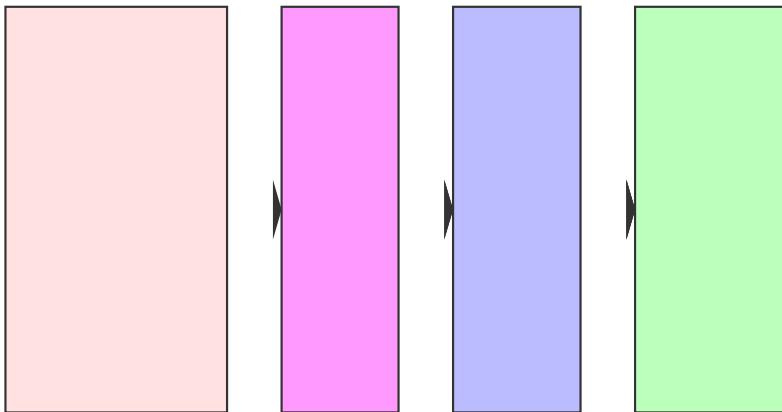
**Rationale:** - Global config simplifies software (single write configures all) - Per-channel enable allows selective activation - Per-channel reset allows fault isolation

---

## Timing Considerations

### AR Arbitration Latency

**Path:**



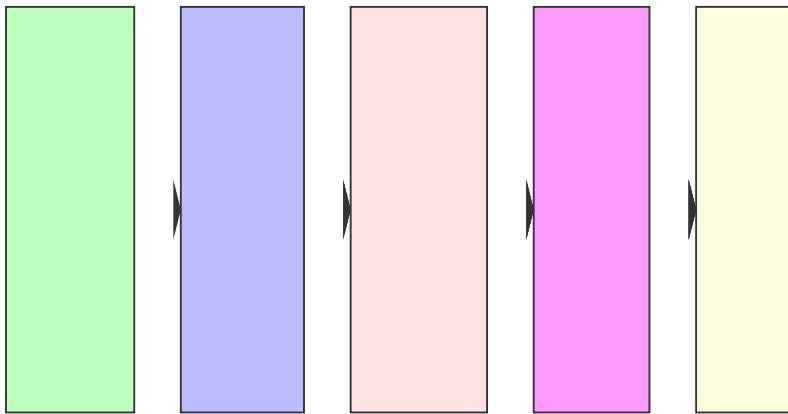
*AR Arbitration Path*

**Source:** [02\\_scheduler\\_group\\_array\\_ar\\_path.mmd](#)

**Latency:** 1 cycle (combinational arbiter + registered mux)

### R Broadcast Latency

**Path:**



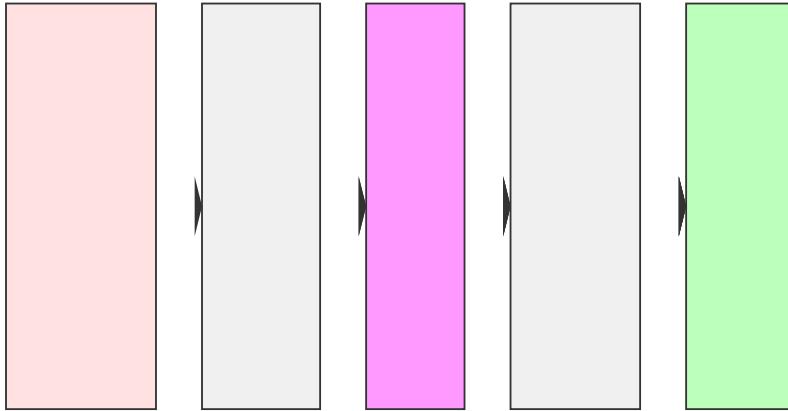
### *R Broadcast Path*

**Source:** [02\\_scheduler\\_group\\_array\\_r\\_path.mmd](#)

**Latency:** 1 cycle (combinational broadcast + ID compare)

### **MonBus Aggregation Latency**

**Path:**



### *MonBus Aggregation Path*

**Source:** [02\\_scheduler\\_group\\_array\\_monbus\\_path.mmd](#)

**Latency:** 2-4 cycles (skid buffers absorb backpressure)

---

## Testing

### Test Location:

`projects/components/stream/dv/tests/macro/scheduler_group_array/`

### Key Test Scenarios:

1. **All channels active** - 8 concurrent transfers
  2. **Descriptor AR arbitration** - All channels fetching simultaneously
  3. **R response distribution** - Out-of-order R data to correct channels
  4. **MonBus aggregation** - Burst events from all 9 sources
  5. **Per-channel reset** - Reset one channel without affecting others
  6. **Global config** - Verify config broadcast to all channels
  7. **AXI monitor** - Verify transaction tracking and error detection
- 

## Performance Metrics

### Descriptor Fetch Bandwidth

**Per Channel:** - 1 descriptor = 256 bits = 32 bytes - Fetch latency: ~100 cycles (DDR4 @ 100 cycle read latency) - Max rate: ~1 descriptor / 100 cycles per channel

**All Channels (Shared Master):** -  $8 \text{ channels} \times 1 \text{ descriptor} / 100 \text{ cycles} = 8 / 100 = 0.08 \text{ descriptors/cycle}$  - Negligible AXI bandwidth consumption

### Data Transfer Bandwidth

**Per Channel:** - Depends on engine PIPELINE mode and memory latency - PIPELINE=1: ~0.9 beats/cycle (near-peak)

**All Channels (Engines Arbitrate):** - Engines share single AXI master (read or write) - Total bandwidth = 1 AXI master capacity - Fair sharing via engine round-robin arbitration

---

## Related Documentation

- **Scheduler Group:** `11_scheduler_group.md` - Single channel control
  - **AXI Read Engine:** `08_axi_read_engine.md` - Data read arbitration
  - **AXI Write Engine:** `10_axi_write_engine.md` - Data write arbitration
  - **Stream Core:** `13_stream_core.md` - Top-level integration
-

Last Updated: 2025-11-16

## Scheduler Group Specification

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

projects/components/stream/rtl/macro/ **Status:** Implemented

---

### Overview

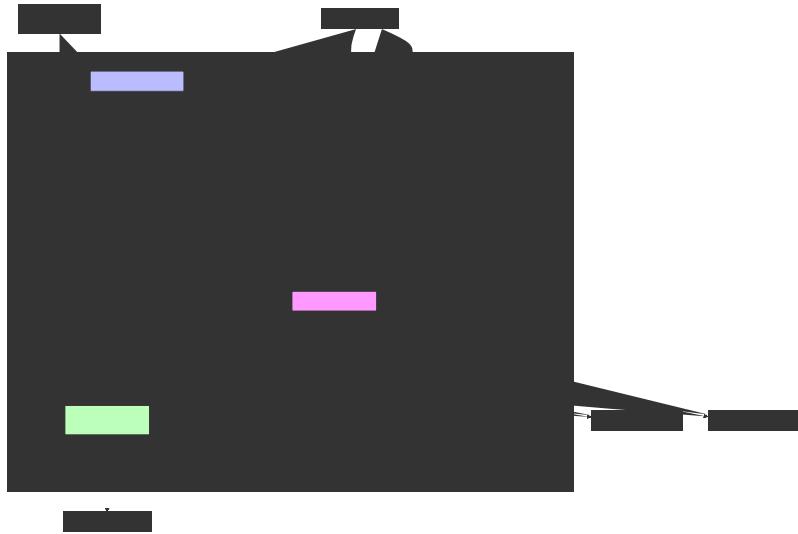
The Scheduler Group is a wrapper module that combines the descriptor engine and scheduler for a single channel, providing a complete descriptor-driven DMA control path. It aggregates MonBus events from both components and presents a unified interface to the system.

### Key Features

- **Component Integration:** Combines descriptor\_engine + scheduler into single unit
  - **MonBus Aggregation:** Arbitrates MonBus events from 2 sources (descriptor engine, scheduler)
  - **APB Kickoff:** Initial descriptor address programming via APB interface
  - **Configuration Management:** Unified config interface for both components
  - **Status Reporting:** Combined idle/state from both components
-

## Architecture

### Block Diagram



Diagram

->

---

### 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
(for scheduler)
parameter int AXI_ID_WIDTH = 8; // AXI ID width
parameter int TIMEOUT_CYCLES = 1000; // Scheduler timeout
threshold

// Monitor Bus Parameters - Base IDs for each component
parameter DESC_MON_AGENT_ID = 16; // 0x10 - Descriptor
Engine
parameter SCHED_MON_AGENT_ID = 48; // 0x30 - Scheduler
parameter MON_UNIT_ID = 1; // 0x1
parameter MON_CHANNEL_ID = 0; // Base channel ID
```

**MonBus Agent ID Convention:** - Descriptor Engine: 0x10 (16) - Scheduler: 0x30 (48) - Allows system to distinguish event sources

---

## 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          | APB descriptor address valid           |
| apb_ready | output    | 1          | Ready to accept APB descriptor address |
| apb_addr  | input     | ADDR_WIDTH | Initial descriptor address             |

### Configuration Interface

#### Channel Control:

| Signal             | Direction | Width | Description        |
|--------------------|-----------|-------|--------------------|
| cfg_channel_enable | input     | 1     | Enable channel     |
| cfg_channel_reset  | input     | 1     | Soft reset channel |

#### Scheduler Configuration (Future Enhancement):

| Signal                   | Direction | Width | Description                      |
|--------------------------|-----------|-------|----------------------------------|
| cfg_sched_timeout_cycles | input     | 16    | Timeout threshold (not used yet) |
| cfg_sched_timeout_enable | input     | 1     | Timeout enable (not used yet)    |
| cfg_sched_error          | input     | 1     | Error reporting (not used)       |

| Signal                     | Direction | Width | Description                       |
|----------------------------|-----------|-------|-----------------------------------|
| r_enable                   |           |       | yet)                              |
| cfg_sched_co<br>mpl_enable | input     | 1     | Completion events (not used yet)  |
| cfg_sched_pe               | input     | 1     | Performance events (not used yet) |
| rf_enable                  |           |       |                                   |

### Descriptor Engine Configuration:

| Signal                  | Direction | Width     | Description             |
|-------------------------|-----------|-----------|-------------------------|
| cfg_desceng_prefetch    | input     | 1         | Enable prefetch         |
| cfg_desceng_fifo_thresh | input     | 4         | FIFO threshold          |
| cfg_desceng_addr0_base  | input     | ADDR_WIDT | Address range 0 base H  |
| cfg_desceng_addr0_limit | input     | ADDR_WIDT | Address range 0 limit H |
| cfg_desceng_addr1_base  | input     | ADDR_WIDT | Address range 1 base H  |
| 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 idle            |
| scheduler_state        | output    | 7     | Scheduler state (ONE-HOT) |

### Descriptor Engine AXI4 Master Read Interface

#### AR Channel:

| Signal        | Direction | Width | Description        |
|---------------|-----------|-------|--------------------|
| desc_ar_valid | output    | 1     | Address read valid |

| Signal         | Direction | Width        | Description                          |
|----------------|-----------|--------------|--------------------------------------|
| desc_ar_ready  | input     | 1            | Address read ready                   |
| desc_ar_addr   | output    | ADDR_WIDTH   | Read address                         |
| desc_ar_len    | output    | 8            | Burst length - 1                     |
| desc_ar_size   | output    | 3            | Burst size<br>(log2(bytes per beat)) |
| desc_ar_burst  | output    | 2            | Burst type<br>(2'b01 = INCR)         |
| desc_ar_id     | output    | AXI_ID_WIDTH | Transaction ID                       |
| desc_ar_lock   | output    | 1            | Lock signal                          |
| desc_ar_cache  | output    | 4            | Cache attributes                     |
| desc_ar_prot   | output    | 3            | Protection attributes                |
| desc_ar_qos    | output    | 4            | Quality of Service                   |
| desc_ar_region | output    | 4            | Region identifier                    |

### R Channel:

| 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          | Descriptor data (FIXED 256-bit) |
| desc_r_resp  | input     | 2            | Read response<br>(2'b00 = OKAY) |
| desc_r_last  | input     | 1            | Last beat in burst              |
| desc_r_id    | input     | AXI_ID_WIDTH | Transaction ID                  |

### Data Read Interface (to AXI Read Engine)

| Signal                 | Direction | Width      | Description     |
|------------------------|-----------|------------|-----------------|
| datard_valid           | output    | 1          | Request read    |
| datard_ready           | input     | 1          | Engine ready    |
| datard_addr            | output    | ADDR_WIDTH | Source address  |
| datard_beats_remaining | output    | 32         | Beats left      |
| datard_channel_id      | output    | 4          | Channel ID      |
| datard_done_s          | input     | 1          | Read burst done |
| datard_beats_done      | input     | 32         | Beats completed |
| datard_error           | input     | 1          | Read error      |

### Data Write Interface (to AXI Write Engine)

| Signal                 | Direction | Width      | Description         |
|------------------------|-----------|------------|---------------------|
| datawr_valid           | output    | 1          | Request write       |
| datawr_ready           | input     | 1          | Engine ready        |
| datawr_addr            | output    | ADDR_WIDTH | Destination address |
| datawr_beats_remaining | output    | 32         | Beats left          |
| datawr_channel_id      | output    | 4          | Channel ID          |
| datawr_done_s          | input     | 1          | Write burst done    |
| datawr_beats_done      | input     | 32         | Beats completed     |
| datawr_error           | input     | 1          | Write error         |

### Unified Monitor Bus Interface

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

| Signal     | Direction | Width | Description               |
|------------|-----------|-------|---------------------------|
| mon_packet | output    | 64    | 64-bit monitor bus packet |

## Interface

### Clock and Reset

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

### APB Programming Interface

#### Initial Descriptor Kickoff:

```
input logic apb_valid;
output logic apb_ready;
input logic [ADDR_WIDTH-1:0] apb_addr; // Initial descriptor address
```

**Protocol:** 1. Software writes descriptor address to channel-specific APB register 2. APB bridge asserts apb\_valid with apb\_addr 3. Descriptor engine captures address when apb\_ready asserted 4. Descriptor engine begins fetching from apb\_addr

### Configuration Interface

#### Channel Control:

```
input logic cfg_channel_enable; // Enable
channel
input logic cfg_channel_reset; // Soft
reset channel
```

#### Scheduler Configuration (Future Enhancement):

```
input logic [15:0] cfg_sched_timeout_cycles; //
Timeout (not used yet)
input logic cfg_sched_timeout_enable; // Timeout enable (not used yet)
input logic cfg_sched_err_enable; // Error reporting (not used yet)
input logic cfg_sched_compl_enable; // Completion events (not used yet)
input logic cfg_sched_perf_enable; // Performance events (not used yet)
```

**Note:** Scheduler currently uses compile-time TIMEOUT\_CYCLES parameter.  
Runtime config inputs are placeholders for future enhancement.

### Descriptor Engine Configuration:

|                                     |                          |    |
|-------------------------------------|--------------------------|----|
| <b>input logic</b>                  | cfg_desceng_prefetch;    | // |
| <i>Enable prefetch</i>              |                          |    |
| <b>input logic [3:0]</b>            | cfg_desceng_fifo_thresh; | // |
| <i>FIFO threshold</i>               |                          |    |
| <b>input logic [ADDR_WIDTH-1:0]</b> | cfg_desceng_addr0_base;  | // |
| <i>Address range 0 base</i>         |                          |    |
| <b>input logic [ADDR_WIDTH-1:0]</b> | cfg_desceng_addr0_limit; | // |
| <i>Address range 0 limit</i>        |                          |    |
| <b>input logic [ADDR_WIDTH-1:0]</b> | cfg_desceng_addr1_base;  | // |
| <i>Address range 1 base</i>         |                          |    |
| <b>input logic [ADDR_WIDTH-1:0]</b> | cfg_desceng_addr1_limit; | // |
| <i>Address range 1 limit</i>        |                          |    |

### Status Interface

|                                  |                         |    |
|----------------------------------|-------------------------|----|
| <b>output logic</b>              | descriptor_engine_idle; | // |
| <i>Descriptor engine idle</i>    |                         |    |
| <b>output logic</b>              | scheduler_idle;         | // |
| <i>Scheduler idle</i>            |                         |    |
| <b>output logic [6:0]</b>        | scheduler_state;        | // |
| <i>Scheduler state (ONE-HOT)</i> |                         |    |

**Combined Idle:** - Channel fully idle when descriptor\_engine\_idle && scheduler\_idle

### Descriptor Engine AXI4 Master Read Interface

#### AR Channel:

|                                        |                 |
|----------------------------------------|-----------------|
| <b>output logic</b>                    | desc_ar_valid;  |
| <b>input logic</b>                     | desc_ar_ready;  |
| <b>output logic [ADDR_WIDTH-1:0]</b>   | desc_ar_addr;   |
| <b>output logic [7:0]</b>              | desc_ar_len;    |
| <b>output logic [2:0]</b>              | desc_ar_size;   |
| <b>output logic [1:0]</b>              | desc_ar_burst;  |
| <b>output logic [AXI_ID_WIDTH-1:0]</b> | desc_ar_id;     |
| <b>output logic</b>                    | desc_ar_lock;   |
| <b>output logic [3:0]</b>              | desc_ar_cache;  |
| <b>output logic [2:0]</b>              | desc_ar_prot;   |
| <b>output logic [3:0]</b>              | desc_ar_qos;    |
| <b>output logic [3:0]</b>              | desc_ar_region; |

#### R Channel:

|                     |               |
|---------------------|---------------|
| <b>input logic</b>  | desc_r_valid; |
| <b>output logic</b> | desc_r_ready; |

```

input logic [255:0] desc_r_data; // FIXED 256-
bit descriptor width
input logic [1:0] desc_r_resp;
input logic desc_r_last;
input logic [AXI_ID_WIDTH-1:0] desc_r_id;

```

**Descriptor Width:** - FIXED at 256 bits (STREAM descriptor format)

#### Data Read Interface (to AXI Read Engine)

|                                      |                         |    |
|--------------------------------------|-------------------------|----|
| <b>output logic</b>                  | datard_valid;           | // |
| <i>Request read</i>                  |                         |    |
| <b>input logic</b>                   | datard_ready;           | // |
| <i>Engine ready</i>                  |                         |    |
| <b>output logic [ADDR_WIDTH-1:0]</b> | datard_addr;            | // |
| <i>Source address</i>                |                         |    |
| <b>output logic [31:0]</b>           | datard_beats_remaining; | // |
| <i>Beats left</i>                    |                         |    |
| <b>output logic [3:0]</b>            | datard_channel_id;      | // |
| <i>Channel ID</i>                    |                         |    |

#### Completion:

|                           |                     |    |
|---------------------------|---------------------|----|
| <b>input logic</b>        | datard_done_strobe; | // |
| <i>Read burst done</i>    |                     |    |
| <b>input logic [31:0]</b> | datard_beats_done;  | // |
| <i>Beats completed</i>    |                     |    |

#### Error:

|                    |               |    |
|--------------------|---------------|----|
| <b>input logic</b> | datard_error; | // |
| <i>Read error</i>  |               |    |

#### Data Write Interface (to AXI Write Engine)

|                                      |                         |    |
|--------------------------------------|-------------------------|----|
| <b>output logic</b>                  | datawr_valid;           | // |
| <i>Request write</i>                 |                         |    |
| <b>input logic</b>                   | datawr_ready;           | // |
| <i>Engine ready</i>                  |                         |    |
| <b>output logic [ADDR_WIDTH-1:0]</b> | datawr_addr;            | // |
| <i>Destination address</i>           |                         |    |
| <b>output logic [31:0]</b>           | datawr_beats_remaining; | // |
| <i>Beats left</i>                    |                         |    |
| <b>output logic [3:0]</b>            | datawr_channel_id;      | // |
| <i>Channel ID</i>                    |                         |    |

#### Completion:

|                           |                     |    |
|---------------------------|---------------------|----|
| <b>input logic</b>        | datawr_done_strobe; | // |
| <i>Write burst done</i>   |                     |    |
| <b>input logic [31:0]</b> | datawr_beats_done;  | // |
| <i>Beats completed</i>    |                     |    |

## Error:

```
input logic datawr_error; //
Write error
```

## Unified Monitor Bus Interface

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

**Aggregation:** - Combines events from descriptor\_engine and scheduler - 2-client round-robin arbiter with skid buffers - Maintains event ordering within each component

---

## Internal Architecture

### Component Interconnect

#### Descriptor Engine → Scheduler:

```
logic desceng_to_sched_valid;
logic desceng_to_sched_ready;
logic desceng_to_sched_packet; // 256-bit
descriptors
logic desceng_to_sched_error;
logic desceng_to_sched_eos; // End of
stream
logic desceng_to_sched_eol; // End of
list
logic desceng_to_sched_eod; // End of
descriptor
logic desceng_to_sched_type; // Descriptor
type
```

#### Scheduler → Descriptor Engine:

```
assign sched_channel_idle = scheduler_idle;
```

**Feedback Loop:** - Scheduler idle signal tells descriptor engine when channel ready for next descriptor - Critical for descriptor chaining

## MonBus Aggregation

### Per-Component MonBus Signals:

```
// Descriptor Engine MonBus
logic desceng_mon_valid;
logic desceng_mon_ready;
```

```

logic [63:0] desceng_mon_packet;
// Scheduler MonBus
logic
logic
logic [63:0] sched_mon_valid;
 sched_mon_ready;
 sched_mon_packet;

```

### Arbiter Instantiation:

```

monbus_arbiter #(
 .CLIENTS (2),
 .INPUT_SKID_ENABLE (1),
 .OUTPUT_SKID_ENABLE (1),
 .INPUT_SKID_DEPTH (2),
 .OUTPUT_SKID_DEPTH (2)
) u_monbus_aggregator (
 .axi_aclk (clk),
 .axi_aresetn (rst_n),
 .block_arb (1'b0),
 // Client inputs (2 sources)
 .monbus_valid_in ('{desceng_mon_valid, sched_mon_valid}),
 .monbus_ready_in ('{desceng_mon_ready, sched_mon_ready}),
 .monbus_packet_in ('{desceng_mon_packet, sched_mon_packet}),
 // Aggregated output
 .monbus_valid (mon_valid),
 .monbus_ready (mon_ready),
 .monbus_packet (mon_packet)
);

```

**Arbitration:** - Round-robin between 2 clients - Input skid buffers prevent backpressure to components - Output skid buffer prevents stalls to system

---

## Operation Flows

### Initialization (APB Kickoff)

1. Software Setup:
  - Configure descriptor engine (address ranges, thresholds)
  - Configure scheduler (enable, timeout)
  - Write initial descriptor address to APB register
2. APB Transaction:

```
apb_valid = 1
apb_addr = 0x1000_0000 (initial descriptor)
```
3. Descriptor Engine:
  - Captures apb\_addr
  - Begins fetching descriptor from 0x1000\_0000

- AXI AR transaction: desc\_ar\_addr = 0x1000\_0000, desc\_ar\_len = 0  
(1 beat = 256 bits)

#### 4. Descriptor Arrives:

```
desc_r_data = {src_addr, dst_addr, length, ...}
desceng_to_sched_valid = 1
desceng_to_sched_packet = desc_r_data
```

#### 5. Scheduler:

- Receives descriptor
- Transitions IDLE → FETCH\_DESC → XFER\_DATA
- Begins concurrent read/write operations

### Normal Transfer Flow

#### Descriptor Engine:

```
[IDLE] → [FETCH] → [PARSE] → [DELIVER] → (chain?) → [IDLE]
| | | |
AR issue R data Pass to scheduler
```

#### Scheduler:

```
[IDLE] → [FETCH_DESC] → [XFER_DATA] → [COMPLETE] → (chain?) → [IDLE]
| | | |
Read/Write concurrent Check next_ptr
```

### Descriptor Chaining Flow

#### Transfer 0 Complete:

Scheduler: [COMPLETE] → Check next\_descriptor\_ptr != 0 → [NEXT\_DESC]  
Descriptor Engine: Receives scheduler\_idle = 0 → Fetch next descriptor

#### Transfer 1 Start:

Descriptor Engine: Fetch descriptor @ next\_ptr → Deliver to scheduler  
Scheduler: [NEXT\_DESC] → [FETCH\_DESC] → [XFER\_DATA] → ...

### MonBus Event Aggregation

#### Concurrent Events:

Cycle N: Descriptor Engine: Fetch complete → desceng\_mon\_valid = 1  
Cycle N: Scheduler: State transition → sched\_mon\_valid = 1

#### Arbiter Behavior:

Cycle N: Grant client 0 (descriptor engine)  
mon\_valid = 1, mon\_packet = desceng\_mon\_packet  
Cycle N+1: Grant client 1 (scheduler)  
mon\_valid = 1, mon\_packet = sched\_mon\_packet

---

## Configuration Details

### Descriptor Engine Configuration

#### Address Range Checking:

```
cfg_desceng_addr0_base = 0x1000_0000
cfg_desceng_addr0_limit = 0x1FFF_FFFF // 256MB range 0
```

```
cfg_desceng_addr1_base = 0x8000_0000
cfg_desceng_addr1_limit = 0x8FFF_FFFF // 256MB range 1
```

**Purpose:** - Validate descriptor addresses (prevent out-of-range fetches) - Support multiple memory regions (DDR, SRAM, etc.)

#### Prefetch:

```
cfg_desceng_prefetch = 1 // Enable prefetch of next descriptor
```

#### FIFO Threshold:

```
cfg_desceng_fifo_thresh = 4 // Minimum entries before scheduling
```

### Scheduler Configuration (Future)

**Currently Unused (compile-time parameters):** - cfg\_sched\_timeout\_cycles → Use TIMEOUT\_CYCLES parameter - cfg\_sched\_timeout\_enable → Always enabled - cfg\_sched\_err\_enable → Always enabled - cfg\_sched\_compl\_enable → Always enabled - cfg\_sched\_perf\_enable → Not implemented

**Future Enhancement:** - Runtime timeout adjustment - Selective MonBus event filtering - Performance counter control

---

## Error Handling

### Error Sources

**Descriptor Engine Errors:** - Invalid descriptor address (out of range) - AXI read error (RRESP != OKAY) - Descriptor fetch timeout

**Scheduler Errors:** - Invalid descriptor (valid bit = 0) - Read engine error (datard\_error) - Write engine error (datawr\_error) - Timeout (no progress for TIMEOUT\_CYCLES)

## Error Propagation

Descriptor Engine Error:  
desceng\_to\_sched\_error = 1  
→ Scheduler receives error  
→ Scheduler transitions to CH\_ERROR  
→ MonBus error **event** generated

Scheduler Error:

Scheduler transitions to CH\_ERROR  
→ MonBus error **event** generated  
→ Channel stops operation  
→ Requires cfg\_channel\_reset to recover

---

## MonBus Event Types

### Descriptor Engine Events

**Event Codes:** - DESC\_FETCH\_START: Descriptor fetch initiated - DESC\_FETCH\_COMPLETE: Descriptor fetched successfully - DESC\_PARSE\_ERROR: Descriptor parsing error - DESC\_ADDR\_ERROR: Address range violation

**Agent ID:** 0x10 (DESC\_MON\_AGENT\_ID)

### Scheduler Events

**Event Codes:** - SCHED\_STATE\_TRANSITION: FSM state change - SCHED\_IRQ: Interrupt generation (descriptor.gen\_irq) - SCHED\_ERROR: Error condition - SCHED\_TIMEOUT: Timeout expired - SCHED\_COMPLETE: Transfer complete

**Agent ID:** 0x30 (SCHED\_MON\_AGENT\_ID)

---

## Timing Characteristics

### Latency

**APB to First Descriptor Fetch:** - APB handshake: 1 cycle - Descriptor engine processing: 2-3 cycles - AXI AR issue: 1 cycle - Total: ~4-5 cycles

**Descriptor to Scheduler Delivery:** - AXI R latency: Variable (memory dependent) - Descriptor parsing: 1 cycle - Scheduler handshake: 1 cycle - Total: Memory latency + 2 cycles

## Throughput

**Descriptor Fetch Bandwidth:** - 256 bits/descriptor - Single outstanding fetch (non-pipelined) - Limited by AXI read latency

**MonBus Bandwidth:** - 2 events/cycle max (1 from each component) - Arbiter serializes to 1 event/cycle output - Skid buffers absorb bursts

---

## Testing

### Test Location:

projects/components/stream/dv/tests/macro/scheduler\_group/

### Key Test Scenarios:

1. **APB kickoff** - Initial descriptor programming
  2. **Single descriptor** - Basic operation
  3. **Descriptor chaining** - Multiple linked descriptors
  4. **MonBus aggregation** - Concurrent events from both components
  5. **Error handling** - Descriptor fetch errors, scheduler errors
  6. **Channel reset** - Soft reset during operation
  7. **Configuration** - Address range validation, prefetch enable
- 

## Assertions

### Formal Verification Properties:

```
// MonBus connectivity
property monitor_bus_connected;
 (desceng_mon_valid || sched_mon_valid) | -> ##[1:10] mon_valid;
endproperty

// Component integration
property descriptor_scheduler_handshake;
 (desceng_to_sched_valid && desceng_to_sched_ready) | ->
 ##[1:5] (scheduler_state != CH_IDLE);
endproperty

// Channel reset propagation
property channel_reset_propagation;
 cfg_channel_reset | -> ##[1:100] (descriptor_engine_idle &&
scheduler_idle);
endproperty
```

---

## Related Documentation

- **Descriptor Engine:** 03\_descriptor\_engine.md - Descriptor fetch and parsing
  - **Scheduler:** 02\_scheduler.md - DMA control logic
  - **MonBus Arbiter:** AMBA monitor bus specification
  - **Scheduler Group Array:** 12\_scheduler\_group\_array.md - Multi-channel integration
- 

**Last Updated:** 2025-11-16

## Scheduler Specification

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

**Status:** Implemented **Last Updated:** 2025-11-21

---

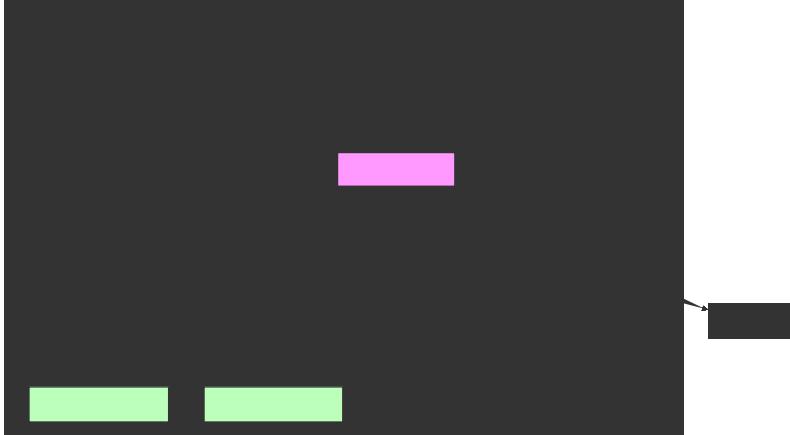
## Overview

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]     |

---

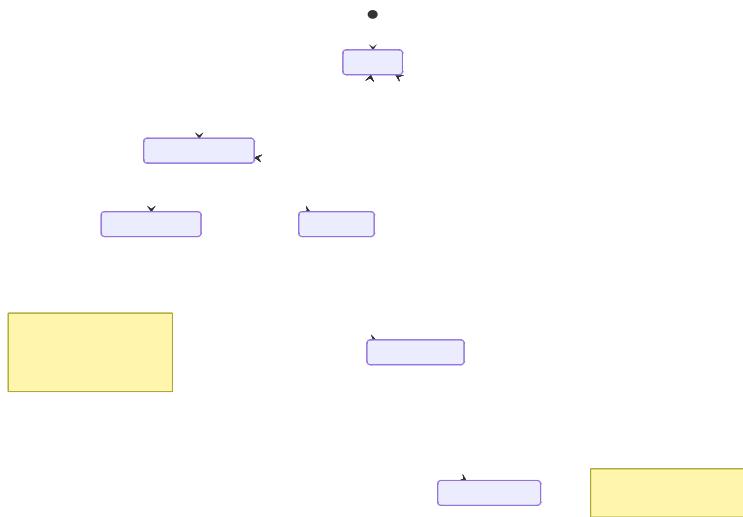
# FSM Operation

## 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:** - **Wait for:** All error flags cleared - **Action:** Report error via MonBus - **Next:** CH\_IDLE

---

## 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
```

**Recovery:** - CH\_ERROR → CH\_IDLE (when all errors cleared) - Software must reset channel (cfg\_channel\_reset)

---

## 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), // [55:52] Unit ID
 MON_UNIT_ID, // [51:46] Channel ID
 MON_CHANNEL_ID, // [45:40] Event code (IRQ)
 STREAM_EVENT_IRQ, // [39:0] Descriptor info
 descriptor_fields
};
```

**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:** 01\_descriptor\_engine.md - Descriptor fetch
  - **AXI Read Engine:** 08\_axi\_read\_engine.md - Source data read
  - **AXI Write Engine:** 10\_axi\_write\_engine.md - Destination data write
  - **SRAM Controller:** 05\_sram\_controller.md - Buffer management
  - **Scheduler Group:** 11\_scheduler\_group.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 |

**Last Updated:** 2025-11-21 (matched to current RTL implementation)

# Descriptor Engine Specification

**Module:** descriptor\_engine.sv **Location:**  
projects/components/stream/rtl/stream\_fub/

---

## Overview

The Descriptor Engine fetches descriptors from system memory via AXI and parses them into structured fields for the Scheduler using a 256-bit descriptor format.

## Key Features

- AXI master for descriptor fetch (256-bit read interface)
  - Descriptor FIFO buffering (depth configurable)
  - Parsing of 256-bit descriptor format
  - Handshake interface to Scheduler
  - Error detection and reporting
- 

## Interface

### Parameters

```
parameter int CHANNEL_ID = 0; // Channel identifier
parameter int NUM_CHANNELS = 32; // Total channels in system
parameter int CHAN_WIDTH = $clog2(NUM_CHANNELS); // Channel ID width
parameter int ADDR_WIDTH = 64; // Address bus width
parameter int AXI_ID_WIDTH = 8; // AXI ID width
parameter int FIFO_DEPTH = 8; // Descriptor FIFO depth
parameter int DESC_ADDR_FIFO_DEPTH = 2; // Descriptor address FIFO depth
parameter int TIMEOUT_CYCLES = 1000; // Timeout threshold
parameter logic [7:0] MON_AGENT_ID = 8'h10; // Monitor agent ID
parameter logic [3:0] MON_UNIT_ID = 4'h1; // Monitor unit ID
parameter logic [5:0] MON_CHANNEL_ID = 6'h0; // Monitor channel ID
```

---

## 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 |

### APB Programming Interface

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

### Scheduler Interface

| Signal                | Direction | Width | Description                          |
|-----------------------|-----------|-------|--------------------------------------|
| channel_idle          | input     | 1     | Scheduler idle (enables APB writes)  |
| descriptor_v<br>alid  | output    | 1     | Descriptor valid to scheduler        |
| descriptor_r<br>eady  | input     | 1     | Scheduler ready to accept descriptor |
| descriptor_p<br>acket | output    | 256   | 256-bit STREAM descriptor            |
| descriptor_e<br>rror  | output    | 1     | Descriptor fetch error               |
| descriptor_e<br>os    | output    | 1     | End of Stream flag                   |
| descriptor_e<br>ol    | output    | 1     | End of Line flag                     |
| descriptor_e<br>od    | output    | 1     | End of Data flag                     |
| descriptor_t<br>ype   | output    | 2     | Packet type                          |

### Shared AXI4 Master AR Channel

| Signal    | Direction | Width              | Description                          |
|-----------|-----------|--------------------|--------------------------------------|
| ar_valid  | output    | 1                  | Address read valid                   |
| ar_ready  | input     | 1                  | Address read ready                   |
| ar_addr   | output    | ADDR_WIDTH<br>H    | Read address                         |
| ar_len    | output    | 8                  | Burst length - 1                     |
| ar_size   | output    | 3                  | Burst size (log2(bytes per beat))    |
| ar_burst  | output    | 2                  | Burst type (2'b01 = INCR)            |
| ar_id     | output    | AXI_ID_WIDTH<br>TH | Transaction ID (contains channel ID) |
| ar_lock   | output    | 1                  | Lock signal                          |
| ar_cache  | output    | 4                  | Cache attributes                     |
| ar_prot   | output    | 3                  | Protection attributes                |
| ar_qos    | output    | 4                  | Quality of Service                   |
| ar_region | output    | 4                  | Region identifier                    |

### Shared AXI4 Master R Channel

| Signal  | Direction | Width        | Description                    |
|---------|-----------|--------------|--------------------------------|
| r_valid | input     | 1            | Read data valid                |
| r_ready | output    | 1            | Read data ready                |
| r_data  | input     | 256          | Read data (256-bit descriptor) |
| r_resp  | input     | 2            | Read response (2'b00 = OKAY)   |
| r_last  | input     | 1            | Last beat in burst             |
| r_id    | input     | AXI_ID_WIDTH | Transaction ID                 |

### Configuration Interface

| Signal        | Direction | Width | Description |
|---------------|-----------|-------|-------------|
| cfg_prefetch_ | input     | 1     | Enable      |

| Signal              | Direction | Width      | Description                 |
|---------------------|-----------|------------|-----------------------------|
| enable              |           |            | prefetch mode               |
| 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_r eset  | input     | 1          | Channel soft reset          |

### Status Interface

| Signal                  | Direction | Width | Description      |
|-------------------------|-----------|-------|------------------|
| descriptor_en gine_idle | output    | 1     | Engine idle flag |

### 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 |

### Ports

#### Clock and Reset:

```
input logic aclk;
input logic aresetn;
```

#### Descriptor Request (from Scheduler):

```



```

### Descriptor Output (to Scheduler):

```

output logic desc_valid;


```

### AXI Master (Descriptor Fetch):

```

// AXI AR (Address Read) Channel
output logic [ADDR_WIDTH-1:0] m_axi_araddr;
output logic [7:0] m_axi_arlen;
output logic [2:0] m_axi_arsize;
output logic [1:0] m_axi_arburst;
output logic [AXI_ID_WIDTH-1:0] m_axi_arid; // Transaction ID
output logic m_axi_arvalid;
input logic m_axi_arready;

// AXI R (Read Data) Channel
input logic [AXI_ID_WIDTH-1:0] m_axi_rid; // Transaction ID
input logic [DATA_WIDTH-1:0] m_axi_rdata;
input logic [1:0] m_axi_rresp;
input logic m_axi_rlast;
input logic m_axi_rvalid;
output logic m_axi_rready;

```

### Critical AXI ID Requirement:

The lower bits of `m_axi_arid` **MUST** contain the channel ID from the arbiter:

```

// Lower bits = channel ID (from arbiter grant)
// Upper bits = transaction counter (for multiple outstanding)
assign m_axi_arid = {transaction_counter, desc_req_channel_id[3:0]};

```

**Rationale:** - Allows responses to be routed back to correct channel - Enables MonBus packet generation with channel ID - Critical for debugging and transaction tracking - Channel ID comes from arbiter (whichever scheduler won arbitration for descriptor fetch)

### MonBus Output:

```

output logic monbus_valid;


```

---

## Descriptor Format

See `projects/components/stream/rtl/includes/stream_pkg.sv` for complete `descriptor_t` definition.

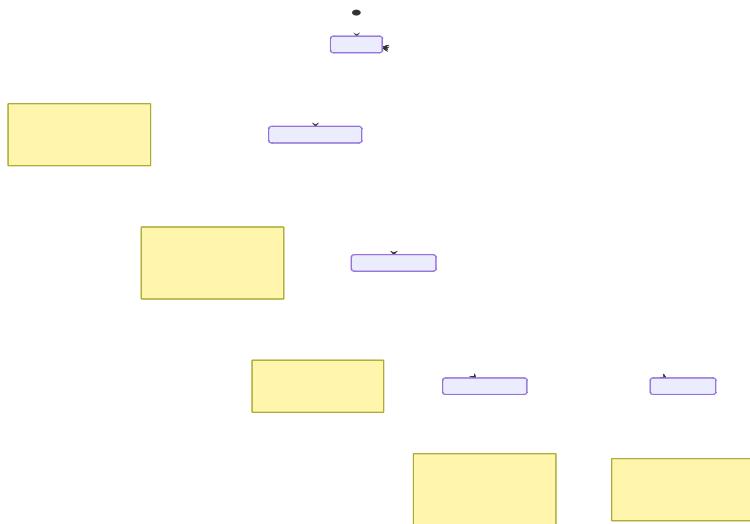
**256-bit structure:** - [63:0] src\_addr - Source address - [127:64] dst\_addr - Destination address - [159:128] length - Transfer length in BEATS - [191:160] next\_descriptor\_ptr - Next descriptor address - [192] valid - Valid flag - [193] interrupt - Interrupt enable - [194] last - Last descriptor flag - [199:196] channel\_id - Channel ID - [207:200] priority - Transfer priority

---

## Operation

### State Machine

The descriptor engine uses a 5-state FSM to manage descriptor fetch operations:



### Descriptor Engine FSM

**Source:** [05\\_descriptor\\_engine\\_fsm.mmd](#)

**States:** - **RD\_IDLE:** Waiting for descriptor address from APB kickoff or autonomous chaining - **RD\_ISSUE\_ADDR:** Issuing AXI AR transaction for 256-bit single-beat read - **RD\_WAIT\_DATA:** Waiting for AXI R response with descriptor data - **RD\_COMPLETE:** Descriptor parsed and forwarded to scheduler, check for

chaining - **RD\_ERROR**: AXI error response, report error via MonBus and return to idle

### Fetch Sequence

1. **Request:** Scheduler asserts desc\_req\_valid with desc\_req\_addr
2. **AXI Read:** Engine issues AXI AR transaction for descriptor
3. **Receive:** AXI R channel delivers 256-bit descriptor
4. **Parse:** Descriptor fields extracted into descriptor\_t structure
5. **Buffer:** Descriptor stored in FIFO
6. **Handoff:** Descriptor presented to Scheduler via desc\_valid/desc\_ready

### Error Handling

- **AXI Error:** RRESP != OKAY -> MonBus error packet
  - **Invalid Descriptor:** valid flag = 0 -> MonBus error packet
  - **FIFO Overflow:** Request rejected if FIFO full
- 

## Testing

### Test Location:

`projects/components/stream/dv/tests/fub_tests(descriptor_engine/`

**Test Scenarios:** 1. Single descriptor fetch 2. Back-to-back fetches 3. FIFO full backpressure 4. AXI error response 5. Invalid descriptor handling

---

## Related Documentation

- **Package:** `projects/components/stream/rtl/includes/stream_pkg.sv` - Descriptor format
  - **Source:** `projects/components/stream/rtl/stream_fub(descriptor_engine.sv)`
- 

**Last Updated:** 2025-10-17

## AXI Read Engine Specification

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

---

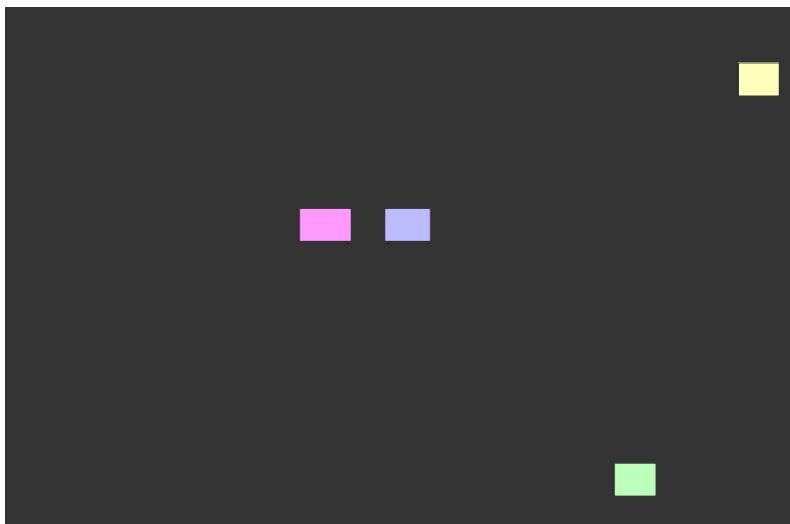
## Overview

The AXI Read Engine autonomously executes AXI read transactions to fetch source data from system memory. It accepts requests from schedulers through a round-robin arbiter, issues AR transactions, and streams data directly to the SRAM controller.

## Key Features

- **Multi-channel arbitration:** Round-robin arbiter across NUM\_CHANNELS
- **Space-aware request masking:** Only arbitrate channels with sufficient SRAM space
- **Pipelined operation:** Configurable PIPELINE parameter (0=non-pipelined, 1=pipelined)
- **Streaming data path:** Direct R channel → SRAM (no buffering)
- **Completion feedback:** Reports beats issued via done\_strobe to schedulers
- **Outstanding transaction tracking:** Per-channel counters for PIPELINE=1

## Block Diagram



Diagram

→

---

## Parameters

```
parameter int NUM_CHANNELS = 8; // Number of
independent channels
```

```

parameter int ADDR_WIDTH = 64; // AXI address width
parameter int DATA_WIDTH = 512; // AXI data width
parameter int ID_WIDTH = 8; // AXI ID width
parameter int SEG_COUNT_WIDTH = 8; // Width of
space/count signals
parameter int PIPELINE = 0; // 1: allow multiple
outstanding per channel // 0: wait for R last

before next AR
parameter int AR_MAX_OUTSTANDING = 8; // Maximum outstanding
AR per channel (PIPELINE=1)
parameter int STROBE_EVERY_BEAT = 0; // 0: strobe only on
AR handshake (default) // 1: strobe on every

R beat (not implemented)

```

### Short Aliases (internal use):

```

parameter int NC = NUM_CHANNELS;
parameter int AW = ADDR_WIDTH;
parameter int DW = DATA_WIDTH;
parameter int IW = ID_WIDTH;
parameter int SCW = SEG_COUNT_WIDTH;
parameter int CIW = (NC > 1) ? $clog2(NC) : 1; // Channel ID width

```

---

## 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 (applies to all channels) |

### Scheduler Interface

### Per-Channel Read Requests:

| Signal               | Direction | Width                            | Description                                   |
|----------------------|-----------|----------------------------------|-----------------------------------------------|
| sched_rd_val_id[ch]  | input     | NUM_CHAN<br>NELS                 | Channel requests read                         |
| sched_rd_ready[ch]   | output    | NUM_CHAN<br>NELS                 | Engine ready for channel                      |
| sched_rd_addr[r[ch]] | input     | NUM_CHAN<br>NELS ×<br>ADDR_WIDTH | Source addresses (static<br>base per channel) |
| sched_rd_beats[chs]  | input     | NUM_CHAN<br>NELS × 32            | Beats remaining to read<br>per channel        |

### Completion Feedback:

| Signal                   | Direction | Width                 | Description                                       |
|--------------------------|-----------|-----------------------|---------------------------------------------------|
| sched_rd_done_strobe[ch] | output    | NUM_CHAN<br>NELS      | AR issued (pulsed 1 cycle<br>per channel)         |
| sched_rd_beats_done[ch]  | output    | NUM_CHAN<br>NELS × 32 | Beats issued in burst per<br>channel              |
| sched_rd_error[or[ch]]   | output    | NUM_CHAN<br>NELS      | Sticky error flag per<br>channel (bad R response) |

## SRAM Allocation Interface

### Pre-allocation (before data arrives):

| Signal                      | Direction | Width                                     | Description                         |
|-----------------------------|-----------|-------------------------------------------|-------------------------------------|
| axi_rd_alloc_req            | output    | 1                                         | Request space<br>reservation        |
| axi_rd_alloc_size           | output    | 8                                         | Beats to reserve                    |
| axi_rd_alloc_id             | output    | ID_WIDTH                                  | Transaction ID →<br>channel mapping |
| axi_rd_alloc_space_free[ch] | input     | NUM_CHAN<br>NELS ×<br>SEG_COUNT<br>_WIDTH | Free space count per<br>channel     |

## SRAM Write Interface

### Data Stream (R → SRAM):

| Signal            | Direction | Width           | Description                      |
|-------------------|-----------|-----------------|----------------------------------|
| axi_rd_sram_valid | output    | 1               | Read data valid                  |
| axi_rd_sram_ready | input     | 1               | Ready to accept data             |
| axi_rd_sram_id    | output    | ID_WIDTH        | Transaction ID → channel mapping |
| axi_rd_sram_data  | output    | DATA_WIDTH<br>H | Read data payload                |

### AXI4 AR Channel (Read Address)

| Signal        | Direction | Width      | Description                          |
|---------------|-----------|------------|--------------------------------------|
| m_axi_arvalid | output    | 1          | Address valid                        |
| m_axi_arready | input     | 1          | Address ready                        |
| m_axi_arid    | output    | ID_WIDTH   | Transaction ID                       |
| m_axi_araddr  | output    | ADDR_WIDTH | Read address                         |
| m_axi_arlen   | output    | 8          | Burst length - 1                     |
| m_axi_arsize  | output    | 3          | Burst size<br>(log2(bytes per beat)) |
| m_axi_arburst | output    | 2          | Burst type<br>(2'b01 = INCR)         |

### AXI4 R Channel (Read Data)

| Signal       | Direction | Width      | Description                     |
|--------------|-----------|------------|---------------------------------|
| m_axi_rvalid | input     | 1          | Read data valid                 |
| m_axi_rready | output    | 1          | Read data ready                 |
| m_axi_rid    | input     | ID_WIDTH   | Transaction ID                  |
| m_axi_rdata  | input     | DATA_WIDTH | Read data                       |
| m_axi_rrresp | input     | 2          | Read response<br>(2'b00 = OKAY) |

| Signal      | Direction | Width | Description        |
|-------------|-----------|-------|--------------------|
| m_axi_rlast | input     | 1     | Last beat in burst |

## Debug Interface

| Signal                  | Direction | Width         | Description                                           |
|-------------------------|-----------|---------------|-------------------------------------------------------|
| dbg_rd_all_complete[ch] | output    | NUM_CHAN_NELS | All reads complete per channel (no outstanding txns)  |
| dbg_r_beats_rcvd        | output    | 32            | Total R beats received from AXI                       |
| dbg_sram_writes         | output    | 32            | Total writes to SRAM controller                       |
| dbg_arb_request[ch]     | output    | NUM_CHAN_NELS | Arbiter request vector (for bubble vs idle detection) |

## Interface

### Clock and Reset

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

### Configuration Interface

```
input logic [7:0] cfg_axi_rd_xfer_beats; // Transfer size in beats (all channels)
```

### Scheduler Interface (Per-Channel Read Requests)

#### Request Handshake:

|                                                         |                            |
|---------------------------------------------------------|----------------------------|
| input logic [NC-1:0]<br>requests read                   | sched_rd_valid; // Channel |
| output logic [NC-1:0]<br>ready for channel              | sched_rd_ready; // Engine  |
| input logic [NC-1:0][AW-1:0]<br>addresses (static base) | sched_rd_addr; // Source   |
| input logic [NC-1:0][31:0]<br>remaining to read         | sched_rd_beats; // Beats   |

#### Completion Feedback:

```

output logic [NC-1:0] sched_rd_done_strobe; // AR
issued (pulsed 1 cycle)
output logic [NC-1:0][31:0] sched_rd_beats_done; // Beats
issued in burst
output logic [NC-1:0] axi_rd_all_complete; // All
reads complete (sticky)

```

**Address Management:** - Scheduler provides **current address** in `sched_rd_addr` - Engine uses address directly from scheduler: `araddr = sched_rd_addr[channel_id]` - Scheduler increments address after each AR handshake

### AXI4 AR Channel (Read Address)

|                                                         |                                           |
|---------------------------------------------------------|-------------------------------------------|
| <b>output logic</b> [IW-1:0]                            | <code>m_axi_arid;</code>                  |
| <b>output logic</b> [AW-1:0]                            | <code>m_axi_araddr;</code>                |
| <b>output logic</b> [7:0]                               | <code>m_axi_arlen;</code> // Burst        |
| <i>length - 1</i>                                       |                                           |
| <b>output logic</b> [2:0]<br>( $\log_2(\text{bytes})$ ) | <code>m_axi_arsize;</code> // Burst size  |
| <b>output logic</b> [1:0]<br>(INCR)                     | <code>m_axi_arburst;</code> // Burst type |
| <b>output logic</b>                                     | <code>m_axi_arvalid;</code>               |
| <b>input logic</b>                                      | <code>m_axi_arready;</code>               |

### AXI ID Encoding:

```

// Lower bits = channel ID (from arbiter grant)
assign m_axi_arid = {{(IW-CW){1'b0}}, w_arb_grant_id};

```

### AXI4 R Channel (Read Data)

|                             |                            |
|-----------------------------|----------------------------|
| <b>input logic</b> [IW-1:0] | <code>m_axi_rid;</code>    |
| <b>input logic</b> [DW-1:0] | <code>m_axi_rdata;</code>  |
| <b>input logic</b> [1:0]    | <code>m_axi_rresp;</code>  |
| <b>input logic</b>          | <code>m_axi_rlast;</code>  |
| <b>input logic</b>          | <code>m_axi_rvalid;</code> |
| <b>output logic</b>         | <code>m_axi_rready;</code> |

### SRAM Allocation Interface (to SRAM Controller)

#### Pre-allocation (before data arrives):

|                                      |                                           |
|--------------------------------------|-------------------------------------------|
| <b>output logic</b>                  | <code>rd_alloc_req;</code> // Request     |
| <i>space reservation</i>             |                                           |
| <b>output logic</b> [7:0]            | <code>rd_alloc_size;</code> // Beats to   |
| <i>reserve</i>                       |                                           |
| <b>output logic</b> [IW-1:0]         | <code>rd_alloc_id;</code> //              |
| <i>Transaction ID → channel</i>      |                                           |
| <b>input logic</b> [NC-1:0][SCW-1:0] | <code>rd_space_free;</code> // Free space |
| <i>count per channel</i>             |                                           |

**Allocation Timing:** - rd\_alloc\_req asserted when AR transaction issues (arvalid && arready) - rd\_alloc\_size = configured burst size (cfg\_axi\_rd\_xfer\_beats) - rd\_alloc\_id = channel ID (lower bits of ARID)

## SRAM Write Interface (to SRAM Controller)

**Data Stream (R → SRAM):**

```

output logic axi_rd_sram_valid; // Read data
valid
input logic axi_rd_sram_ready; // Ready to
accept data
output logic [IW-1:0] axi_rd_sram_id; //
Transaction ID → channel
output logic [DW-1:0] axi_rd_sram_data; // Read data
payload

```

**Direct Passthrough (no buffering):**

```

assign axi_rd_sram_valid = m_axi_rvalid;
assign axi_rd_sram_id = m_axi_rid;
assign axi_rd_sram_data = m_axi_rdata;
assign m_axi_rready = axi_rd_sram_ready;

```

**Debug Interface**

|                              |                   |            |
|------------------------------|-------------------|------------|
| <b>output logic</b> [31:0]   | dbg_r_beats_rcvd; | // Total R |
| <i>beats from AXI</i>        |                   |            |
| <b>output logic</b> [31:0]   | dbg_sram_writes;  | // Total   |
| <i>writes to SRAM</i>        |                   |            |
| <b>output logic</b> [NC-1:0] | dbg_arb_request;  | // Arbiter |
| <i>request vector</i>        |                   |            |

**Purpose:** - dbg\_arb\_request exposed for **bubble vs idle detection** - When arb\_request == 0, system is idle (all channels in WRITE phase, no reads needed) - When arb\_request != 0 but arvalid == 0, true bubble (arbiter stalled)

---

## Operation

### Space-Aware Request Masking

**Critical:** Engine only arbitrates channels with sufficient SRAM space.

```

// Per-channel checks:
w_space_ok[i] = (rd_space_free[i] >= (cfg_axi_rd_xfer_beats << 1));
// 2x margin

w_below_outstanding_limit[i] = !r_outstanding_limit[i]; // Can issue

```

*new AR*

```
w_beats_ok[i] = ({24'h0, cfg_axi_rd_xfer_beats}) <= sched_rd_beats[i];
// Final arbitration request
w_arb_request[i] = sched_rd_valid[i] && w_space_ok[i] &&
w_below_outstanding_limit[i] && w_beats_ok[i];
```

**2x Space Margin:** - Requires  $2 \times \text{burst\_size}$  free space (not just `burst_size`) -  
Accounts for in-flight allocation timing - Prevents SRAM overflow during  
pipelined operation

## Round-Robin Arbitration

### Multi-Channel (NC > 1):

```
arbiter_round_robin #(
 .CLIENTS (NC),
 .WAIT_GNT_ACK (1) // Use ACK mode (wait for AR
handshake)
) u_arbiter (
 .request (w_arb_request),
 .grant_ack (w_arb_grant_ack),
 .grant_valid (w_arb_grant_valid),
 .grant (w_arb_grant),
 .grant_id (w_arb_grant_id)
)

assign w_arb_grant_ack = w_arb_grant & {NC{({m_axi_arvalid &&
m_axi_arready})}};
```

### Single Channel (NC == 1):

```
// Direct passthrough (no arbiter overhead)
assign w_arb_grant_valid = w_arb_request[0];
assign w_arb_grant = w_arb_request;
assign w_arb_grant_id = 1'b0;
```

## AR Channel Management (Combinational)

**Critical:** AR outputs driven **combinatorially** from arbiter (not registered).

```
// COMBINATIONAL outputs (no flops!)
assign m_axi_arvalid = w_arb_grant_valid;
assign m_axi_arid = {{(IW-CW){1'b0}}, w_arb_grant_id};
assign m_axi_araddr = sched_rd_addr[w_arb_grant_id] +
 (AW'(r_beats_issued[w_arb_grant_id]) << AXSIZE);
assign m_axi_arlen = cfg_axi_rd_xfer_beats - 8'd1;
```

```

assign m_axi_araddr = 3'(AXSIZE);
assign m_axi_arburst = 2'b01; // INCR

```

**Why Combinational:** - When rd\_space\_free goes to 0, arvalid must drop **same cycle** - No 1-cycle delay (immediate backpressure response)

## Outstanding Transaction Tracking

### PIPELINE=0 (Non-pipelined):

```

logic [NC-1:0] r_outstanding_limit; // Binary: 0 or 1 outstanding

// Set when AR issues
if (m_axi_arvalid && m_axi_arready && (w_arb_grant_id == i[CW-1:0]))
 r_outstanding_limit[i] <= 1'b1;

// Clear when R last arrives
if (m_axi_rvalid && m_axi_rready && m_axi_rlast && (m_axi_rid[CW-1:0]
== i[CW-1:0]))
 r_outstanding_limit[i] <= 1'b0;

```

### PIPELINE=1 (Pipelined):

```

logic [NC-1:0][MOW-1:0] r_outstanding_count; // 0 to
AR_MAX_OUTSTANDING

// Increment on AR issue, decrement on R last
case ({w_incr[i], w_decr[i]})

 2'b10: r_outstanding_count[i] <= r_outstanding_count[i] + 1'b1;
 // AR only
 2'b01: r_outstanding_count[i] <= r_outstanding_count[i] - 1'b1;
 // R last only
endcase

// Boolean flag = at or exceeds limit
r_outstanding_limit[i] = (r_outstanding_count[i] >=
AR_MAX_OUTSTANDING);

```

## Completion Signal Behavior

### Sticky Completion Flag:

```

// Set when outstanding transactions reach zero
if (r_outstanding_count[i] == '0)
 r_all_complete[i] <= 1'b1;

// Clear on 0-to-nonzero transition (first AR of new descriptor)
else if (r_all_complete_prev[i] && (r_outstanding_count[i] != '0))
 r_all_complete[i] <= 1'b0;

```

- Why Sticky:** - Prevents false pulses when count temporarily hits 0 between bursts
- Scheduler sees stable completion signal

## Completion Strobe Generation

**Critical:** Strobe asserted on **AR handshake**, not R completion.

```
// Pulse when AR transaction accepted (arvalid && arready)
if (m_axi_arvalid && m_axi_arready) begin
 r_done_strobe[w_arb_grant_id] <= 1'b1;
 r_beats_done[w_arb_grant_id] <= {24'd0, cfg_axi_rd_xfer_beats};
end
```

**Rationale:** - Scheduler needs immediate feedback (doesn't wait for R data) -  
 Allows scheduler to decrement beats and issue next request - Pipelined operation  
 hides R latency

## SRAM Pre-Allocation

### Allocation Request:

```
// Assert allocation when AXI AR command issues
if (m_axi_arvalid && m_axi_arready) begin
 r_alloc_req <= 1'b1;
 r_alloc_size <= cfg_axi_rd_xfer_beats;
 r_alloc_id <= {{(IW-CW){1'b0}}, w_arb_grant_id};
end
```

**Single-Cycle Pulse:** - rd\_alloc\_req asserted for 1 cycle - SRAM controller reserves space immediately - Prevents race condition (R data arriving before allocation)

---

## PIPELINE Modes

### PIPELINE=0 (Non-pipelined)

**Behavior:** - Wait for R last before issuing next AR (per channel) - Only 1 outstanding transaction per channel - Simple tracking (binary flag)

**Performance:** - Latency-limited (each burst waits for previous R last) - Good for low-latency SRAM (2-3 cycle read latency) - ~0.40 beats/cycle (FPGA SRAM)

**Area:** - Minimal logic (~1,250 LUTs)

## PIPELINE=1 (Pipelined)

**Behavior:** - Issue multiple AR before R last (per channel) - Up to AR\_MAX\_OUTSTANDING transactions per channel - Counter tracking (0 to AR\_MAX\_OUTSTANDING)

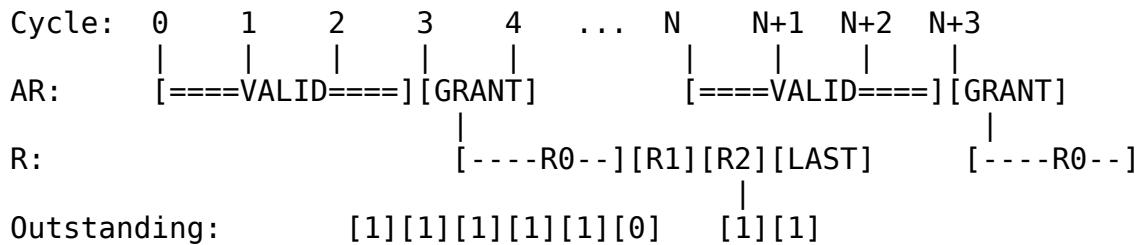
**Performance:** - Hides memory latency (command pipelining) - Good for DDR3/DDR4 (50-100 cycle read latency) - ~0.94 beats/cycle (DDR4)

**Area:** - Moderate increase (~2,000 LUTs, 1.6x)

---

## Timing Diagrams

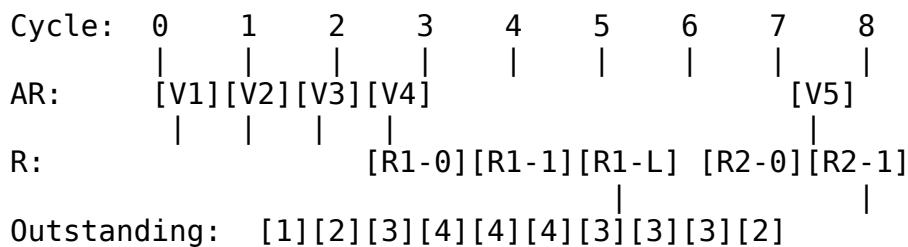
### PIPELINE=0 Transaction Flow



#### Notes:

- AR waits for previous R last before issuing
- Outstanding flag: 1 = AR issued, 0 = R last received

### PIPELINE=1 Transaction Flow



#### Notes:

- AR issues without waiting for R last
  - Outstanding counter: 0-4 (AR\_MAX\_OUTSTANDING=4)
  - R data arrives asynchronously
-

## Address Management

### Scheduler Handles Address Increment

Scheduler behavior:

```
// Provides current read address
sched_rd_addr[ch] <= current_address;

// After AR handshake, increment address
if (ar_handshake) begin
 sched_rd_addr[ch] <= sched_rd_addr[ch] + (burst_len *
BYTES_PER_BEAT);
end
```

### Engine Uses Address Directly

Engine behavior:

```
// Use address directly from scheduler (no internal tracking)
assign m_axi_araddr = sched_rd_addr[w_arb_grant_id];
```

**Key Design Choice:** - Scheduler manages address increment after each AR transaction  
- Engine uses address directly from scheduler input  
- Engine reports completion via sched\_rd\_done\_strobe and sched\_rd\_beats\_done  
- Scheduler updates address based on beats\_done feedback

---

## Debug and Verification

### Bubble vs Idle Detection

**System Idle (NOT a bubble):**

```
// When all channels in WRITE phase (no READ needed)
dbg_arb_request == 0 && m_axi_arvalid == 0
```

**True Bubble (stalled arbiter):**

```
// Channels requesting but arbiter not granting
dbg_arb_request != 0 && m_axi_arvalid == 0
```

**Example Scenario (4 channels, xb2 bursts):** - All 4 channels complete READ phase → start WRITE phase - Brief period: all channels writing, none reading - arb\_request = 4'b0000 (system idle, not bubble) - This is **expected behavior**, not performance bug

## Debug Counters

```
dbg_r_beats_rcvd // Total R beats received from AXI
dbg_sram_writes // Total writes to SRAM controller

// Should match at end of test
assert (dbg_r_beats_rcvd == dbg_sram_writes);
```

---

## Error Handling

### AXI Response Errors

#### Detection:

```
if (m_axi_rvalid && m_axi_rready && (m_axi_rresp != 2'b00)) begin
 // Log error (via MonBus or internal register)
 // Continue operation (depending on error policy)
end
```

**Current Implementation:** - Errors logged but not exposed on interface - Future enhancement: error output signals

### Space Overflow Protection

#### Prevention:

```
// Only arbitrate channels with sufficient space
w_space_ok[i] = (rd_space_free[i] >= (cfg_axi_rd_xfer_beats << 1));

// 2x margin prevents overflow during pipelined operation
```

---

## Testing

### Test Location:

projects/components/stream/dv/tests/fub\_tests/axi\_engines/

### Key Test Scenarios:

1. **Single burst read** - Basic functionality
2. **Multi-burst transfer** - beats > xfer\_beats
3. **SRAM backpressure** - rd\_space\_free = 0
4. **Multi-channel arbitration** - All channels requesting
5. **PIPELINE=0 mode** - Sequential operation
6. **PIPELINE=1 mode** - Pipelined operation

7. **Outstanding limit** - Hit AR\_MAX\_OUTSTANDING
  8. **Address increment** - Verify correct offset calculation
- 

## Performance Characteristics

### PIPELINE=0 Performance

| Memory Type | Latency (cycles) | Throughput<br>(beats/cycle) |
|-------------|------------------|-----------------------------|
| FPGA SRAM   | 2-3              | 0.40                        |
| DDR3        | 50-70            | 0.17                        |
| DDR4        | 70-100           | 0.14                        |

### PIPELINE=1 Performance

| Memory Type | Latency (cycles) | Throughput (beats/cycle) | Improvement vs PIPELINE=0 |
|-------------|------------------|--------------------------|---------------------------|
| FPGA SRAM   | 2-3              | 0.85                     | 2.1x                      |
| DDR3        | 50-70            | 0.89                     | 5.2x                      |
| DDR4        | 70-100           | 0.94                     | 6.7x                      |

**Key Insight:** Pipelined mode provides greatest benefit with high-latency memory.

---

## Related Documentation

- **AXI Write Engine:** 10\_axi\_write\_engine.md - Complementary write path
  - **Scheduler:** 02\_scheduler.md - Request interface
  - **SRAM Controller:** 05\_sram\_controller.md - Data destination
  - **Stream Core:** 12\_stream\_core.md - Top-level integration
- 

**Last Updated:** 2025-11-16

## Stream Allocation Controller

**Module:** stream\_alloc\_ctrl.sv **Category:** FUB (Functional Unit Block) **Parent:** sram\_controller\_unit.sv

## 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:

$$\text{space\_free} = \text{DEPTH} - (\text{wr\_ptr} - \text{rd\_ptr})$$

### 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        |
| 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 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     |

| Signal          | Direction | Width | Description  |
|-----------------|-----------|-------|--------------|
| 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

## References

- **Drain Controller:** stream\_drain\_ctrl.md
- **SRAM Controller Unit:** sram\_controller\_unit.md
- **AXI Read Engine:** axi\_read\_engine.md

## SRAM Controller Specification

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

**Status:** Implemented **Last Updated:** 2025-11-21

---

## 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

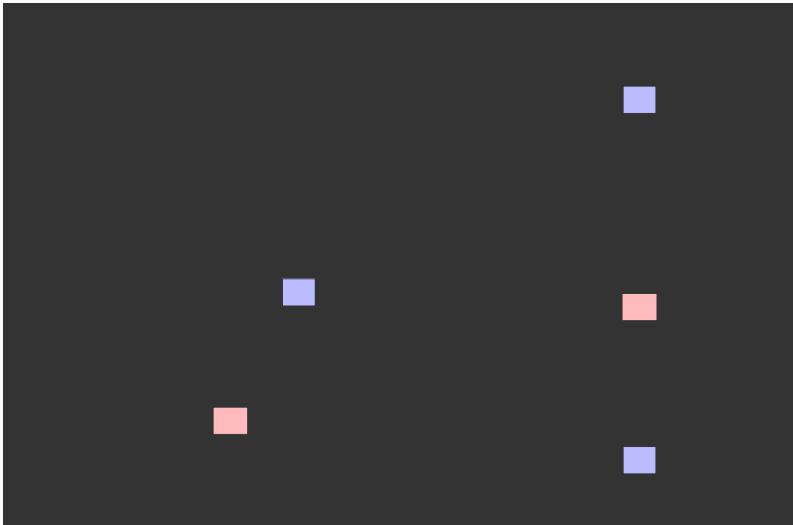
**Trade-offs:** - More SRAM resources ( $NC \times DEPTH$  vs. shared pool) - Potential waste if channels idle (unused FIFO space)

**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:** [06\\_sram\\_controller\\_unit.md](#) - Per-channel implementation
  - **Allocation Controller:** [07\\_stream\\_alloc\\_ctrl.md](#) - Write-side flow control
  - **Drain Controller:** [09\\_stream\\_drain\\_ctrl.md](#) - Read-side flow control
  - **AXI Read Engine:** [08\\_axi\\_read\\_engine.md](#) - Write interface usage
  - **AXI Write Engine:** [10\\_axi\\_write\\_engine.md](#) - Read interface usage
- 

## 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 |

**Last Updated:** 2025-11-21 (verified against RTL implementation)

## SRAM Controller Unit

**Module:** `sram_controller_unit.sv` **Category:** FUB (Functional Unit Block)

**Parent:** `sram_controller.sv`

## Overview

The `sram_controller_unit` module is a **per-channel data buffering pipeline** combining three components to provide flow-controlled data storage between the AXI read engine and AXI write engine. Each channel instantiates its own `sram_controller_unit`.

### What Makes This Different from Other Approaches

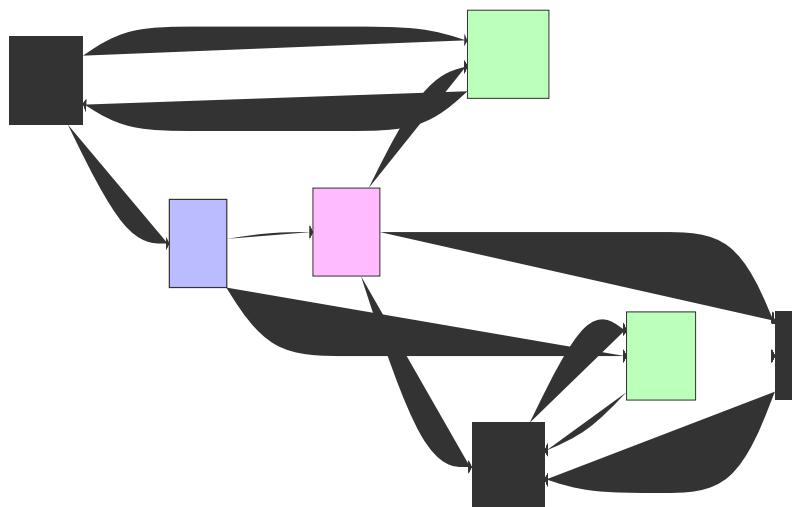
**Previous Approach (Documented but Not Implemented):** - Monolithic SRAM buffer divided into segments per channel - Complex pointer arithmetic to manage segment boundaries - Shared read/write ports requiring arbitration

**Current Approach (Actual Implementation):** - **Independent FIFO per channel** - No segment management, FIFO handles internally - **No pointer arithmetic** - FIFO pointer logic is self-contained - **No arbitration** - Each channel has dedicated FIFO

**Why This Architecture:** 1. **Simplicity** - FIFO manages address generation internally 2. **Independence** - Channels operate without coordination 3. **Predictability** - No arbitration contention 4. **Scalability** - Easy to parameterize depth per channel

## Three-Component Architecture

The `sram_controller_unit` combines three blocks:



*Diagram*

→

## 1. Allocation Controller (`stream_alloc_ctrl`)

**Purpose:** Track reserved vs. committed FIFO space

**Why Needed:** - AXI read engine requests bursts BEFORE data arrives - FIFO space must be reserved to prevent overflow - Separate tracking of allocation (request) vs. fulfillment (data arrival)

**Operation:**

```
// Reserve space (burst request)
rd_alloc_req = 1'b1;
rd_alloc_size = 8'd16; // Reserve 16 beats

// Later: Data arrives, fulfills reservation
fifo_wr_valid = 1'b1; // Advance FIFO write pointer (one beat)

// Space tracking
rd_space_free updates to show available UNRESERVED space
```

**Key Insight:** Allocation controller is a “virtual FIFO” without data - just pointer tracking.

## 2. FIFO Buffer (`gaxi_fifo_sync`)

**Purpose:** Physical data storage

**Configuration:**

```
gaxi_fifo_sync #(
 .MEM_STYLE(FIFO_AUTO), // Let synthesis tool choose RAM type
 .REGISTERED(1), // 1-cycle read latency (mimics SRAM
behavior)
 .DATA_WIDTH(DW),
 .DEPTH(SD)
)
```

**Why REGISTERED=1:** - Mimics true SRAM behavior (read latency) - Improves timing closure - Enables use of block RAM resources

**Data Path:** - Write: Direct from AXI read engine - Read: To latency bridge (internal connection)

## 3. Latency Bridge (`stream_latency_bridge`)

**Purpose:** Compensate for FIFO 1-cycle read latency

**Problem Without Bridge:**

```
Cycle 0: rd_ready asserted → rd_valid asserted
Cycle 1: rd_data appears (1-cycle latency)
Result: valid and data misaligned!
```

**Solution:** - Bridge buffers read data - Aligns valid/ready/data on output - Provides backpressure to FIFO read port

**Output:** Clean valid/ready/data interface to AXI write engine

## Flow Control Architecture

### Allocation Controller Naming (CONFUSING!)

**CRITICAL:** Allocation controller uses OPPOSITE naming from normal FIFO:

**Normal FIFO:** - wr\_\* = Write data (consumes space) - rd\_\* = Read data (frees space)

**Allocation Controller:** - wr\_\* = ALLOCATE space (reserve, decreases space\_free) - rd\_\* = RELEASE allocation (data exits controller, increases space\_free)

**Why This Matters:**

```
// Allocation controller connections (COUNTERINTUITIVE!)

// "Write" side = ALLOCATE (reserve space for upcoming burst)
.wr_valid (rd_alloc_req), // Read engine says "I need 16 beats"
.wr_size (rd_alloc_size), // Number of beats to reserve

// "Read" side = RELEASE (data exits controller, free space)
.rd_valid (axi_wr_sram_valid && axi_wr_sram_ready), // Output handshake!
```

**Key Insight:** Allocation controller tracks OUTPUT side (after latency bridge), not FIFO input!

### Drain Controller Naming (NORMAL!)

**Drain controller uses STANDARD FIFO naming:**

**Drain Controller:** - wr\_\* = Data written to FIFO (increment occupancy) - rd\_\* = Drain request from write engine (reserve data)

```
// Drain controller connections (STANDARD!)

// Write side = Data written to FIFO
.wr_valid (axi_rd_sram_valid && axi_rd_sram_ready), // FIFO write handshake
```

```
// Read side = Drain request (reserve data for upcoming write burst)
.rd_valid (wr_drain_req),
.rd_size (wr_drain_size)
```

## Data Available Calculation

**Total data available = FIFO data + Latency bridge buffered data**

```
assign wr_drain_data_avail = drain_data_available + bridge_occupancy;
```

**Why Add Bridge Occupancy:** - Drain controller tracks FIFO only (doesn't see latency bridge) - Bridge can hold up to 5 beats (1 in-flight + 4 in skid buffer) - Write engine needs total available data count

## Example:

FIFO count: 100 beats

Bridge occupancy: 3 beats

`wr_drain_data_avail = 100 + 3 = 103 beats available to write engine`

## 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 |

**Note:** SEG\_COUNT\_WIDTH has +1 bit to distinguish full (count=DEPTH) from empty (count=0).

---

## 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     | Allocate space request      |
| axi_rd_alloc_size       | input     | 8     | Number of beats to allocate |
| axi_rd_alloc_space_free | output    | SCW   | Available unreserved space  |

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

| Signal            | Direction | Width | Description                          |
|-------------------|-----------|-------|--------------------------------------|
| axi_rd_sram_valid | input     | 1     | Data valid from read engine          |
| axi_rd_sram_ready | output    | 1     | Ready to accept data (FIFO not full) |
| axi_rd_sram_data  | input     | DW    | Data from read engine                |

### Drain Interface (Write Engine Flow Control)

| Signal                  | Direction | Width | Description                    |
|-------------------------|-----------|-------|--------------------------------|
| axi_wr_drain_data_avail | output    | SCW   | Available data (FIFO + bridge) |
| axi_wr_drain_req        | input     | 1     | Reserve data request           |
| axi_wr_drain_size       | input     | 8     | Number of beats to reserve     |

### Read Interface (FIFO → Latency Bridge → AXI Write Engine)

| Signal            | Direction | Width | Description                |
|-------------------|-----------|-------|----------------------------|
| axi_wr_sram_valid | output    | 1     | Data valid to write engine |
| axi_wr_sram_ready | input     | 1     | Write engine ready         |
| axi_wr_sram_data  | output    | DW    | Data to write engine       |

## Debug Interface

| Signal               | Direction | Width | Description                     |
|----------------------|-----------|-------|---------------------------------|
| dbg_bridge_pending   | output    | 1     | Latency bridge has pending read |
| dbg_bridge_out_valid | output    | 1     | Latency bridge output valid     |

## Interfaces

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

| Signal            | Direction | Width | Description                          |
|-------------------|-----------|-------|--------------------------------------|
| axi_rd_sram_valid | Input     | 1     | Data valid from read engine          |
| axi_rd_sram_ready | Output    | 1     | Ready to accept data (FIFO not full) |
| axi_rd_sram_data  | Input     | DW    | Data from read engine                |

**Connection:** Direct to FIFO write port

### Read Interface (Latency Bridge → AXI Write Engine)

| Signal            | Direction | Width | Description                |
|-------------------|-----------|-------|----------------------------|
| axi_wr_sram_valid | Output    | 1     | Data valid to write engine |
| axi_wr_sram_ready | Input     | 1     | Write engine ready         |
| axi_wr_sram_data  | Output    | DW    | Data to write engine       |

**Connection:** From latency bridge output

### Allocation Interface (Read Engine Flow Control)

| Signal        | Direction | Width | Description            |
|---------------|-----------|-------|------------------------|
| rd_alloc_req  | Input     | 1     | Allocate space request |
| rd_alloc_size | Input     | 8     | Number of beats to     |

| Signal        | Direction | Width | Description                |
|---------------|-----------|-------|----------------------------|
| rd_space_free | Output    | SCW   | Available unreserved space |

### Usage:

```
// Read engine checks space before issuing AR
if (rd_space_free >= (cfg_axi_rd_xfer_beats << 1)) begin // 2x margin
 rd_alloc_req = 1'b1;
 rd_alloc_size = cfg_axi_rd_xfer_beats;
end
```

### Drain Interface (Write Engine Flow Control)

| Signal               | Direction | Width | Description                    |
|----------------------|-----------|-------|--------------------------------|
| wr_drain_req         | Input     | 1     | Reserve data request           |
| wr_drain_size        | Input     | 8     | Number of beats to reserve     |
| wr_drain_dat_a_avail | Output    | SCW   | Available data (FIFO + bridge) |

### Usage:

```
// Write engine checks data before issuing AW
if (wr_drain_data_avail >= cfg_axi_wr_xfer_beats) begin
 wr_drain_req = 1'b1;
 wr_drain_size = cfg_axi_wr_xfer_beats;
end
```

## Operation

### Normal Data Flow

#### Step 1: Read Engine Allocates Space

rd\_alloc\_req = 1, rd\_alloc\_size = 16  
 → Allocation wr\_ptr += 16  
 → rd\_space\_free decreases by 16

#### Step 2: Read Engine Issues AXI AR

m\_axi\_arvalid = 1, m\_axi\_arlen = 15 (16 beats)  
 → AXI read starts

#### Step 3: AXI Read Data Arrives

For each beat:

- axi\_rd\_sram\_valid = 1
- → FIFO write (data enters)
- → Drain wr\_ptr += 1 (data available increases)

#### **Step 4: Write Engine Checks Data Available**

if (wr\_drain\_data\_avail >= 16) → Can issue AXI write

#### **Step 5: Write Engine Drains Data**

wr\_drain\_req = 1, wr\_drain\_size = 16  
 → Drain rd\_ptr += 16 (reserve data)  
 → wr\_drain\_data\_avail decreases by 16

For each beat:

- axi\_wr\_sram\_ready = 1
- → FIFO read → Latency bridge → Output
- → Allocation rd\_ptr += 1 (release space)
- → rd\_space\_free increases by 1

#### **Concurrent Read/Write**

**CRITICAL:** Read and write can occur simultaneously!

Cycle N:

axi\_rd\_sram\_valid = 1 (read engine writing to FIFO)  
 axi\_wr\_sram\_valid = 1, axi\_wr\_sram\_ready = 1 (write engine draining)

Result:

FIFO count remains constant (1 write + 1 read = no net change)  
 Allocation rd\_ptr += 1 (space released)  
 Drain wr\_ptr += 1, rd\_ptr += 0 (data added faster than reserved)

**Why This Matters:** - Enables streaming operation - Prevents deadlock when transfer size > FIFO depth - Maximizes throughput

#### **Debug Signals**

| Signal                | Width | Description                     |
|-----------------------|-------|---------------------------------|
| dbg_bridge_pending    | 1     | Latency bridge has pending read |
| dbg_bridge_out_val_id | 1     | Latency bridge output valid     |

**Use Case:** Diagnose bridge backpressure issues

## Timing

### Critical Paths

**Space Free Output:** - COMBINATIONAL path from allocation controller - Registered output to break long paths to read engine - 1-cycle latency in space reporting (acceptable, conservative)

```
`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
)
```

**Data Available Output:** - COMBINATIONAL from drain controller + bridge occupancy - Immediate update (no registration)

```
assign wr_drain_data_avail = drain_data_available +
SCW'(bridge_occupancy);
```

### Resource Utilization

**Per Channel:** -  $1 \times$  FIFO ( $\text{SRAM\_DEPTH} \times \text{DATA\_WIDTH}$  bits) -  $2 \times$  Virtual FIFOs (allocation + drain controllers) - pointer logic only -  $1 \times$  Latency bridge ( $5 \times \text{DATA\_WIDTH}$  bits for skid buffer)

**Example (512-bit data, 512-deep FIFO):** - FIFO:  $512 \times 512 = 262,144$  bits = 32 KB - Controllers: ~100 FFs (pointers + control) - Bridge:  $5 \times 512 = 2,560$  bits = 320 bytes

**Total per channel:** ~32 KB RAM + minimal logic

**For 8 channels:** ~256 KB RAM

### Related Modules

- **Parent:** `sram_controller.sv` - Instantiates 8 units
- **Components:** `stream_alloc_ctrl.sv`, `gaxi_fifo_sync.sv`, `stream_latency_bridge.sv`
- **Users:** `axi_read_engine.sv` (allocation), `axi_write_engine.sv` (drain)

### References

- **Allocation Controller:** `stream_alloc_ctrl.md`
- **Drain Controller:** `stream_drain_ctrl.md`
- **Latency Bridge:** `stream_latency_bridge.md`

- **FIFO:** gaxi\_fifo\_sync from framework

## Stream Latency Bridge Specification

**Module:** stream\_latency\_bridge.sv **Purpose:** Simple Latency-1 Bridge with Skid Buffer for Backpressure **Version:** 0.90 **Date:** 2025-11-22 **Author:** RTL Design Sherpa Project

---

### 1. Overview

The Stream Latency Bridge provides a simple, efficient interface between a registered FIFO (with 1-cycle read latency) and a consumer. It uses minimal glue logic combined with a small skid buffer to handle backpressure without complex ready calculations.

#### 1.1 Key Features

- **1-Cycle Latency Compensation:** Handles registered FIFO output delay
- **Skid Buffer:** 4-deep FIFO absorbs consumer backpressure
- **Simple Design:** Minimal logic - easy to understand and verify
- **Full Throughput:** Maintains maximum data rate when consumer is ready
- **Occupancy Tracking:** Provides beat count for flow control visibility

#### 1.2 Block Diagram



*Block Diagram*

---

## 2. Design Rationale

### 2.1 Problem Statement

**Registered FIFO Read Latency:** - Cycle 0: Assert drain signal - Cycle 1: Data arrives from FIFO

**Challenge:** Consumer needs data immediately (no cycle delay), but FIFO has 1-cycle latency.

### 2.2 Solution Approach

**Two-Part Design:**

#### 1. Glue Logic (1 Flop):

- Tracks FIFO drain in progress
- When drain asserted → data will arrive next cycle
- Flop ( $r\_drain\_ip$ ) indicates “data in flight”

#### 2. Skid Buffer (4-Deep FIFO):

- Absorbs all backpressure complexity
- Provides elastic buffering between FIFO and consumer
- Handles consumer  $m\_ready$  variations

**Why This Works:** - Glue logic compensates for FIFO 1-cycle latency - Skid buffer provides backpressure absorption - No complex ready calculations needed - Clean separation of concerns

---

## 3. Architecture Details

### 3.1 Data Flow Sequence

**Normal Operation (No Backpressure):**

Cycle 0: FIFO has data ( $s\_valid=1$ ), skid has room ( $skid\_ready=1$ )  
→ Assert drain:  $w\_drain\_fifo = s\_valid \&& skid\_ready = 1$   
→ Flop stores:  $r\_drain\_ip \leq 1$

Cycle 1: Data arrives from FIFO ( $s\_data$  stable)  
→ Push to skid:  $skid\_wr\_valid = r\_drain\_ip = 1$   
→ Skid accepts:  $skid\_wr\_data = s\_data$

Cycle 2+: Consumer drains skid at its own pace  
→ Backpressure isolated to skid buffer

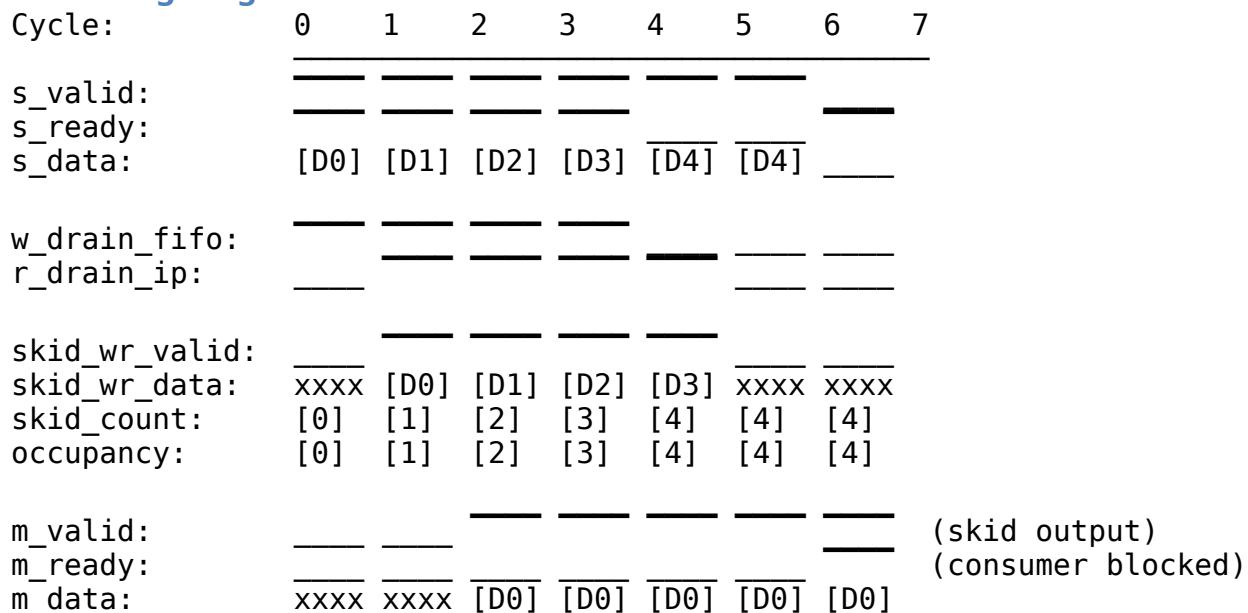
## With Consumer Backpressure:

Cycle 0-3: Fill skid buffer (4 beats)  
→ occupancy increases:  $0 \rightarrow 1 \rightarrow 2 \rightarrow 3 \rightarrow 4$

Cycle 4: Skid full (occupancy=4)  
→ `skid_wr_ready = 0`  
→ `s_ready = 0` (block FIFO drain)  
→ `r_drain_ip = 0` (no new data in flight)

Cycle 5+: Consumer starts draining (`m_ready=1`)  
→ Skid empties: occupancy decreases  
→ `skid_wr_ready = 1` when occupancy < 4  
→ `s_ready = 1` (resume FIFO drain)

## 3.2 Timing Diagram



### Notes:

- Cycle 0-3: Data flows through (`s_valid && s_ready`)
- Cycle 4: Skid full (occupancy=4) → `s_ready=0`
- Cycle 5-6: Backpressure active (`m_ready=0`)
- Cycle 7: Consumer ready → drain starts

## 4. Interface Specification

### 4.1 Parameters

```
parameter int DATA_WIDTH = 64, // Data path width
parameter int SKID_DEPTH = 4, // Skid buffer depth (2-4
 recommended)
```

```
parameter int DW = DATA_WIDTH // Internal alias
```

**Recommended SKID\_DEPTH:** - Minimum: 2 (minimal buffering) - Typical: 4 (good balance) - Maximum: 8 (excessive for most use cases)

---

## 4.2 Port List

### Clock and Reset

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

### Upstream Interface (from Registered FIFO)

| Signal  | Direction | Width      | Description                           |
|---------|-----------|------------|---------------------------------------|
| s_valid | input     | 1          | FIFO has data available               |
| s_ready | output    | 1          | Ready to drain FIFO                   |
| s_data  | input     | DATA_WIDTH | Data from FIFO (stable after 1 cycle) |

### 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    | DATA_WIDTH | 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     | Drain in progress flag        |
| dbg_r_out_val_id | output    | 1     | Data in flight to skid buffer |

## 4.3 Clock and Reset

```
input logic clk, // System clock
input logic rst_n // Active-low async reset
```

## 4.3 Upstream Interface (from Registered FIFO)

```
input logic s_valid, // FIFO has data available
output logic s_ready, // Ready to drain FIFO
input logic [DW-1:0] s_data // Data from FIFO (stable after
1 cycle)
```

**Key Behavior:** - s\_valid && s\_ready on Cycle N → data arrives on s\_data at Cycle N+1 - s\_ready deasserts when skid buffer full (occupancy=SKID\_DEPTH)

## 4.4 Downstream Interface (to Consumer)

```
output logic m_valid, // Data available for consumer
input logic m_ready, // Consumer ready to accept data
output logic [DW-1:0] m_data // Data to consumer
```

**Standard AXI-Style Handshake:** - Data transfers when m\_valid && m\_ready - m\_valid remains asserted until consumer asserts m\_ready

## 4.5 Status Interface

```
output logic [2:0] occupancy // Beats in skid buffer (0-4 for
SKID_DEPTH=4)
```

**Occupancy Meaning:** - 0: Bridge empty, no data buffered - 1-3: Partial fill, data flowing - 4: Full (for SKID\_DEPTH=4), backpressure active

**Note:** Changed from original design (which counted in-flight + skid) to strict skid buffer count for simplicity.

## 4.6 Debug Interface

```
output logic dbg_r_pending, // Debug: r_drain_ip state
output logic dbg_r_out_valid // Debug: m_valid state
```

---

## 5. Implementation Details

### 5.1 Glue Logic

**Purpose:** Track FIFO drain in progress

```
// Decide to drain FIFO
wire w_drain_fifo = s_valid && skid_wr_ready;

// Flop tracks 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; // Data arriving next cycle
 end
end

// Push to skid when data arrives
assign skid_wr_valid = r_drain_ip;
assign skid_wr_data = s_data; // Stable (registered FIFO output)
```

### 5.2 Backpressure Logic

**s\_ready Calculation:**

```
assign s_ready = skid_wr_ready; // Direct passthrough from skid buffer
```

**Original (more complex):**

```
assign s_ready = r_drain_skid ? skid_wr_ready :
 (occupancy <= 3) ? skid_wr_ready : 1'b0;
```

**Simplified:** s\_ready directly reflects skid buffer capacity.

### 5.3 Skid Buffer Instantiation

```
gaxi_fifo_sync #(
 .MEM_STYLE(FIFO_AUTO), // Let tool decide (SRL for small
 depth),
 .REGISTERED(0), // No extra latency needed
 .DATA_WIDTH(DATA_WIDTH),
 .DEPTH(SKID_DEPTH)
) u_skid_buffer (
 .axi_aclk (clk),
 .axi_aresetn (rst_n),
 // Write port (from glue logic)
 .wr_valid (skid_wr_valid),
```

```

 .wr_ready (skid_wr_ready),
 .wr_data (skid_wr_data),

 // Read port (to consumer)
 .rd_valid (m_valid),
 .rd_ready (m_ready),
 .rd_data (m_data),

 // Status
 .count (skid_count)
);

```

#### 5.4 Occupancy Tracking

```
// Occupancy = beats in skid buffer (strict count)
assign occupancy = skid_count[2:0];
```

**Original Design (commented out):**

```
// assign occupancy = 3'(r_drain_ip) + skid_count;
// Max = 1 (in flight) + SKID_DEPTH (in buffer) = 5
```

**Current Design:** Simplified to only count skid buffer occupancy (max = SKID\_DEPTH).

---

## 6. Reset Behavior

**On Reset (rst\_n deasserted):** - r\_drain\_ip = 0 (no data in flight) - Skid buffer empties (internal FIFO reset) - occupancy = 0 - m\_valid = 0 (no output data) - s\_ready = 1 (ready to accept new data)

**Post-Reset:** - Bridge immediately ready to accept data from FIFO - First drain: Cycle 0 (drain asserted) → Cycle 1 (data to skid) → Cycle 2 (data to consumer)

---

## 7. Operational Examples

### 7.1 Example: Full Throughput (No Backpressure)

**Scenario:** Consumer always ready ( $m_{ready}=1$ )

Cycle 0:  $s_{valid}=1$ ,  $s_{ready}=1 \rightarrow$  drain FIFO  
 Cycle 1:  $s_{data}$  arrives  $\rightarrow$  push to skid (occupancy=1)  
 Cycle 2: Consumer drains ( $m_{valid} \&& m_{ready}$ )  $\rightarrow$  occupancy=0

Result: 1 beat/cycle throughput (maximum)

## 7.2 Example: Consumer Backpressure

**Scenario:** Consumer stalls (`m_ready=0`)

Cycles 0-3: Fill skid (occupancy: 0→1→2→3→4)  
Cycle 4: Skid full → `s_ready=0` (block FIFO)  
Cycle 5-10: Consumer blocked (`m_ready=0`)  
Cycle 11: Consumer ready (`m_ready=1`) → drain 1 beat (occupancy=3)  
Cycle 12: `s_ready=1` → resume FIFO drain

Result: Skid buffer absorbed backpressure, no data loss

## 7.3 Example: Occupancy Tracking

**Monitoring Data Flow:**

```
Testbench monitoring
async def monitor_occupancy(dut):
 while True:
 await RisingEdge(dut.clk)
 occ = int(dut.occupancy.value)
 if occ > 0:
 print(f"@{get_sim_time('ns')} Occupancy: {occ}")
```

**Expected Pattern:** - Rapid fill: 0 → 4 (4 cycles if no consumer drain) - Steady state: 1-2 (balanced producer/consumer) - Backpressure: 4 (consumer stalled)

---

## 8. Verification Strategy

### 8.1 Test Scenarios

**Implemented Tests (`test_stream_latency_bridge.py`):**

1. **Basic Transfer:** Single beat, no backpressure
2. **Streaming:** Continuous data flow, consumer always ready
3. **Backpressure:** Fill skid buffer, verify `s_ready` deassertion
4. **Occupancy Tracking:** Verify occupancy matches expected beat count

**Test Parameters:** - Data widths: 256-bit, 512-bit - Skid depth: 4 (default)

### 8.2 Coverage Metrics

**Functional Coverage:** - Empty bridge (occupancy=0) - Partial fill (occupancy=1-3) - Full skid (occupancy=4) - Backpressure assertion (`s_ready=0`) - Backpressure release (`s_ready=1` after drain) - Continuous streaming (no gaps)

**Code Coverage:** - Line coverage: 100% (simple design) - Branch coverage: 100% - Toggle coverage: >95%

### 8.3 Key Assertions

```
// Occupancy bounds
assert property (@(posedge clk) occupancy <= SKID_DEPTH);

// Backpressure correctness
assert property (@(posedge clk)
 (occupancy == SKID_DEPTH) |-> !s_ready
);

// Data integrity (no drops)
assert property (@(posedge clk)
 (s_valid && s_ready) |=> (skid_count > $past(skid_count)) ||
m_ready
);
```

---

## 9. Performance Characteristics

### 9.1 Latency

**Best Case (Skid Empty):** - FIFO drain → Data to consumer: **2 cycles** - Cycle 0: Drain asserted - Cycle 1: Data to skid - Cycle 2: Data to consumer

**Worst Case (Skid Full):** - Wait for consumer drain + 2 cycles: **Variable (depends on consumer)**

### 9.2 Throughput

**Maximum:** 1 beat/cycle (when consumer always ready)

**Sustained:** Depends on consumer `m_ready` pattern - If consumer accepts 1 beat every 2 cycles → 0.5 beats/cycle average - Skid buffer smooths variations

### 9.3 Resource Usage

**Logic:** - 1 flop (`r_drain_ip`) - Minimal combinational logic (AND gates for ready)

**Memory:** - Skid buffer: `SKID_DEPTH × DATA_WIDTH` bits - Example:  $4 \times 512 = 2048$  bits (2 Kb)

**Total:** Extremely lightweight (dominated by skid FIFO)

---

## 10. Integration with STREAM

### 10.1 SRAM Controller Integration

**Use Case:** Bridge between SRAM read output and AXI write engine

```
// SRAM controller read interface (registered output)
logic [NUM_CHANNELS-1:0] sram_rd_valid;
logic [NUM_CHANNELS-1:0][DATA_WIDTH-1:0] sram_rd_data;
logic [NUM_CHANNELS-1:0] sram_rd_req;

// Latency bridge per channel
for (genvar ch = 0; ch < NUM_CHANNELS; ch++) begin : gen_bridges
 stream_latency_bridge #(
 .DATA_WIDTH(DATA_WIDTH),
 .SKID_DEPTH(4)
) u_bridge (
 .clk (clk),
 .rst_n (rst_n),

 // From SRAM controller
 .s_valid (sram_rd_valid[ch]),
 .s_ready (sram_rd_req[ch]),
 .s_data (sram_rd_data[ch]),

 // To AXI write engine
 .m_valid (wr_data_valid[ch]),
 .m_ready (wr_data_ready[ch]),
 .m_data (wr_data[ch]),

 .occupancy (bridge_occupancy[ch])
);
end
```

### 10.2 Descriptor Engine Integration

**Use Case:** Buffer descriptor output

```
stream_latency_bridge #(
 .DATA_WIDTH(256), // Descriptor width
 .SKID_DEPTH(2) // Small buffer for descriptors
) u_desc_bridge (
 .clk (clk),
 .rst_n (rst_n),
 .s_valid (desc_valid),
 .s_ready (desc_ready),
 .s_data (descriptor),
 .m_valid (scheduler_valid),
 .m_ready (scheduler_ready),
```

```
.m_data (scheduler_desc),
.occupancy () // Unused for descriptors
);
```

---

## 11. References

- **STREAM PRD:** projects/components/stream/PRD.md
  - **Source RTL:**  
projects/components/stream/rtl/stream\_fub/stream\_latency\_bridge.sv
  - **Tests:**  
projects/components/stream/dv/tests/fub\_tests/test\_stream\_latency\_bridge.py
  - **Testbench:**  
projects/components/stream/dv/tbclasses/stream\_latency\_bridge\_tb.py
  - **gaxi\_fifo\_sync:** rtl/amba/gaxi/gaxi\_fifo\_sync.sv
  - **FIFO Defs:** rtl/amba/includes/fifo\_defs.svh
- 

**Document Version:** 0.90 **Last Updated:** 2025-11-22 **Review Status:** Pre-release  
**Next Review:** After integration testing

---

## Appendix A: Signal Summary Table

| Signal                        | Direction | Width      | Purpose                |
|-------------------------------|-----------|------------|------------------------|
| <b>Clock/Reset</b>            |           |            |                        |
| clk                           | input     | 1          | System clock           |
| rst_n                         | input     | 1          | Active-low<br>reset    |
| <b>Upstream<br/>Interface</b> |           |            |                        |
| s_valid                       | input     | 1          | FIFO has data          |
| s_ready                       | output    | 1          | Ready to drain<br>FIFO |
| s_data                        | input     | DATA_WIDTH | Data from<br>FIFO      |
| <b>Downstream</b>             |           |            |                        |

| Signal                        | Direction | Width      | Purpose                     |
|-------------------------------|-----------|------------|-----------------------------|
| <b>Interface</b>              |           |            |                             |
| <code>m_valid</code>          | output    | 1          | Data available for consumer |
| <code>m_ready</code>          | input     | 1          | Consumer ready              |
| <code>m_data</code>           | output    | DATA_WIDTH | Data to consumer            |
| <b>Status</b>                 |           |            |                             |
| <code>occupancy</code>        | output    | 3          | Beats in skid buffer (0-4)  |
| <b>Debug</b>                  |           |            |                             |
| <code>dbg_r_pending</code>    | output    | 1          | Data in flight indicator    |
| <code>dbg_r_out_val_id</code> | output    | 1          | Output valid indicator      |

## End of Specification

## Stream Drain Controller

**Module:** stream\_drain\_ctrl.sv **Category:** FUB (Functional Unit Block) **Parent:** sram\_controller\_unit.sv

### Overview

The `stream_drain_ctrl` module is a **virtual FIFO without data storage** that tracks data availability using FIFO pointer logic. It provides pre-reservation support for AXI write engines to prevent underflow when draining data from the FIFO.

### What Makes This a "Virtual FIFO"

**Virtual FIFO:** - Has write pointer (data entry pointer) - Has read pointer (drain reservation pointer) - Calculates full/empty/data\_available - **NO data storage** - only pointer arithmetic

**Use Case:** Reserve FIFO data BEFORE AXI write burst starts

## The Drain Problem

### Without Drain Controller

**Problem:** Race condition between data check and drain

Cycle 0: Write engine checks FIFO occupancy  
data\_available = 100 beats → OK to issue 16-beat burst

Cycle 5: AW handshake completes, AXI write starts

Cycle 10: Meanwhile, read engine allocates 90 beats in FIFO  
→ Only 10 beats of actual data remain (NOT ENOUGH!)

Cycle 15: AXI write engine tries to drain 16 beats  
→ UNDERFLOW! Only 10 beats available

### With Drain Controller

**Solution:** Reserve data when issuing AW, not when draining

Cycle 0: Write engine checks data\_available = 100 beats  
Reserve 16 beats:  
wr\_drain\_req = 1, wr\_drain\_size = 16  
→ data\_available = 84 beats (reserved!)

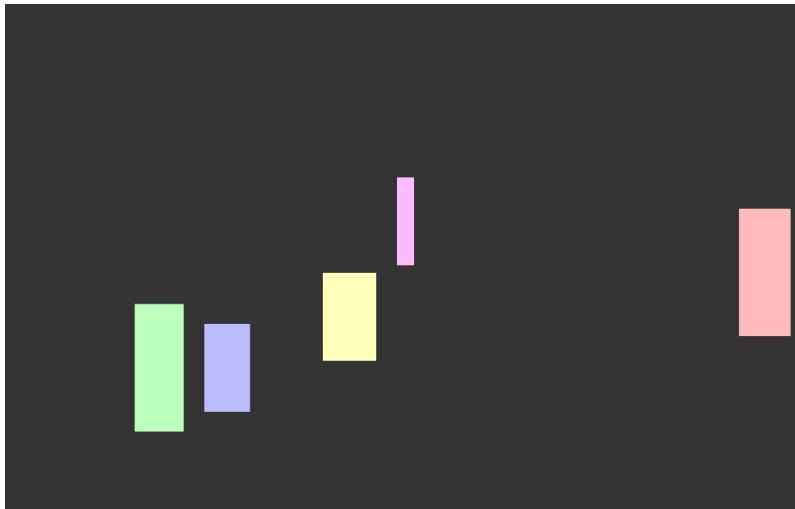
Cycle 5: AW handshake completes

Cycle 10: Read engine checks data\_available = 84 beats  
→ Sees reduced data, won't over-allocate

Cycle 15: AXI write data drains from FIFO  
→ Data was reserved, guaranteed available

## Architecture

### Two-Pointer System



*Diagram*

→

**Write Pointer (Data Entry Pointer):** - Advances when data **enters FIFO** (FIFO write handshake) - Single-beat increment - Represents “data added”

**Read Pointer (Drain Reservation Pointer):** - Advances when data is **reserved** (drain request) - Variable-size increment (wr\_drain\_size) - Represents “data reserved”

**Data Calculation:**

```
data_available = wr_ptr - rd_ptr
```

### NORMAL FIFO Naming Convention

**YES GOOD NEWS:** Drain controller uses STANDARD FIFO naming (unlike allocation controller)!

**Standard FIFO:** | Signal | Meaning | |——|——| | wr\_\* | Write data → Increment occupancy → data\_available increases | | rd\_\* | Read data → Decrement occupancy → data\_available decreases |

**Drain Controller:** | Signal | Meaning | |——|——| | wr\_\* | Data **enters FIFO** → Increment occupancy → data\_available **increases** | | rd\_\* | Reserve data → Decrement availability → data\_available **decreases** |

This matches normal FIFO intuition!

```
// Write side = Data enters FIFO
.wr_valid (axi_rd_sram_valid && axi_rd_sram_ready), // FIFO write
handshake

// Read side = Reserve data for upcoming write burst
.rd_valid (wr_drain_req),
.rd_size (wr_drain_size) // Variable size (e.g., 16 beats)
```

## Parameters

| Parameter            | Type | Default | Description                                   |
|----------------------|------|---------|-----------------------------------------------|
| DEPTH                | int  | 512     | Virtual FIFO depth (must match physical FIFO) |
| ALMOST_WR_MAR<br>GIN | int  | 1       | Almost full threshold                         |
| ALMOST_RD_MAR<br>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 (Data Enters FIFO)

| Signal   | Direction | Width | Description          |
|----------|-----------|-------|----------------------|
| wr_valid | input     | 1     | Data written to FIFO |
| wr_ready | output    | 1     | Not full (!wr_full)  |

### 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 ( $\neg \text{rd\_empty}$ ) |

### Status Outputs

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

## Interfaces

### Write Interface (Data Enters FIFO)

| Signal   | Direction | Width | Description                         |
|----------|-----------|-------|-------------------------------------|
| wr_valid | Input     | 1     | Data written to FIFO                |
| wr_ready | Output    | 1     | Not full ( $\neg \text{wr\_full}$ ) |

### Usage:

```
// Connect to FIFO write handshake
assign wr_valid = (axi_rd_sram_valid && axi_rd_sram_ready);
```

## Read Interface (Drain Requests)

| Signal   | Direction | Width | Description                |
|----------|-----------|-------|----------------------------|
| rd_valid | Input     | 1     | Request to drain           |
| rd_size  | Input     | 8     | Number of entries to drain |
| rd_ready | Output    | 1     | Data available (!rd_empty) |

### Usage:

```
// Write engine reserves data before issuing AW
wr_drain_req = (data_check_ok && !aw_pending);
wr_drain_size = cfg_axi_wr_xfer_beats;
```

## Status Outputs

| Signal          | Direction | Width | Description               |
|-----------------|-----------|-------|---------------------------|
| data_available  | Output    | AW+1  | Available unreserved data |
| wr_full         | Output    | 1     | FIFO full                 |
| wr_almost_full  | Output    | 1     | Almost full               |
| rd_empty        | Output    | 1     | No data available         |
| rd_almost_empty | Output    | 1     | Almost empty              |

**Note:** data\_available is the most important output - used by write engine for data checking.

## Operation

### Drain Flow

#### Step 1: Data Enters FIFO

```
// AXI read data arrives, enters FIFO
axi_rd_sram_valid = 1, axi_rd_sram_ready = 1

// Drain controller sees write
```

```

wr_valid = 1
r_wr_ptr_bin <= r_wr_ptr_bin + 1
data_available increases by 1

```

### Step 2: Write Engine Checks Data

```

// Write engine checks data availability
if (data_available >= cfg_axi_wr_xfer_beats) begin
 // Data OK, proceed to reservation
end

```

### Step 3: Reserve Data

```

wr_drain_req = 1'b1;
wr_drain_size = 8'd16; // Reserve 16 beats

// Next cycle: rd_ptr advances
r_rd_ptr_bin <= r_rd_ptr_bin + 16;
data_available decreases by 16

```

### Step 4: AXI Write Executes

```

// Some cycles later, AW handshake completes
m_axi_awvalid = 1, m_axi_awready = 1
// AXI write starts

```

### Step 5: Data Drains from FIFO

```

// Write engine drains data from FIFO
axi_wr_sram_valid = 1, axi_wr_sram_ready = 1
// Data exits FIFO → Actual drain happens
// Drain controller already accounted for this via reservation

```

## Pointer Arithmetic

### Write Pointer (Data Entry):

```

// Single-beat increment (uses counter_bin utility)
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)
);

```

### Read Pointer (Drain Reservation):

```

// Variable-size increment
if (w_read && !r_rd_empty) begin
 r_rd_ptr_bin <= r_rd_ptr_bin + (AW+1)'(rd_size);
end

```

### Data Calculation:

```

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

// Data available = occupancy
data_available = w_count;

```

## Timing Behavior

### Data Entry Latency

**Data entry is IMMEDIATE (combinational + 1 cycle):**

```

Cycle N: axi_rd_sram_valid = 1, axi_rd_sram_ready = 1
 → wr_valid = 1
Cycle N+1: r_wr_ptr_bin = old_value + 1
 data_available = old_value + 1

```

**Write engine sees updated data\_available on next cycle.**

### Drain Reservation Latency

**Reservation is IMMEDIATE (combinational + 1 cycle):**

```

Cycle N: wr_drain_req = 1, wr_drain_size = 16
Cycle N+1: r_rd_ptr_bin = old_value + 16
 data_available = old_value - 16

```

**Read engine sees updated data\_available on next cycle.**

## Integration Example

### In sram\_controller\_unit.sv

```

stream_drain_ctrl #(
 .DEPTH(SD),
 .REGISTERED(1)
) u_drain_ctrl (
 .axi_aclk (clk),
 .axi_arstn (rst_n),

 // DATA WRITTEN (increment occupancy)
 .wr_valid (axi_rd_sram_valid && axi_rd_sram_ready), //
FIFO write
 .wr_ready (), // Unused

```

```

// DRAIN REQUEST (reserve data for upcoming write)
.rd_valid (wr_drain_req), // From write engine
.rd_size (wr_drain_size), // Burst size
.rd_ready (), // Unused (data check
uses data_available)

// Data tracking
.data_available (drain_data_available), // FIFO-only data
count

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

// CRITICAL: Add latency bridge occupancy!
// Drain controller tracks FIFO only, not latency bridge buffered data
assign wr_drain_data_avail = drain_data_available +
SCW'(bridge_occupancy);

```

### Why Add Bridge Occupancy:

The drain controller only sees data in the FIFO, not data buffered in the latency bridge. The latency bridge can hold up to 5 beats (1 in-flight + 4 in skid buffer).

Example:

```

FIFO count (drain_data_available): 100 beats
Bridge occupancy: 3 beats
Total available to write engine: 100 + 3 = 103 beats

```

**Without this addition, write engine underestimates available data!**

### Debug Support

#### Display Statements

Data Entry:

```

$display("DRAIN_CTRL @ %t: data written, wr_ptr: %0d -> %0d,
data_available will be %0d",
 $time, r_wr_ptr_bin, w_wr_ptr_bin_next,
 w_wr_ptr_bin_next - r_rd_ptr_bin);

```

**No explicit drain display** - drain requests are application-level, not controller-level events.

## Waveform Analysis

**Key Signals to Monitor:** - r\_wr\_ptr\_bin - Data entry pointer - r\_rd\_ptr\_bin - Drain reservation pointer - data\_available - Available unreserved data - axi\_rd\_sram\_valid, axi\_rd\_sram\_ready - FIFO write handshake - wr\_drain\_req, wr\_drain\_size - Drain requests

## Common Issues

### Issue 1: Data Available Too Low

**Symptom:** Write engine sees less data than expected

**Root Cause:** Bridge occupancy not added

**Wrong:**

```
assign wr_drain_data_avail = drain_data_available; // Missing bridge data!
```

**Correct:**

```
assign wr_drain_data_avail = drain_data_available + bridge_occupancy;
```

### Issue 2: Underflow Despite Drain Controller

**Symptom:** FIFO underflows even with drain controller

**Root Cause:** Write engine drains without reservation

**Wrong:**

```
// Issue AW without checking data_available
m_axi_awvalid = 1; // No drain request!
```

**Correct:**

```
// Always reserve before issuing AW
if (data_available >= cfg_axi_wr_xfer_beats) begin
 wr_drain_req = 1'b1;
 wr_drain_size = cfg_axi_wr_xfer_beats;
 // THEN issue AW on next cycle
end
```

### Issue 3: Data Not Incrementing

**Symptom:** data\_available never increases

**Root Cause:** wr\_valid not connected to FIFO write handshake

**Wrong:**

```
.wr_valid (fifo_rd_valid) // FIFO read - WRONG side!
```

**Correct:**

```
.wr_valid (axi_rd_sram_valid && axi_rd_sram_ready) // FIFO write -
CORRECT!
```

## Comparison with Allocation Controller

| Aspect            | Drain Controller                | Allocation Controller           |
|-------------------|---------------------------------|---------------------------------|
| <b>Purpose</b>    | Reserve FIFO data               | Reserve FIFO space              |
| <b>User</b>       | AXI write engine                | AXI read engine                 |
| <b>Write side</b> | Data enters FIFO<br>(increment) | Allocation request (reserve)    |
| <b>Read side</b>  | Drain request (reserve)         | Data exits controller (release) |
| <b>Naming</b>     | SAME as normal FIFO             | OPPOSITE of normal FIFO         |
| <b>Output</b>     | data_available                  | space_free                      |
| <b>Bridge</b>     | YES (add                        | NO (space is absolute)          |
| <b>compensat</b>  | bridge_occupancy)               |                                 |
| <b>ion</b>        |                                 |                                 |

## Resource Utilization

**Per Instance:** -  $2 \times (\text{AW}+1)$ -bit counters (wr\_ptr, rd\_ptr) -  $1 \times (\text{AW}+1)$ -bit data\_available 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\_alloc\_ctrl.sv - Allocation-side flow control
- **Parent:** sram\_controller\_unit.sv - Instantiates drain controller
- **User:** axi\_write\_engine.sv - Checks wr\_drain\_data\_avail before issuing AW

## References

- **Allocation Controller:** stream\_alloc\_ctrl.md

- **SRAM Controller Unit:** sram\_controller\_unit.md
- **AXI Write Engine:** axi\_write\_engine.md
- **Latency Bridge:** stream\_latency\_bridge.md (for bridge\_occupancy signal)

## AXI Write Engine Specification

**Module:** axi\_write\_engine.sv **Location:**

projects/components/stream/rtl/fub/ **Status:** Implemented **Last Updated:**  
2025-11-21

---

### Overview

The AXI Write Engine is a high-performance multi-channel AXI4 write engine that manages write transactions from multiple independent channels to system memory. It features FIFO-based W-phase tracking, space-aware arbitration, streaming data path from SRAM controller, and support for pipelined operation.

### Key Features

- **Multi-channel support:** Arbitrates across NUM\_CHANNELS independent channels
- **FIFO-based W-phase tracking:** Single shared W-phase FIFO preserves AW command order
- **Per-channel B-phase FIFOs:** Handles out-of-order B responses correctly
- **Space-aware arbitration:** Only grants to channels with sufficient SRAM data available
- **Pre-allocation handshake:** Reserves SRAM data before issuing AXI AW command
- **Streaming pipeline:** No internal buffering - direct SRAM-to-AXI streaming
- **Channel ID tracking:** Encodes channel ID in AXI transaction ID and USER field
- **Pipelined operation:** Optional pipelining with outstanding transaction support
- **Completion feedback:** Reports burst completion back to schedulers
- **Bubble-free operation:** Continuous W transfers when multiple transactions pending

## Architecture Overview

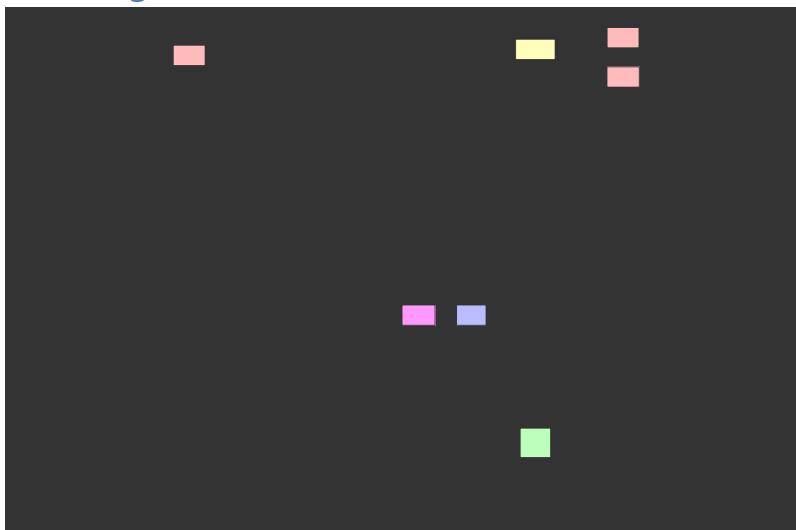
The write engine uses a dual-FIFO architecture optimized for AXI4 ordering requirements:

**W-Phase FIFO (Single Shared):** - AXI4 spec requires W-phase to be in-order with AW commands - Single FIFO preserves AW issue order across all channels - Pushed on AW handshake, popped when W-phase completes

**B-Phase FIFOs (Per-Channel):** - AXI4 allows B responses to arrive out-of-order - Separate FIFO per channel handles concurrent outstanding transactions - Pushed on AW handshake, popped when matching B response arrives - Stores burst length and last-descriptor flag for completion signaling

This architecture provides simple control logic, excellent scalability, and bubble-free operation.

## Block Diagram



Diagram

---

## Parameters

```
parameter int NUM_CHANNELS = 8; // Number of independent channels
parameter int ADDR_WIDTH = 64; // AXI address bus width
parameter int DATA_WIDTH = 512; // AXI data bus width
parameter int ID_WIDTH = 8; // AXI ID field width
parameter int USER_WIDTH = 1; // AXI USER field width
(parameter int)
parameter int SEG_COUNT_WIDTH = 8; // Width of space/count
```

```

signals
parameter int PIPELINE = 0; // 0: Single txn/channel,
1: Multiple outstanding
parameter int AW_MAX_OUTSTANDING = 8; // Max outstanding AW per
channel (PIPELINE=1)

// NEW PARAMETERS (November 2025)
parameter int W_PHASE_FIFO_DEPTH = 64; // W-phase transaction
FIFO depth (in-order with AW)
parameter int B_PHASE_FIFO_DEPTH = 16; // B-phase transaction
FIFO depth (out-of-order responses)

```

## Parameter Guidelines

**NUM\_CHANNELS:** - Typical: 2-8 channels - Must match scheduler array size - Affects arbiter complexity and area

**DATA\_WIDTH:** - Common: 128, 256, 512 bits - Must match SRAM controller data width - Determines AXI AWSIZE encoding

**PIPELINE:** - 0 = Non-pipelined: Wait for B response before next AW (simple, lower throughput) - 1 = Pipelined: Allow multiple outstanding AW per channel (higher throughput)

**AW\_MAX\_OUTSTANDING:** - Only used when PIPELINE=1 - Typical: 4-16 depending on memory latency - Higher values improve throughput but increase area

**W\_PHASE\_FIFO\_DEPTH:** - Default: 64 entries - Must be  $\geq$  total outstanding AW across all channels - Single shared FIFO, so size = NC  $\times$  max\_outstanding\_per\_channel - Deeper FIFO supports more concurrent channels but increases area

**B\_PHASE\_FIFO\_DEPTH:** - Default: 16 entries per channel - Must be  $\geq$  max outstanding transactions for a single channel - Per-channel FIFOs handle out-of-order B responses - Shallower than W-phase (only single-channel depth needed)

**USER\_WIDTH:** - Typically  $\$clog2(\text{NUM\_CHANNELS})$  to carry channel ID - Enables transaction tracking in debug/monitoring - Set to 1 if AXI USER not used

---

## 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 (applies to all channels) |

### Scheduler Interface

#### Per-Channel Write Requests:

| Signal                 | Direction | Width                        | Description                          |
|------------------------|-----------|------------------------------|--------------------------------------|
| sched_wr_val_id[ch]    | input     | NUM_CHAN NELS                | Channel requests write               |
| sched_wr_ready[ch]     | output    | NUM_CHAN NELS                | Engine ready for channel             |
| sched_wr_addr[ch]      | input     | NUM_CHAN NELS × ADDR_WIDTH H | Destination addresses per channel    |
| sched_wr_beats[ch]     | input     | NUM_CHAN NELS × 32           | Beats remaining to write per channel |
| sched_wr_burst_len[ch] | input     | NUM_CHAN NELS × 8            | Requested burst length per channel   |

#### Completion Feedback:

| Signal                   | Direction | Width              | Description                                  |
|--------------------------|-----------|--------------------|----------------------------------------------|
| sched_wr_done_strobe[ch] | output    | NUM_CHAN NELS      | Burst completed (pulsed 1 cycle per channel) |
| sched_wr_beats_done[ch]  | output    | NUM_CHAN NELS × 32 | Beats completed in burst per channel         |

| Signal                 | Direction | Width            | Description                                       |
|------------------------|-----------|------------------|---------------------------------------------------|
| sched_wr_err<br>or[ch] | output    | NUM_CHAN<br>NELS | Sticky error flag per<br>channel (bad B response) |

## SRAM Drain Interface

Pre-drain (reserves data before AW command):

| Signal                      | Direction | Width                                     | Description                                      |
|-----------------------------|-----------|-------------------------------------------|--------------------------------------------------|
| axi_wr_drain_req[ch]        | output    | NUM_CHAN<br>NELS                          | Channel requests to<br>reserve data              |
| axi_wr_drain_size[ch]       | output    | NUM_CHAN<br>NELS × 8                      | Beats to reserve per<br>channel                  |
| axi_wr_drain_data_avail[ch] | input     | NUM_CHAN<br>NELS ×<br>SEG_COUNT<br>_WIDTH | Data available after<br>reservations per channel |

## SRAM Read Interface

Data Stream (SRAM → W):

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

## AXI4 AW Channel (Write Address)

| Signal        | Direction | Width      | Description    |
|---------------|-----------|------------|----------------|
| m_axi_awvalid | output    | 1          | Address valid  |
| m_axi_awready | input     | 1          | Address ready  |
| m_axi_awid    | output    | ID_WIDTH   | Transaction ID |
| m_axi_awaddr  | output    | ADDR_WIDTH | Write address  |

| Signal        | Direction | Width | Description                          |
|---------------|-----------|-------|--------------------------------------|
| m_axi_awlen   | output    | 8     | Burst length - 1                     |
| m_axi_awsize  | output    | 3     | Burst size<br>(log2(bytes per beat)) |
| m_axi_awburst | output    | 2     | Burst type<br>(2'b01 = INCR)         |

#### AXI4 W Channel (Write Data)

| Signal       | Direction | Width             | Description                         |
|--------------|-----------|-------------------|-------------------------------------|
| m_axi_wvalid | output    | 1                 | Write data valid                    |
| m_axi_wready | input     | 1                 | Write data ready                    |
| m_axi_wdata  | output    | DATA_WIDTH<br>H   | Write data                          |
| m_axi_wstrb  | output    | DATA_WIDTH<br>H/8 | Write strobe (byte enables)         |
| m_axi_wlast  | output    | 1                 | Last beat in burst                  |
| m_axi_wuser  | output    | USER_WIDTH<br>H   | Channel ID for transaction tracking |

#### AXI4 B Channel (Write Response)

| Signal       | Direction | Width    | Description                   |
|--------------|-----------|----------|-------------------------------|
| m_axi_bvalid | input     | 1        | Write response valid          |
| m_axi_bready | output    | 1        | Write response ready          |
| m_axi_bid    | input     | ID_WIDTH | Transaction ID                |
| m_axi_bresp  | input     | 2        | Write response (2'b00 = OKAY) |

#### Debug Interface

| Signal                      | Direction | Width            | Description                                           |
|-----------------------------|-----------|------------------|-------------------------------------------------------|
| dbg_wr_all_c<br>omplete[ch] | output    | NUM_CHAN<br>NELS | All writes complete per channel (no outstanding txns) |
| dbg_aw_trans<br>actions     | output    | 32               | Total AW transactions                                 |

| Signal      | Direction | Width | Description                  |
|-------------|-----------|-------|------------------------------|
|             |           |       | issued                       |
| dbg_w_beats | output    | 32    | Total W beats written to AXI |

## Interface

### Clock and Reset

```
input logic clk; // System clock
input logic rst_n; // Active-low async reset
```

### Configuration Interface

```
input logic [7:0] cfg_axi_wr_xfer_beats; // Transfer size in beats
```

The `cfg_axi_wr_xfer_beats` parameter sets the AXI burst length for all channels. This is converted to AXI AWLEN format (beats-1) internally.

### Scheduler Interface (Per-Channel)

```
// Request interface - one set per channel
input logic [NC-1:0] requests write sched_wr_valid; // Channel
output logic [NC-1:0] for channel sched_wr_ready; // Engine ready
input logic [NC-1:0][AW-1:0] addresses sched_wr_addr; // Destination
input logic [NC-1:0][31:0] remaining to write sched_wr_beats; // Beats
input logic [NC-1:0][7:0] burst length sched_wr_burst_len; // Requested

// Completion interface
output logic [NC-1:0] completed (1 cycle pulse) sched_wr_done_strobe; // Burst
output logic [NC-1:0][31:0] completed in burst sched_wr_beats_done; // Beats
output logic [NC-1:0] this channel sched_wr_error; // Error on
output logic [NC-1:0] outstanding transactions axi_wr_all_complete; // No
```

**Interface Protocol:** 1. Scheduler asserts `sched_wr_valid[i]` with address, beats remaining 2. Engine asserts `sched_wr_ready[i]` when it can accept the request 3. Handshake occurs when both valid and ready are asserted 4. Engine pushes transaction to W-phase FIFO and issues AXI AW command 5. When B response

arrives, engine pulses `sched_wr_done_strobe[i]` and provides `sched_wr_beats_done[i]`. Scheduler updates its remaining beat count and repeats if more beats remain

**Timing Note:** `sched_wr_ready` is registered and asserts 1 cycle after the channel becomes ready (not combinatorial from B response).

### SRAM Drain Interface

```
// Pre-allocation interface
output logic [NC-1:0] wr_drain_req; // Channel
requests to reserve data
output logic [NC-1:0][7:0] wr_drain_size; // Beats to
reserve
input logic [NC-1:0][SCW-1:0] wr_drain_data_avail; // Data available
after reservation

// Data drain interface (ID-based muxed output from SRAM controller)
input logic [NC-1:0] axi_wr_sram_valid; // Per-channel
data valid
output logic axi_wr_sram_drain; // Drain request
(consumer ready)
output logic [CIW-1:0] axi_wr_sram_id; // Channel ID
select for drain
input logic [DW-1:0] axi_wr_sram_data; // Data from
selected channel (muxed)
```

**SRAM Protocol:** 1. **Pre-allocation:** Before issuing AXI AW, engine checks `wr_drain_data_avail[ch]` to ensure sufficient data 2. **Space-aware arbitration:** Arbiter only grants to channels with `wr_drain_data_avail >= burst_len` 3. **Reservation:** When AW handshakes, engine pulses `wr_drain_req[ch]` with `wr_drain_size[ch] = burst_len` 4. **Data streaming:** During W phase, engine sets `axi_wr_sram_id = channel_id` and asserts `axi_wr_sram_drain` when ready 5. **Muxed data:** SRAM controller provides data from selected channel via `axi_wr_sram_data`

### AXI4 Master Interface

```
// AXI AW (Write Address) Channel
output logic [IW-1:0] m_axi_awid; // Transaction ID
contains channel ID
output logic [AW-1:0] m_axi_awaddr; // Write address
output logic [7:0] m_axi_awlen; // Burst length -
1
output logic [2:0] m_axi_awsize; // Burst size
(log2(bytes))
output logic [1:0] m_axi_awburst; // Burst type
(INCR=0b01)
output logic m_axi_awvalid;
```

```



```

**AXI ID Encoding:** - Lower bits [CIW-1:0] contain channel ID - Allows B responses to be routed back to correct channel - Enables MonBus packet generation with channel ID - Critical for multi-channel operation

**AXI USER Field:** - Contains channel ID: `m_axi_wuser = UW'(channel_id)` - Useful for transaction tracking and debugging - Can be monitored to verify correct channel assignment

### Debug Interface

```

output logic [31:0] dbg_aw_transactions; // Total AW
transactions issued
output logic [31:0] dbg_w_beats; // Total W beats
written

```

---

## Operation

### W-Phase and B-Phase FIFO Architecture

**Key Design Decision:** Separate W-phase and B-phase FIFOs based on AXI4 ordering requirements:

**W-Phase FIFO (Single Shared):** - Why single? AXI4 spec requires W-phase to be in-order with AW commands - **Structure:** {beats[7:0], channel\_id[CIW-1:0]} - **Depth:** `W_PHASE_FIFO_DEPTH` (default 64) - sized for total outstanding across all

channels - **Push**: On AW handshake for any channel - **Pop**: When W-phase completes (wlast) and loads next transaction

**B-Phase FIFOs (Per-Channel)**: - **Why per-channel?** AXI4 allows B responses to arrive out-of-order - **Structure**: {beats[7:0], last[0]} - **Depth**:

B\_PHASE\_FIFO\_DEPTH (default 16) per channel - sized for single channel outstanding - **Push (during AW handshake)**: When `m_axi_awvalid && m_axi_awready` occurs, the engine simultaneously: 1. Pushes {beats, channel\_id} to W-phase FIFO (for in-order W streaming) 2. Pushes {beats, last} to B-phase FIFO[channel\_id] (for out-of-order B tracking) 3. Sets last flag if this is the final burst for the descriptor - **Pop (during B response)**: When B response arrives with matching `m_axi_bid`, pop from corresponding channel FIFO

### Example Multi-Channel Flow:

| Cycle     | Action                   | W-Phase FIFO           | B-Phase   |
|-----------|--------------------------|------------------------|-----------|
| FIFO[CH0] | B-Phase FIFO[CH1]        |                        |           |
| -----     | -----                    | -----                  | -----     |
| 0         | AW CH0 handshake         | [{16, CH0}]            | [{16, 0}] |
| [ ]       |                          |                        |           |
| 1         | AW CH1 handshake         | [{16, CH0}, {32, CH1}] | [{16, 0}] |
| [{32, 0}] |                          |                        |           |
| 2         | W starts CH0             | [{32, CH1}] (pop CH0)  | [{16, 0}] |
| [{32, 0}] |                          |                        |           |
| 18        | W completes CH0, B pends | [{32, CH1}]            | [{16, 0}] |
| [{32, 0}] |                          |                        |           |
| 19        | W starts CH1             | [] (pop CH1)           | [{16, 0}] |
| [{32, 0}] |                          |                        |           |
| 50        | B arrives CH0            | []                     | [] (pop)  |
| [{32, 0}] |                          |                        |           |
| 51        | W completes CH1          | []                     | []        |
| [{32, 0}] |                          |                        |           |
| 85        | B arrives CH1            | []                     | []        |
| [ ] (pop) |                          |                        |           |

Notice W-phase is strictly in-order (CH0 then CH1), but B responses can arrive in any order.

### Arbitration and Channel Selection

The write engine uses a round-robin arbiter with space-aware masking:

```
// Space check: Only arbitrate channels with sufficient SRAM data
w_space_ok[i] = (wr_drain_data_avail[i] >= (cfg_axi_wr_xfer_beats << 1));
```

```

// Mask off channels without space or not requesting
w_arb_req_masked[i] = sched_wr_valid[i] && w_space_ok[i] && !
r_outstanding_limit[i];

// Round-robin arbitration among masked requests
arbiter_round_robin #(.WIDTH(NC)) u_arbiter (
 .i_req (w_arb_req_masked),
 .o_grant (w_arb_grant),
 .o_grant_valid(w_arb_grant_valid),
 .o_grant_id (w_arb_grant_id)
);

```

**Key Points:** - Only channels with 2x burst length of available data participate in arbitration  
- Outstanding transaction limits prevent channel from exceeding max outstanding count  
- Arbiter provides fair round-robin scheduling across ready channels

## W-Phase Control Logic

The W-phase uses FIFO peek/pop operations for simple and efficient control:

```

// W-phase control logic - FIFO-based, NO FSM!
always_ff @(posedge clk or negedge rst_n) begin
 if (!rst_n) begin
 r_w_beats_remaining <= '0;
 r_w_channel_id <= '0;
 r_w_active <= 1'b0;
 end else begin
 // Case 1: No active W transfer, check W-phase FIFO
 if (!r_w_active) begin
 if (!w_phase_txn_fifo_empty) begin
 // Start W transfer using info from FIFO head
 r_w_active <= 1'b1;
 r_w_channel_id <= w_phase_txn_fifo_dout.channel_id;
 r_w_beats_remaining <= w_phase_txn_fifo_dout.beats;
 end
 end

 // Case 2: Active W transfer, decrement beats on W handshake
 else if (m_axi_wvalid && m_axi_wready) begin
 r_w_beats_remaining <= r_w_beats_remaining - 8'd1;

 // Last beat of current transfer
 if (m_axi_wlast) begin
 // Check if more pending transactions in FIFO
 if (!w_phase_txn_fifo_empty) begin
 // Continue with next transaction from FIFO (NO

```

```

BUBBLE!)
 r_w_channel_id <=
w_phase_txn_fifo_dout.channel_id;
 r_w_beats_remaining <=
w_phase_txn_fifo_dout.beats;
 // r_w_active stays 1 → Continuous wvalid!
 end else begin
 // No more transactions in FIFO, go idle
 r_w_active <= 1'b0;
 end
 end
end
end
end

// Pop W-phase FIFO on wlast
assign w_phase_txn_fifo_rd = m_axi_wvalid && m_axi_wready &&
m_axi_wlast;

```

**Key Features:** - **Peek-before-pop:** Reads FIFO head to get next channel before popping - **Bubble-free:** If FIFO has next entry on wlast, loads immediately without idle cycle - **Simple control logic:** No state machine needed, just FIFO empty check

### Transaction Flow

1. Arbiter grants to channel with sufficient SRAM data
2. Engine latches AW parameters (address, length, channel ID)
3. Engine issues AXI AW transaction
4. When AW handshakes:
  - Push {beats, channel\_id} to W-phase FIFO
  - Push {beats, last} to B-phase FIFO[channel\_id]
  - Pulse wr\_drain\_req to reserve SRAM data
5. W-phase control:
  - If idle: Check W-phase FIFO, start if not empty
  - If active: Decrement on each W handshake
  - On wlast: Pop W-phase FIFO, load next if available (bubble-free!)
6. B-phase response:
  - Extract channel\_id from m\_axi\_bid
  - Pop from B-phase FIFO[channel\_id]
  - Pulse sched\_wr\_done\_strobe[channel\_id]
  - Provide sched\_wr\_beats\_done[channel\_id]

**Key Features:** - Simple control logic using FIFO peek/pop operations - Natural multi-channel support through FIFO ordering - Bubble-free operation (peek next entry before pop) - Excellent scalability (adding channels only increases FIFO depth)

## Outstanding Transaction Tracking

### PIPELINE=0:

```
// Simple boolean per channel (0 or 1 outstanding)
logic [NC-1:0] r_outstanding_limit;

// Set when AW issues, clear when B arrives
if (aw_handshake && channel_id == i) r_outstanding_limit[i] <= 1'b1;
if (b_handshake && bid[CIW-1:0] == i) r_outstanding_limit[i] <= 1'b0;
```

### PIPELINE=1:

```
// Counter per channel (0 to AW_MAX_OUTSTANDING)
logic [NC-1:0][MOW-1:0] r_outstanding_count;

// Increment on AW issue, decrement on B response
if (aw_handshake && channel_id == i)
 r_outstanding_count[i] <= r_outstanding_count[i] + 1;
if (b_handshake && bid[CIW-1:0] == i)
 r_outstanding_count[i] <= r_outstanding_count[i] - 1;

// Limit check
r_outstanding_limit[i] = (r_outstanding_count[i] >=
AW_MAX_OUTSTANDING);
```

## Address Management

The scheduler is responsible for address management:

```
// Engine uses address directly from scheduler
assign m_axi_awaddr = sched_wr_addr[r_aw_channel_id];
```

**Scheduler Responsibilities:** - Provides current write address in sched\_wr\_addr[ch] - Increments address after each AW handshake - Tracks total bytes written per descriptor

**Engine Responsibilities:** - Uses address directly from scheduler (no internal tracking) - Reports completion via sched\_wr\_done\_strobe and sched\_wr\_beats\_done - Scheduler updates address based on beats\_done feedback

## Completion Signal Behavior

**axi\_wr\_all\_complete[i]:** - Asserts when channel i has no outstanding transactions (B-phase FIFO empty) - **Clears on 0-to-nonzero transition** (when first AW issues after idle) - Stays low while transactions are outstanding - Used by scheduler to verify all transfers complete before moving to next descriptor

## Timing

### W-Phase Timing

| Cycle                          | AW0   | AW1   | W-FIFO              | W-Data      | B0    | B1 |
|--------------------------------|-------|-------|---------------------|-------------|-------|----|
| 0                              | REQ   | -     | []                  | IDLE        | -     | -  |
| 1                              | GRANT | REQ   | []                  | IDLE        | -     | -  |
| 2                              | VALID | -     | []                  | IDLE        | -     | -  |
| 3                              | DONE  | GRANT | [{16,CH0}]          | ACTIVE(CH0) | -     | -  |
| AW0→FIFO, W starts             |       |       |                     |             |       |    |
| 4                              | -     | VALID | [{16,CH0}]          | BEAT1(CH0)  | -     | -  |
| 5                              | -     | DONE  | [{16,CH0},{32,CH1}] | BEAT2(CH0)  | -     | -  |
| AW1→FIFO                       |       |       |                     |             |       |    |
| ..                             | ..    | ..    | ..                  | ..          | ..    | .. |
| 18                             | -     | -     | [{16,CH0},{32,CH1}] | LAST(CH0)   | -     | -  |
| wlast CH0                      | ..    | ..    | ..                  | ..          | ..    | .. |
| 19                             | -     | -     | [{32,CH1}]          | BEAT1(CH1)  | -     | -  |
| Pop CH0, load CH1 (NO BUBBLE!) |       |       |                     |             |       |    |
| ..                             | ..    | ..    | ..                  | ..          | ..    | .. |
| 50                             | -     | -     | [{32,CH1}]          | BEAT31(CH1) | VALID | -  |
| B0 arrives (async)             |       |       |                     |             |       |    |
| 51                             | -     | -     | []                  | LAST(CH1)   | DONE  | -  |
| wlast CH1, pop CH1             |       |       |                     |             |       |    |
| 52                             | -     | -     | []                  | IDLE        | -     | -  |
| VALID B1 arrives               |       |       |                     |             |       |    |

**Key Observation:** W-phase loads next channel on wlast (cycle 19) without bubble because it peeks FIFO before popping.

### Throughput Analysis

#### Single Channel Streaming:

| Memory Type   | B Latency | PIPELINE= |                    |                    |
|---------------|-----------|-----------|--------------------|--------------------|
|               |           | 0         | PIPELINE=1 (4 out) | PIPELINE=1 (8 out) |
| Embedded SRAM | 5-10      | 0.62      | 0.85               | 0.90               |
| DDR3          | 40-60     | 0.24      | 0.89               | 0.93               |
| DDR4          | 60-100    | 0.14      | 0.92               | 0.95               |
| PCIe Gen3     | 30-50     | 0.31      | 0.87               | 0.91               |

**Key Observation:** Pipelining benefit scales with memory latency - higher latency = greater improvement.

---

## Area and Performance

### Resource Estimates

| Mode                 | LUTs  | FFs   | BRAM | FMax (MHz) | Notes                         |
|----------------------|-------|-------|------|------------|-------------------------------|
| PIPELINE<br>=0, NC=4 | ~850  | ~650  | 0    | ~300       | FIFO overhead minimal         |
| PIPELINE<br>=0, NC=8 | ~1300 | ~1000 | 0    | ~280       |                               |
| PIPELINE<br>=1, NC=4 | ~1700 | ~1300 | 2    | ~250       | W+B FIFOs<br>(can be LUT RAM) |
| PIPELINE<br>=1, NC=8 | ~2600 | ~2000 | 3-4  | ~230       | W-FIFO shared, B-FIFO per-CH  |

**Notes:** - Estimates for DATA\_WIDTH=512, ADDR\_WIDTH=64 - W-phase FIFO: 1 BRAM (shared across channels) - B-phase FIFOs: 1-2 BRAM total (can use LUT RAM for NC≤4) - Control logic uses simple FIFO peek/pop operations - FMax depends on arbiter fan-out and SRAM interface timing

### Architecture Benefits

- **Control Logic:** Simple FIFO peek/pop operations instead of complex state machines
  - **Area:** Minimal overhead for multi-channel support
  - **Scalability:** Linear growth with channels ( $O(NC)$ )
  - **Bubble Cycles:** Zero cycles between bursts (peek-before-pop)
  - **Debugging:** FIFO contents directly visible in simulation
  - **Throughput:** Bubble-free operation maximizes memory bandwidth utilization
- 

## Testing

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

**Integration Tests:** - `test_stream_core.py` - Full system integration with multi-channel operation  
- `test_datapath_wr_test.py` - Write datapath with multiple schedulers  
- `b2b_multi_channel` test - Validates W-phase operation with 4 channels, back-to-back requests

**Test Scenarios:** 1. Single channel, single burst 2. Multi-channel concurrent writes (validates W-phase FIFO ordering) 3. Back-to-back requests (validates bubble-free operation) 4. Out-of-order B responses (validates per-channel B-phase FIFOs) 5. SRAM backpressure handling 6. AXI backpressure (wready deasserted) 7. Channel ID encoding verification 8. USER field verification 9. Completion signal timing (`axi_wr_all_complete`)

---

## Related Documentation

- **Scheduler:** `04_scheduler.md` - Interface contract and address tracking
  - **Read Engine:** `08_axi_read_engine.md` - Companion read engine
  - **SRAM Controller:** `09_sram_controller.md` - Data buffering
  - **Scheduler Group:** `01_scheduler_group.md` - Multi-channel integration
- 

## Revision History

| Date       | Version | Changes                                                                                                                                                                                                                                                                                                                                                |
|------------|---------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 2025-11-16 | 1.0     | Initial documentation matched to FSM-based RTL                                                                                                                                                                                                                                                                                                         |
| 2025-11-21 | 2.0     | Major update for FIFO-based W-phase refactoring:-<br>Single shared W-phase FIFO (not per-channel)- Per-channel B-phase FIFOs-<br>Bubble-free operation-<br>Added<br><code>W_PHASE_FIFO_DEPTH</code> ,<br><code>B_PHASE_FIFO_DEPTH</code><br>parameters- Updated block diagram and architecture description- Removed FSM-based control flow description |

**Last Updated:** 2025-11-21 (matched to current FIFO-based RTL implementation)

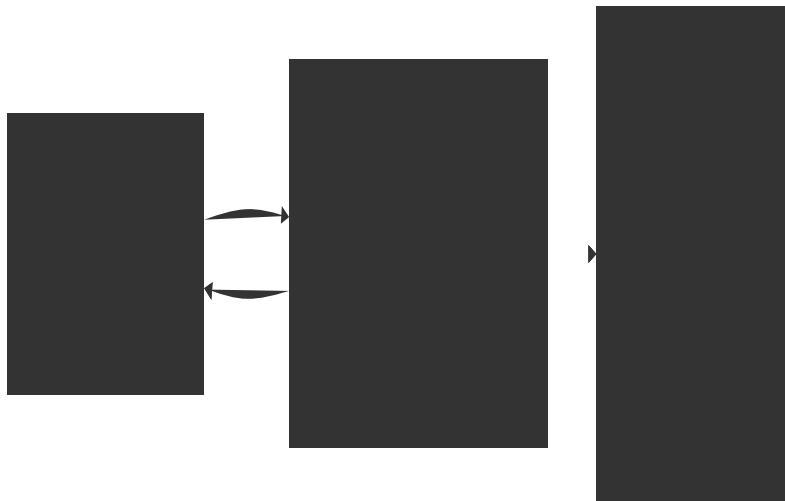
## APB-to-Descriptor Router (apbtodescr)

### Overview

The apbtodescr module is a lightweight address-decode router that connects the APB configuration slave to the descriptor engines' APB kick-off ports. It translates APB register writes into descriptor engine start commands.

**Key Characteristics:** - No internal registers - Pure combinational decode + FSM control - Parameterized base address - Flexible placement in address space - Back-pressure handling - Delays APB response if descriptor engine busy - 8-channel routing - Two registers per channel: LOW (buffered) + HIGH (triggers kick-off) - 64-bit descriptor addresses - Split across LOW/HIGH register pairs (0x00 through 0x3F)

### Block Diagram



*Block Diagram*

### Address Map

Relative to BASE\_ADDR parameter (default: 0x0000\_0000):

| Offset | Register          | Chann<br>el | Description                                 |
|--------|-------------------|-------------|---------------------------------------------|
| 0x00   | CH0_CTRL_LO<br>W  | 0           | Channel 0 descriptor address [31:0]         |
| 0x04   | CH0_CTRL_HI<br>GH | 0           | Channel 0 descriptor address [63:32] + kick |
| 0x08   | CH1_CTRL_LO<br>W  | 1           | Channel 1 descriptor address [31:0]         |
| 0x0C   | CH1_CTRL_HI<br>GH | 1           | Channel 1 descriptor address [63:32] + kick |
| 0x10   | CH2_CTRL_LO<br>W  | 2           | Channel 2 descriptor address [31:0]         |
| 0x14   | CH2_CTRL_HI<br>GH | 2           | Channel 2 descriptor address [63:32] + kick |
| 0x18   | CH3_CTRL_LO<br>W  | 3           | Channel 3 descriptor address [31:0]         |
| 0x1C   | CH3_CTRL_HI<br>GH | 3           | Channel 3 descriptor address [63:32] + kick |
| 0x20   | CH4_CTRL_LO<br>W  | 4           | Channel 4 descriptor address [31:0]         |
| 0x24   | CH4_CTRL_HI<br>GH | 4           | Channel 4 descriptor address [63:32] + kick |
| 0x28   | CH5_CTRL_LO<br>W  | 5           | Channel 5 descriptor address [31:0]         |
| 0x2C   | CH5_CTRL_HI<br>GH | 5           | Channel 5 descriptor address [63:32] + kick |
| 0x30   | CH6_CTRL_LO<br>W  | 6           | Channel 6 descriptor address [31:0]         |
| 0x34   | CH6_CTRL_HI<br>GH | 6           | Channel 6 descriptor address [63:32] + kick |
| 0x38   | CH7_CTRL_LO<br>W  | 7           | Channel 7 descriptor address [31:0]         |
| 0x3C   | CH7_CTRL_HI<br>GH | 7           | Channel 7 descriptor address [63:32] + kick |

### Address Decode:

```

relative_addr = apb_cmd_addr - BASE_ADDR
channel_id = relative_addr[5:3] // Bits [5:3] select channel 0-7
addr_type = relative_addr[2] // Bit [2]: 0=LOW, 1=HIGH
// Bits [1:0] ignored (word-aligned)

```

**Valid Range:** BASE\_ADDR + 0x00 to BASE\_ADDR + 0x3F (64 bytes, 16 registers)

## Write Transaction Flow

### Normal Write (No Back-pressure)

#### 64-bit Descriptor Address Write Sequence:

Software → APB Slave → apbtodescr → Descriptor Engine

Descriptor address: 0x0000\_0001\_8000\_0000 (64-bit)

Phase 1 - Write LOW 32 bits (buffered):

1. Software writes 0x8000\_0000 to BASE+0x00 (CH0\_CTRL\_LOW)
2. APB slave presents:
  - apb\_cmd\_valid = 1
  - apb\_cmd\_addr = BASE+0x00
  - apb\_cmd\_wdata = 0x8000\_0000
  - apb\_cmd\_write = 1
3. apbtodescr decodes:
  - channel\_id = 0, addr\_type = LOW
  - Buffers data internally, NO kick-off yet
4. APB transaction completes immediately  
Duration: 2-3 cycles

Phase 2 - Write HIGH 32 bits (triggers kick-off):

5. Software writes 0x0000\_0001 to BASE+0x04 (CH0\_CTRL\_HIGH)
6. apbtodescr decodes:
  - channel\_id = 0, addr\_type = HIGH
  - Combines buffered LOW + new HIGH → 64-bit address
  - FSM: IDLE → ROUTE
7. apbtodescr routes to descriptor engine:
  - desc\_apb\_valid[0] = 1
  - desc\_apb\_addr[0] = 0x0000\_0001\_8000\_0000 (64-bit)
8. Descriptor engine accepts (has space in skid buffer):
  - desc\_apb\_ready[0] = 1
9. apbtodescr completes:
  - FSM: ROUTE → RESPOND
  - apb\_rsp\_valid = 1
  - apb\_rsp\_error = 0
10. APB slave completes HIGH write transaction  
Duration: 3-4 cycles

Total Duration: 5-7 cycles (both writes)

## Write with Back-pressure

### 64-bit Write with Descriptor Engine Busy:

Software → APB Slave → apbtodescr → Descriptor Engine (busy)

Descriptor address: 0x0000\_0000\_9000\_0000 (64-bit)

Phase 1 - Write LOW 32 bits (always succeeds):

1. Software writes 0x9000\_0000 to BASE+0x08 (CH1\_CTRL\_LOW)
2. apbtodescr buffers data, completes immediately

Duration: 2-3 cycles

Phase 2 - Write HIGH 32 bits (blocks on busy descriptor engine):

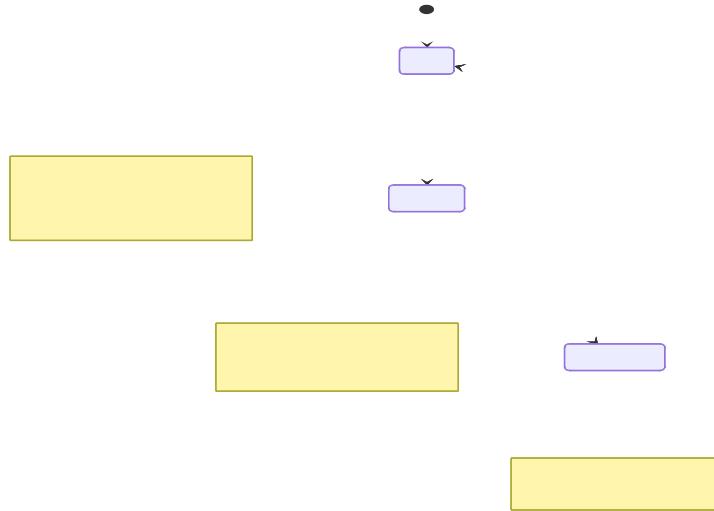
3. Software writes 0x0000\_0000 to BASE+0x0C (CH1\_CTRL\_HIGH)
4. apbtodescr decodes channel\_id = 1, enters ROUTE state
5. Descriptor engine NOT ready (skid buffer full):
  - desc\_apb\_ready[1] = 0
6. apbtodescr WAITS in ROUTE state:
  - desc\_apb\_valid[1] stays asserted with 64-bit address
  - apb\_rsp\_valid stays low
  - APB HIGH write transaction BLOCKED
  - Software waits (APB PREADY = 0)
7. ... N cycles later ...
8. Descriptor engine becomes ready:
  - desc\_apb\_ready[1] = 1
9. apbtodescr completes:
  - FSM: ROUTE → RESPOND
  - apb\_rsp\_valid = 1
10. APB HIGH write transaction completes

Duration: (2-3) + (3 + N) cycles

LOW write + HIGH write with N cycles back-pressure

## FSM States

### State Diagram



*FSM State Diagram*

IDLE IDLE → ROUTE: apb\_cmd\_valid && apb\_cmd\_write  
ROUTE → RESPOND:  
desc\_apb\_ready or r\_error  
RESPOND → IDLE: apb\_rsp\_ready

```
note right of IDLE
 apb_cmd_ready = 1
 Waiting for command
 Latch: channel_id, wdata, error
end note
```

```
note right of ROUTE
 desc_apb_valid[ch] = 1
 Waiting for descriptor engine
end note
```

```
note right of RESPOND
 apb_rsp_valid = 1
 apb_rsp_error = r_error
end note
```

-->

### ### State Descriptions

| State Condition | Description | Exit |
|-----------------|-------------|------|
|                 |             |      |

```

| IDLE | Waiting for APB command, ready to accept |
` apb_cmd_valid` |
| ROUTE | Routing to descriptor engine, waiting for ack |
` desc_apb_ready[ch]` or error |
| RESPOND | Sending APB response, waiting for ack |
` apb_rsp_ready` |

```

## ## Error Handling

### ### Supported Errors

| Error Condition            | Detection Point | Response            |
|----------------------------|-----------------|---------------------|
| Read request               | IDLE state      | `apb_rsp_error = 1` |
| Address out of range       | IDLE state      | `apb_rsp_error = 1` |
| (Future: channel disabled) | ROUTE state     | `apb_rsp_error = 1` |

### ### Error Flow

1. Software reads from BASE+0x00 (read not supported) OR Software writes to BASE+0x100 (out of range)
2. apbtodescr IDLE state:
  - Latches r\_error = 1
  - FSM: IDLE → ROUTE (briefly) → RESPOND
3. ROUTE state:
  - Skips descriptor engine routing (r\_error asserted)
  - Immediately transitions to RESPOND
4. RESPOND state:
  - apb\_rsp\_valid = 1
  - apb\_rsp\_error = 1 ← Error flagged
5. APB slave returns error to software

Duration: 2-3 cycles (faster than normal write)

## ## Parameters

| Parameter    | Type        | Default     | Description                          |
|--------------|-------------|-------------|--------------------------------------|
| ADDR_WIDTH   | int         | 32          | APB address bus width                |
| DATA_WIDTH   | int         | 32          | APB data bus width                   |
| NUM_CHANNELS | int         | 8           | Number of descriptor engine channels |
| BASE_ADDR    | logic[31:0] | 0x0000_0000 | Base address for channel registers   |

## ## Port List

### ### Clock and Reset

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

### ### APB CMD Interface (Slave Side)

\*\*From PeakRDL APB Slave:\*\*

| Signal          | Direction | Width      | Description                          |
|-----------------|-----------|------------|--------------------------------------|
| `apb_cmd_valid` | input     | 1          | Command valid from APB slave         |
| `apb_cmd_ready` | output    | 1          | Ready to accept command (IDLE state) |
| `apb_cmd_addr`  | input     | ADDR_WIDTH | Target address                       |
| `apb_cmd_wdata` | input     | DATA_WIDTH | Write data (descriptor address)      |
| `apb_cmd_write` | input     | 1          | 1=write, 0=read                      |

### ### APB RSP Interface (Slave Side)

\*\*To PeakRDL APB Slave:\*\*

| Signal          | Direction | Width      | Description                        |
|-----------------|-----------|------------|------------------------------------|
| `apb_rsp_valid` | output    | 1          | Response valid to APB slave        |
| `apb_rsp_ready` | input     | 1          | APB slave ready to accept response |
| `apb_rsp_rdata` | output    | DATA_WIDTH | Read data (always 0 - writes       |

```

only) |
| `apb_rsp_error` | output | 1 | Transaction error flag |

Descriptor Engine APB Interface (Master Side)

To descriptor_engine[0..7] APB ports:

Signal	Direction	Width	Description
`desc_apb_valid`	output	NUM_CHANNELS	Per-channel valid (one-hot)
`desc_apb_ready`	input	NUM_CHANNELS	Per-channel ready from desc engine
`desc_apb_addr`	output	NUM_CHANNELS × 64	Descriptor address (zero-extended from 32-bit)

```

**Notes:**

- Only one `desc\_apb\_valid[ch]` asserted at a time (one-hot)
- All channels receive same address data (only selected channel's valid asserted)
- 32-bit APB write data zero-extended to 64-bit descriptor address
- Upper 32 bits always 0 (assumes descriptors in lower 4GB address space)

### ### Integration Control Signal

**For parent module response muxing:**

| Signal                       | Direction | Width | Description                              |
|------------------------------|-----------|-------|------------------------------------------|
| `apb_descriptor_kickoff_hit` | output    | 1     | This block handling kick-off transaction |

**Behavior:**

- Asserted when: `(state == ROUTE || state == RESPOND) && !r\_error`
- Indicates apbtodescr is actively handling a valid kick-off write
- Parent module should route `apb\_rsp\_\*` from this block when asserted
- Deasserted for error cases (read requests, out-of-range addresses)

---

### ## Interface Details

#### ### APB CMD/RSP Interface (Slave Side)

**From PeakRDL APB Slave:**

| Signal        | Width      | Direction | Description                          |
|---------------|------------|-----------|--------------------------------------|
| apb_cmd_valid | 1          | Input     | Command valid from APB slave         |
| apb_cmd_ready | 1          | Output    | Ready to accept command (IDLE state) |
| apb_cmd_addr  | ADDR_WIDTH | Input     | Target address                       |
| apb_cmd_wdata | DATA_WIDTH | Input     | Write data (descriptor address)      |
| apb_cmd_write | 1          | Input     | 1=write, 0=read                      |

| Signal        | Width      | Direction | Description                        |
|---------------|------------|-----------|------------------------------------|
| apb_rsp_valid | 1          | Output    | Response valid to APB slave        |
| apb_rsp_ready | 1          | Input     | APB slave ready to accept response |
| apb_rsp_rdata | DATA_WIDTH | Output    | Read data (always 0 - writes only) |
| apb_rsp_error | 1          | Output    | Transaction error flag             |

### ### Descriptor Engine APB Interface (Master Side)

\*\*To descriptor\_engine[0..7] APB ports:\*\*

| Signal         | Width                    | Direction | Description                        |
|----------------|--------------------------|-----------|------------------------------------|
| desc_apb_valid | [NUM_CHANNELS-1:0]       | Output    | Per-channel valid (one-hot)        |
| desc_apb_ready | [NUM_CHANNELS-1:0]       | Input     | Per-channel ready from desc engine |
| desc_apb_addr  | [NUM_CHANNELS-1:0][63:0] | Output    | Descriptor address (zero-extended) |

\*\*Notes:\*\*

- Only one `desc\_apb\_valid[ch]` asserted at a time (one-hot)
- All channels receive same address data (only selected channel's valid asserted)
- 32-bit APB write data zero-extended to 64-bit descriptor address
- Upper 32 bits always 0 (assumes descriptors in lower 4GB address)

space)

### ### Integration Control Signal

\*\*For parent module response muxing:\*\*

| Signal                     | Width | Direction | Description                              |
|----------------------------|-------|-----------|------------------------------------------|
| apb_descriptor_kickoff_hit | 1     | Output    | This block handling kick-off transaction |

\*\*Behavior:\*\*

- Asserted when: `(state == ROUTE || state == RESPOND) && !r\_error`
- Indicates apbtodescr is actively handling a valid kick-off write
- Parent module should route `apb\_rsp\_\*` from this block when asserted
- Deasserted for error cases (read requests, out-of-range addresses)

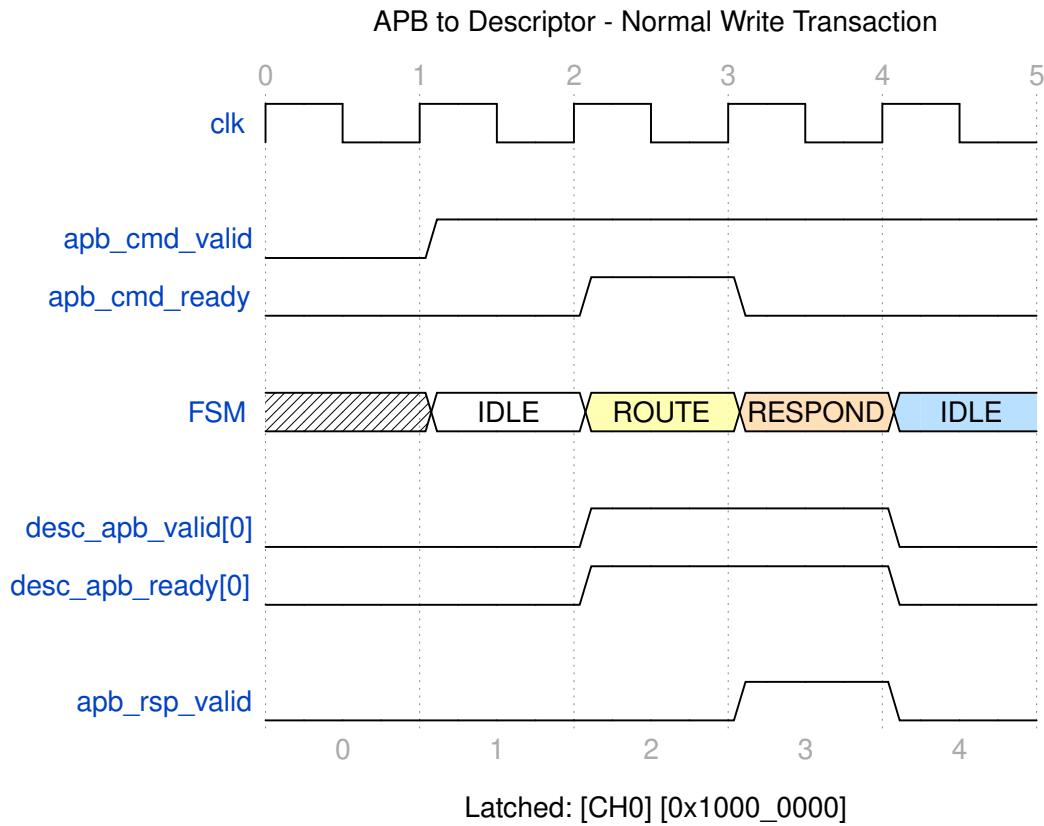
\*\*Usage:\*\*

```
```systemverilog
// Parent module muxing APB responses
assign apb_rsp_valid = apb_descriptor_kickoff_hit ?
                        apbtodescr_rsp_valid :
                        register_file_rsp_valid;

assign apb_rsp_error = apb_descriptor_kickoff_hit ?
                        apbtodescr_rsp_error :
                        register_file_rsp_error;
```

Timing Diagrams

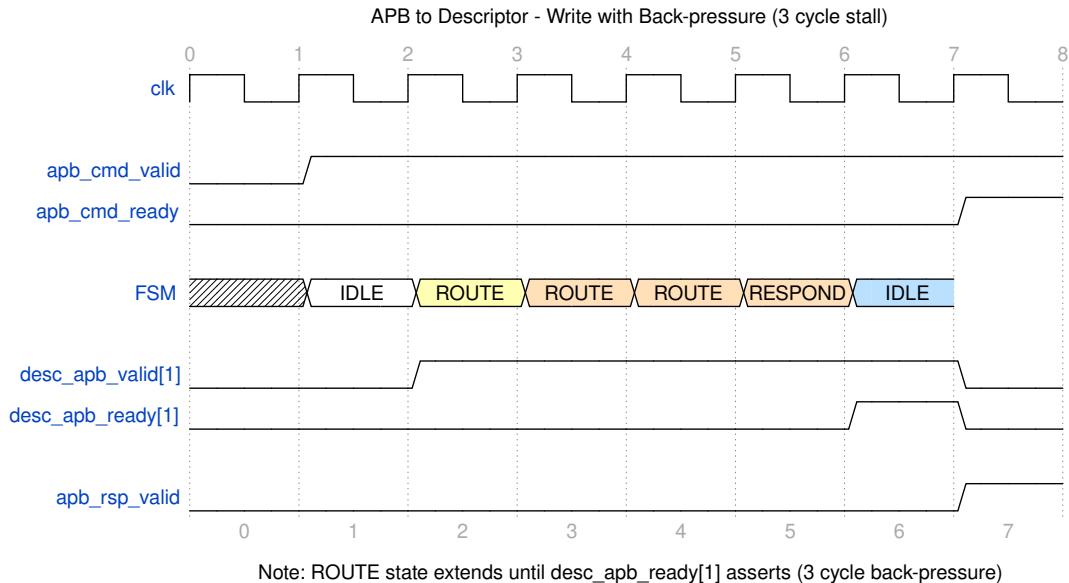
Normal Write Transaction



APB to Descriptor Normal Write

Source: [apbtodescr_normal_write.json](#)

Write with Back-pressure (3 cycle stall)



APB to Descriptor Write with Back-pressure

Source: [apbtodescr_backpressure_write.json](#)

Design Rationale

Why No Internal Registers?

Advantages: 1. **Zero latency** - Descriptor address passes directly through (no buffering delay) 2. **Simplicity** - FSM only, no FIFO management 3. **Area efficient** - Minimal logic (address decode + 3-state FSM) 4. **Natural back-pressure** - APB transaction blocks if descriptor engine busy

Trade-offs: - APB master (software) must wait if descriptor engine skid buffer full
- Acceptable: Descriptor fetch is infrequent (only at transfer start)

Why Zero-extend to 64-bit?

Rationale: - Descriptor addresses are 64-bit in STREAM architecture - APB data width is 32-bit (standard config bus width) - Assume descriptors reside in lower 4GB physical address space - Upper 32 bits hardcoded to 0 (physical address constraint)

Future Enhancement: - Add second register for upper 32 bits if full 64-bit addressing needed - Example: CH0_CTRL_LO (0x00), CH0_CTRL_HI (0x04), CH1_CTRL_LO (0x08), etc.

Why FSM vs Combinational?

FSM Benefits: 1. **Explicit state visibility** - Clear transaction phases for debug 2. **Clean handshaking** - Well-defined valid/ready protocol timing 3. **Error handling** - Centralized error response generation 4. **Timing closure** - Registered paths avoid long combinational chains

Integration Example

Typical System Integration

```
// PeakRDL APB slave (register interface)
apb_slave #(
    .BASE_ADDR(32'h4000_0000)
) u_apb_slave (
    .s_apb_*(...),           // System APB bus
    .reg_cmd_valid(apb_cmd_valid),
    .reg_cmd_ready(apb_cmd_ready),
    .reg_cmd_addr(apb_cmd_addr),
    .reg_cmd_wdata(apb_cmd_wdata),
    .reg_cmd_write(apb_cmd_write),

    .reg_rsp_valid(apb_rsp_valid_muxed), // After mux
    .reg_rsp_ready(apb_rsp_ready),
    .reg_rsp_rdata(apb_rsp_rdata_muxed), // After mux
    .reg_rsp_error(apb_rsp_error_muxed) // After mux
);

// APB-to-Descriptor router (handles 0x00-0x1C)
apbtodescr #(
    .NUM_CHANNELS(8),
    .BASE_ADDR(32'h4000_0000) // Match slave base
) u_apbtodescr (
    .clk(clk),
    .rst_n(rst_n),

    // APB slave interface
    .apb_cmd_valid(apb_cmd_valid),
    .apb_cmd_ready(apb_cmd_ready),
    .apb_cmd_addr(apb_cmd_addr),
    .apb_cmd_wdata(apb_cmd_wdata),
    .apb_cmd_write(apb_cmd_write),

    .apb_rsp_valid(apbtodescr_rsp_valid),
    .apb_rsp_ready(apb_rsp_ready),
    .apb_rsp_rdata(apbtodescr_rsp_rdata),
    .apb_rsp_error(apbtodescr_rsp_error),
```

```

// Descriptor engine connections
.desc_apb_valid(desc_apb_valid[7:0]),
.desc_apb_ready(desc_apb_ready[7:0]),
.desc_apb_addr(desc_apb_addr[7:0]),

// Integration control
.apb_descriptor_kickoff_hit(kickoff_hit)
);

// Register file (handles 0x100+)
stream_reg_file #(...) u_reg_file (
    .clk(clk),
    .rst_n(rst_n),

    .apb_cmd_valid(apb_cmd_valid),
    .apb_cmd_ready(), // Not used - always ready
    .apb_cmd_addr(apb_cmd_addr),
    .apb_cmd_wdata(apb_cmd_wdata),
    .apb_cmd_write(apb_cmd_write),

    .apb_rsp_valid(regfile_rsp_valid),
    .apb_rsp_ready(apb_rsp_ready),
    .apb_rsp_rdata(regfile_rsp_rdata),
    .apb_rsp_error(regfile_rsp_error)
);

// Response muxing: Select between kick-off and register responses
assign apb_rsp_valid_muxed = kickoff_hit ? apbtodescr_rsp_valid : regfile_rsp_valid;
assign apb_rsp_rdata_muxed = kickoff_hit ? apbtodescr_rsp_rdata : regfile_rsp_rdata;
assign apb_rsp_error_muxed = kickoff_hit ? apbtodescr_rsp_error : regfile_rsp_error;

// Scheduler group array (contains 8 descriptor engines)
scheduler_group_array #(...) u_scheduler_groups (
    .apb_valid(desc_apb_valid),
    .apb_ready(desc_apb_ready),
    .apb_addr(desc_apb_addr),
    // ...
);

```

Key Integration Points:

1. **Command Distribution:** Both apbtodescr and register file receive same CMD interface
 - apbtodescr handles 0x00-0x1C (kick-off registers)

- Register file handles 0x100+ (config/status registers)
- 2. **Response Muxing:** apb_descriptor_kickoff_hit selects response source
 - When HIGH: Route apbtodescr responses (kick-off transaction active)
 - When LOW: Route register file responses (config access)
- 3. **Address Decode:** apbtodescr internally decodes its range, asserts hit signal
 - Parent module doesn't need address decode logic
 - Clean separation of concerns

Verification Considerations

Testbench Focus Areas

1. **Address Decode**
 - All 8 channels addressable
 - Out-of-range addresses flagged as error
 - Channel ID extraction correct
2. **Back-pressure Handling**
 - Transaction blocks when desc_apb_ready = 0
 - Multiple cycle stalls handled correctly
 - No protocol violations during back-pressure
3. **Error Cases**
 - Read requests return error
 - Out-of-range writes return error
 - Error response faster than normal write
4. **Concurrent Operations**
 - Sequential writes to different channels
 - Verify no crosstalk between channels

Key Assertions

```

// Only one channel valid at a time
assert property (@(posedge clk) $onehot0(desc_apb_valid));

// Valid only in ROUTE state
assert property (@(posedge clk) |desc_apb_valid| -> (state == ROUTE));

// No simultaneous cmd_ready and rsp_valid
assert property (@(posedge clk) !(apb_cmd_ready && apb_rsp_valid));

```

Performance Characteristics

Metric	Value	Notes
Latency (no back-pressure)	3 cycles	IDLE → ROUTE → RESPOND
Latency (with back-pressure)	3 + N cycles	N = descriptor engine ready delay
Throughput	1 write/3 cyc	Sequential writes to same channel
Area	~500 gates	Estimate: FSM + decode logic
Max Frequency	System clock	Single-cycle paths, no timing issues

Related Modules

- **descriptor_engine** - Receives kick-off via desc_apb_* interface
- **scheduler_group_array** - Contains 8 descriptor engines
- **apb_slave** (PeakRDL) - Upstream APB register interface

Revision History

Version	Date	Author	Description
1.0	2025-10-20	sean galloway	Initial creation

File: projects/components/stream/rtl/stream_macro/apbtodescr.sv

Documentation:

projects/components/stream/docs/stream_spec/ch02_blocks/02_apbtodescr.md

APB Config Specification

Module: stream_regs.rdl (PeakRDL definition) + wrapper **Location:** regs/(definition), rtl/stream_macro/ (wrapper) **Status:** Future - PeakRDL generation planned (following HPET pattern)

Overview

The APB Config provides the APB slave interface for STREAM configuration and control. Following the same pattern as apb_hpet, this will be implemented using:
1. PeakRDL register definition (stream_regs.rdl)
2. Auto-generated register RTL (peakrdl_generate.py)
3. Wrapper module with optional CDC (apb_slave_cdc)

Implementation Status

Current: Not yet implemented - deferred until after Phase 2 RTL completion
Future: Will follow HPET pattern with PeakRDL-generated registers

Reference: See projects/components/apb_hpet/ for proven implementation pattern

Key Features

- APB slave interface (APB4 protocol)
 - 8 channel register sets (16 bytes per channel)
 - Global control and status registers
 - Kick-off via single APB write
 - Optional CDC wrapper (like HPET apb_slave_cdc)
-

Register Map

Global Registers

Offset	Name	Access	Width	Description
0x00	GLOBAL_C_TRL	RW	32	Global enable, channel resets
0x04	GLOBAL_STATUS	RO	32	Channel idle/error status
0x08	GLOBAL_CONFIG	RW	32	Global configuration
0x0C	(Reserved)	-	-	-

GLOBAL_CTRL (0x00):

Bits [31:24] - Reserved
Bits [23:16] - Channel reset (one-hot, auto-clear)
Bits [15:8] - Reserved
Bit [0] - Global enable

GLOBAL_STATUS (0x04):

Bits [31:24] - Reserved
 Bits [23:16] - Channel error flags
 Bits [15:8] - Reserved
 Bits [7:0] - Channel idle flags

Channel Registers (8 channels 0x10 bytes)

Base addresses: - CH0: 0x10 - 0x1F - CH1: 0x20 - 0x2F - CH2: 0x30 - 0x3F - CH3: 0x40 - 0x4F - CH4: 0x50 - 0x5F - CH5: 0x60 - 0x6F - CH6: 0x70 - 0x7F - CH7: 0x80 - 0x8F

Per-Channel Registers:

Offset	Name	Access	Width	Description
+0x00	CHx_CTRL	WO	32	Descriptor address (write to kick off)
+0x04	CHx_STATUS	RO	32	Channel status
+0x08	CHx_RD_BU	RW RST	32	Read burst length config
+0x0C	CHx_WR_B	RW URST	32	Write burst length config

CHx_CTRL (+0x00):

Bits [31:0] - Descriptor address (word-aligned)

Action: Write to this register kicks off descriptor chain fetch

CHx_STATUS (+0x04):

Bits [31:3] - Reserved
 Bit [2] - Error flag
 Bit [1] - Idle flag
 Bit [0] - Enable flag

CHx_RD_BURST (+0x08):

Bits [31:8] - Reserved
 Bits [7:0] - Read burst length (beats)

Used by AXI Read Engine for cfg_burst_len

CHx_WR_BURST (+0x0C):

Bits [31:8] - Reserved
Bits [7:0] - Write burst length (beats)

Used by AXI Write Engine for cfg_burst_len

Interface

Parameters

```
parameter int NUM_CHANNELS = 8;  
parameter int ADDR_WIDTH = 32;  
parameter int DATA_WIDTH = 32;
```

Port List

Clock and Reset

Signal	Direction	Width	Description
pclk	input	1	APB clock
presetn	input	1	APB active-low reset

APB Slave Interface

Signal	Direction	Width	Description
paddr	input	ADDR_WIDTH	APB address
psel	input	1	APB select
penable	input	1	APB enable
pwrite	input	1	APB write (1=write, 0=read)
pwdata	input	DATA_WIDTH	APB write data
pstrb	input	4	APB byte strobe
pready	output	1	APB transfer

Signal	Direction	Width	Description
			ready
prdata	output	DATA_WIDTH	APB read data
pslverr	output	1	APB slave error

Configuration Outputs (Per-Channel)

Signal	Direction	Width	Description
ch_enable	output	NUM_CHAN NELS	Per-channel enable flags
ch_reset	output	NUM_CHAN NELS	Per-channel reset flags
ch_desc_addr	output	NUM_CHAN NELS × 64	Per-channel descriptor addresses
ch_read_burst_len	output	NUM_CHAN NELS × 8	Per-channel read burst lengths
ch_write_burst_st_len	output	NUM_CHAN NELS × 8	Per-channel write burst lengths

Status Inputs (Per-Channel)

Signal	Direction	Width	Description
ch_idle	input	NUM_CHAN NELS	Per-channel idle status
ch_error	input	NUM_CHAN NELS	Per-channel error flags
ch_bytes_xferred	input	NUM_CHAN NELS × 32	Per-channel bytes transferred

Ports

APB Slave Interface:

```



```

```

input logic pwrite;
input logic [DATA_WIDTH-1:0] pwdata;
input logic [3:0] pstrb;
output logic pready;
output logic [DATA_WIDTH-1:0] prdata;
output logic pslverr;

```

Configuration Outputs (per channel):

```

output logic [NUM_CHANNELS-1:0] ch_enable;
output logic [NUM_CHANNELS-1:0] ch_reset;
output logic [NUM_CHANNELS-1:0][63:0] ch_desc_addr;
output logic [NUM_CHANNELS-1:0][7:0] ch_read_burst_len;
output logic [NUM_CHANNELS-1:0][7:0] ch_write_burst_len;

```

Status Inputs (per channel):

```

input logic [NUM_CHANNELS-1:0] ch_idle;
input logic [NUM_CHANNELS-1:0] ch_error;
input logic [NUM_CHANNELS-1:0][31:0] ch_bytes_xfered;

```

Operation

Kick-Off Sequence

Software writes descriptor address to CHx_CTRL:

```

// Software: Kick off channel 0 transfer
write_apb(ADDR_CH0_CTRL, 0x1000_0000);

// Hardware response:
// 1. Register captures descriptor address
// 2. ch_desc_addr[0] <= 0x1000_0000
// 3. ch_enable[0] auto-asserts
// 4. Scheduler begins descriptor fetch

```

Auto-Enable Behavior

```

// On CHx_CTRL write
if (pwrite && paddr == CHx_CTRL_ADDR) begin
    r_ch_desc_addr[channel_id] <= pwdata;
    r_ch_enable[channel_id] <= 1'b1; // Auto-enable on kick-off
end

// Auto-clear when transfer completes
if (ch_idle[channel_id]) begin
    r_ch_enable[channel_id] <= 1'b0;
end

```

PeakRDL Generation Workflow

Step-by-Step Process

See: `regs/README.md` for complete workflow details

1. **Create:** `regs/stream_regs.rdl` (register definition)

2. **Generate:**

```
cd projects/components/stream/regs
../../bin/peakrdl_generate.py stream_regs.rdl --copy-rtl
./rtl/stream_macro
```

3. **Create Wrapper:** `rtl/stream_macro/app_config.sv` to instantiate generated registers

Wrapper Pattern (Future Implementation)

```
// apb_config.sv wrapper (to be created)
module apb_config (
    // APB interface
    input logic      pclk,
    // ...

    // Configuration outputs
    output logic [7:0] ch_enable,
    // ...
);

// Instantiate PeakRDL-generated registers
stream_regs u_regs (
    .pclk      (pclk),
    .presetn   (presetn),
    .paddr     (paddr),
    // ... APB signals

    // Generated field outputs
    .global_ctrl_enable  (global_enable),
    .ch0_ctrl_desc_addr (ch_desc_addr[0]),
    .ch0_rd_burst       (ch_read_burst_len[0]),
    // ...
);

// Optional CDC wrapper (if crossing clock domains)
// Like HPET: apb_hpet.sv wraps apb_slave_cdc
```

```
endmodule
```

CDC Considerations

If APB clock != STREAM aclk:

Use CDC wrapper pattern from HPET:

```
// APB domain (pclk)
apb_slave_cdc #(
    .ADDR_WIDTH(32),
    .DATA_WIDTH(32)
) u_cdc (
    // APB side (pclk domain)
    .s_pclk      (pclk),
    .s_presetn   (presetn),
    .s_paddr     (paddr),
    // ...
    // Core side (aclk domain)
    .m_pclk      (aclk),
    .m_presetn   (aresetn),
    .m_paddr     (paddr_sync),
    // ...
);
// STREAM registers in aclk domain
stream_regs u_regs (
    .pclk      (aclk), // Note: aclk, not pclk
    .paddr     (paddr_sync),
    // ...
);
```

Default Values

On reset (presetn = 0):

```
// Global
global_enable <= 1'b0;

// Per-channel
for (int i = 0; i < NUM_CHANNELS; i++) begin
    ch_enable[i] <= 1'b0;
    ch_reset[i] <= 1'b0;
```

```
ch_desc_addr[i] <= 64'h0;
ch_read_burst_len[i] <= 8'd8; // Default: 8-beat read bursts
ch_write_burst_len[i] <= 8'd16; // Default: 16-beat write bursts
end
```

Testing

Test Location: projects/components/stream/dv/tests/integration_tests/

Test Scenarios: 1. Register read/write (all registers) 2. Kick-off via CHx_CTRL write 3. Auto-enable behavior 4. Status register reads 5. Multi-channel configuration 6. Reset behavior

Related Documentation

- **Register Generation:** regs/README.md - PeakRDL workflow
 - **HPET Example:** projects/components/apb_hpét/ - Reference implementation
 - **Scheduler:** 02_scheduler.md - Consumer of configuration
-

Last Updated: 2025-10-17

Performance Profiler Specification

Module: perf_profiler.sv **Location:**

projects/components/stream/rtl/stream_fub/ **Source:** New for STREAM

Overview

The Performance Profiler captures timing information for channel activity to enable performance analysis and bottleneck identification. It monitors channel idle signals and records timestamps or elapsed times into a FIFO for later retrieval via APB registers.

Key Features

- Dual-mode profiling: timestamp mode or elapsed time mode
- 32-bit free-running counter for timing reference

- 256-entry FIFO for buffering performance events
 - Per-channel tracking (up to 8 channels)
 - Channel ID tagging for multi-channel identification
 - Priority encoder for simultaneous events
 - Two-register APB read interface (36-bit data over 32-bit bus)
-

Interface

Parameters

```
parameter int NUM_CHANNELS = 8;           // Number of channels to monitor
parameter int CHANNEL_WIDTH = $clog2(NUM_CHANNELS);
parameter int TIMESTAMP_WIDTH = 32;        // Timestamp counter width
parameter int FIFO_DEPTH = 256;            // Performance event FIFO depth
parameter int FIFO_ADDR_WIDTH = $clog2(FIFO_DEPTH);
```

Port List

Clock and Reset

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

Channel Monitoring

Signal	Direction	Width	Description
channel_idle	input	NUM_CHAN_NELS	Idle status per channel (1=idle, 0=active)

Configuration Interface

Signal	Direction	Width	Description
cfg_enable	input	1	Enable profiling
cfg_mode	input	1	Profiling mode (0=timestamp,

Signal	Direction	Width	Description
cfg_clear	input	1	1=elapsed) Clear FIFO and counters

FIFO Read Interface

To APB Config Space:

Signal	Direction	Width	Description
perf_fifo_rd	input	1	Read strobe (pops FIFO)
perf_fifo_da_ta_low	output	32	Timestamp or elapsed time [31:0]
perf_fifo_da_ta_high	output	32	Metadata: {28'b0, event_type, channel_id[2:0]}
perf_fifo_empty	output	1	FIFO empty flag
perf_fifo_full	output	1	FIFO full flag
perf_fifo_count	output	16	Number of entries in FIFO

FIFO Entry Format: - perf_fifo_data_low[31:0]: Timestamp (mode 0) or Elapsed time (mode 1) - perf_fifo_data_high[3]: Event type (0=start, 1=end) - perf_fifo_data_high[2:0]: Channel ID (0-7)

Ports

Clock and Reset:

```
input logic clk;
input logic rst_n;
```

Channel Idle Signals (from Schedulers):

```
input logic [NUM_CHANNELS-1:0] channel_idle; // Idle status per channel
                                                // 1 = idle, 0 = active
```

Configuration Interface:

```



```

FIFO Read Interface (to APB Config Space):

```




---



```

Profiling Modes

Mode 0: TIMESTAMP_MODE (More Flexible)

Operation: - Records timestamp when channel transitions idle → active (start event) - Records timestamp when channel transitions active → idle (end event) - Software calculates elapsed time from timestamp pairs - Simpler hardware, more flexible post-processing

FIFO Entry Format: - perf_fifo_data_low[31:0]: Timestamp (cycles) - perf_fifo_data_high[3]: Event type (0=start, 1=end) - perf_fifo_data_high[2:0]: Channel ID (0-7)

Software Processing:

```

// Read two FIFO entries for start/end pair
uint32_t timestamp_start = read_perf_fifo(); // event_type=0
uint32_t timestamp_end   = read_perf_fifo(); // event_type=1
uint32_t elapsed = timestamp_end - timestamp_start;

```

Mode 1: ELAPSED_MODE (Simpler Software)

Operation: - Captures timestamp when channel becomes active (idle falls) - Computes elapsed time when channel returns to idle (idle rises) - Records elapsed time directly in FIFO - Hardware calculates duration, simpler software

FIFO Entry Format: - perf_fifo_data_low[31:0]: Elapsed time (cycles) - perf_fifo_data_high[3]: Always 1 (end event) - perf_fifo_data_high[2:0]: Channel ID (0-7)

Software Processing:

```
// Read single FIFO entry for complete measurement
uint32_t elapsed = read_perf_fifo(); // Direct elapsed time
```

Two-Register Read Interface

The profiler stores 36-bit entries (32-bit timestamp/elapsed + 4-bit metadata) in the FIFO. Since APB registers are 32 bits wide, the data is split across two registers with atomic read semantics.

Read Sequence

Hardware Behavior: 1. **Read PERF_FIFO_DATA_LOW (address 0xXXX):** - APB slave asserts perf_fifo_rd for one cycle - FIFO pops and provides 36-bit data - Data is latched into internal register - APB returns [31:0] (timestamp or elapsed time)

2. **Read PERF_FIFO_DATA_HIGH (address 0xXXX+4):**

- APB returns {28'b0, [35:32]} from latched data
- No FIFO pop occurs (reads previously latched data)

Software Example:

```
// APB register addresses (example)
#define PERF_FIFO_DATA_LOW 0x200
#define PERF_FIFO_DATA_HIGH 0x204
#define PERF_FIFO_STATUS 0x208

// Check if data available
if (!(read_apb(PERF_FIFO_STATUS) & FIFO_EMPTY)) {
    // Read 36-bit entry atomically
    uint32_t timestamp = read_apb(PERF_FIFO_DATA_LOW); // Pops FIFO,
    latches data
    uint32_t metadata = read_apb(PERF_FIFO_DATA_HIGH); // Reads
    latched data

    // Parse metadata
    uint8_t channel_id = metadata & 0x7;           // Bits [2:0]
    uint8_t event_type = (metadata >> 3) & 1; // Bit [3]
```

```

// Process based on mode
if (cfg_mode == TIMESTAMP_MODE) {
    if (event_type == 0) {
        // Start event
        start_times[channel_id] = timestamp;
    } else {
        // End event - calculate elapsed
        uint32_t elapsed = timestamp - start_times[channel_id];
        printf("Channel %d: %u cycles\n", channel_id, elapsed);
    }
} else {
    // ELAPSED_MODE - timestamp is already elapsed time
    printf("Channel %d: %u cycles\n", channel_id, timestamp);
}

```

Operation Details

Free-Running Timestamp Counter

- 32-bit counter increments every clock cycle when profiling enabled
- Provides global timing reference for all measurements
- Wraps around at $2^{32} - 1$ (software must handle rollover)
- Can be cleared via cfg_clear

Example: - Clock frequency: 100 MHz - Counter range: 0 to 4,294,967,295 - Maximum time: ~42.9 seconds before rollover - Software should handle wrap-around in timestamp calculations

Edge Detection

Falling Edge (Idle → Active): - Channel starts operation - In TIMESTAMP_MODE: Records timestamp with event_type=0 (start) - In ELAPSED_MODE: Records start time internally, no FIFO write

Rising Edge (Active → Idle): - Channel completes operation - In TIMESTAMP_MODE: Records timestamp with event_type=1 (end) - In ELAPSED_MODE: Computes elapsed time, writes to FIFO

Multi-Channel Priority Encoder

When multiple channels have events simultaneously (rare but possible): - Priority encoder selects lowest channel number first - One event processed per cycle - Remaining events captured in subsequent cycles

Example:

```
Cycle N:    Channels 0, 3, 5 all transition  
            → Process Channel 0 event, push to FIFO  
Cycle N+1:  Channels 3, 5 still pending  
            → Process Channel 3 event, push to FIFO  
Cycle N+2:  Channel 5 still pending  
            → Process Channel 5 event, push to FIFO
```

Configuration

Enable Profiling

```
// Enable profiler in timestamp mode  
write_apb(PERF_CTRL, (1 << ENABLE_BIT) | (0 << MODE_BIT));  
  
// Enable profiler in elapsed mode  
write_apb(PERF_CTRL, (1 << ENABLE_BIT) | (1 << MODE_BIT));
```

Clear FIFO and Counters

```
// Clear all profiler state  
write_apb(PERF_CTRL, (1 << CLEAR_BIT));  
// Wait one cycle, then clear bit  
write_apb(PERF_CTRL, 0);
```

Read FIFO Entries

```
void read_all_perf_events(void) {  
    uint32_t status;  
  
    // Poll until FIFO empty  
    while ((status = read_apb(PERF_FIFO_STATUS)) & FIFO_EMPTY_BIT) {  
        // Read 36-bit entry (two registers)  
        uint32_t data_low = read_apb(PERF_FIFO_DATA_LOW);  
        uint32_t data_high = read_apb(PERF_FIFO_DATA_HIGH);  
  
        // Parse and process  
        uint8_t channel_id = data_high & 0x7;  
        uint8_t event_type = (data_high >> 3) & 1;  
  
        printf("Channel %d: %s at %u\n",  
               channel_id,  
               event_type ? "END" : "START",  
               data_low);  
    }  
}
```

FIFO Behavior

Buffering

- 256-entry FIFO (configurable via FIFO_DEPTH parameter)
- Each entry: 36 bits (32-bit timestamp/elapsed + 4-bit metadata)
- Status outputs: perf_fifo_empty, perf_fifo_full, perf_fifo_count

Full Handling

When FIFO full: - New events are **DROPPED** (no backpressure to schedulers) - Profiler continues tracking but cannot record new events - Software should poll frequently to prevent loss

Best Practices: 1. Monitor perf_fifo_count to detect near-full conditions 2. Enable interrupt on FIFO half-full threshold (future enhancement) 3. Poll FIFO regularly during active profiling 4. Increase FIFO_DEPTH parameter if loss occurs

Integration Example

```
perf_profiler #(
    .NUM_CHANNELS(8),
    .TIMESTAMP_WIDTH(32),
    .FIFO_DEPTH(256)
) u_perf_profiler (
    // Clock and Reset
    .clk                    (aclk),
    .rst_n                  (aresetn),

    // Channel idle signals from schedulers
    .channel_idle           ({sched7_idle, sched6_idle, ..., sched0_idle}),

    // Configuration from APB registers
    .cfg_enable              (perf_ctrl_reg[0]),
    .cfg_mode                (perf_ctrl_reg[1]),
    .cfg_clear               (perf_ctrl_reg[2]),

    // FIFO read interface (connect to APB slave)
    .perf_fifo_rd            (perf_fifo_rd_strobe), // Triggered by LOW
    register read
    .perf_fifo_data_low      (perf_fifo_data_low),   // Maps to APB
    register 0x200
    .perf_fifo_data_high     (perf_fifo_data_high),  // Maps to APB
    register 0x204
    .perf_fifo_empty          (perf_fifo_empty),     // Status register
```

```

bit .perf_fifo_full      (perf_fifo_full),      // Status register
bit .perf_fifo_count     (perf_fifo_count)      // Status register
[15:0]
);

```

APB Register Map Example

Suggested APB register mapping (to be integrated into `stream_regs.rdl`):

Address	Register	Access	Description
0x1F0	PERF_CTRL	RW	[2]=clear, [1]=mode, [0]=enable
0x1F4	PERF_STATUS	RO	[31:16]=count, [1]=full, [0]=empty
0x1F8	PERF_FIFO_DATA _LOW	RO	[31:0] timestamp/elapsed (pops FIFO)
0x1FC	PERF_FIFO_DATA _HIGH	RO	{28'b0, event_type, channel_id[2:0]}

Use Cases

1. Identify Busy Channels

Monitor which channels spend most time active (not idle):

```

// Enable elapsed mode profiling
enable_profiler(ELAPSED_MODE);

// Run workload for 1 second
sleep(1);

// Read and accumulate per-channel totals
uint32_t channel_busy_time[8] = {0};
while (!fifo_empty()) {
    uint32_t elapsed = read_perf_fifo_low();
    uint32_t meta    = read_perf_fifo_high();
    uint8_t ch = meta & 0x7;
    channel_busy_time[ch] += elapsed;
}

```

```

// Report busiest channels
for (int ch = 0; ch < 8; ch++) {
    printf("Channel %d: %.1f% busy\n",
        ch,
        100.0 * channel_busy_time[ch] / total_cycles);
}

```

2. Measure Transfer Latency

Measure time from descriptor kick-off to completion:

```

// Enable timestamp mode
enable_profiler(TIMESTAMP_MODE);

// Kick off descriptor
kick_channel(0);

// Wait for completion
while (!channel_done(0)) {
    // Poll FIFO for events
    if (!fifo_empty()) {
        uint32_t timestamp = read_perf_fifo_low();
        uint32_t meta      = read_perf_fifo_high();
        uint8_t ch = meta & 0x7;
        uint8_t type = (meta >> 3) & 1;

        if (ch == 0 && type == 0) {
            start_time = timestamp;
        } else if (ch == 0 && type == 1) {
            end_time = timestamp;
            printf("Transfer latency: %u cycles\n", end_time -
start_time);
        }
    }
}

```

3. Detect Performance Bottlenecks

Identify channels with unexpectedly long active periods:

```

#define EXPECTED_MAX_CYCLES 10000

while (!fifo_empty()) {
    uint32_t elapsed = read_perf_fifo_low();
    uint32_t meta   = read_perf_fifo_high();
    uint8_t ch = meta & 0x7;

    if (elapsed > EXPECTED_MAX_CYCLES) {

```

```
    printf("WARNING: Channel %d took %u cycles (expected < %u)\n",
           ch, elapsed, EXPECTED_MAX_CYCLES);
}
```

Testing

Test Location:

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

Test Scenarios: 1. Timestamp mode - single channel 2. Timestamp mode - multiple channels 3. Elapsed mode - single channel 4. Elapsed mode - multiple channels 5. FIFO full handling (event dropping) 6. Simultaneous channel transitions (priority encoder) 7. Counter rollover handling 8. Two-register read sequence (atomic access)

Formal Verification

The module includes formal assertions (enabled with `FORMAL` define):

timestamp_monotonic: - Timestamp counter never decreases (unless clear or rollover) - Ensures timing measurements are valid

elapsed_requires_active: - In elapsed mode, channel must be active before recording elapsed time - Prevents spurious elapsed time entries

fifo_write_conditions: - FIFO write only when enabled and event detected - Ensures all FIFO entries are valid events

Related Documentation

- **Source:** `projects/components/stream/rtl/stream_fub/perf_profiler.sv`
 - **Integration:** `projects/components/stream/rtl/stream_macro/stream_top.sv`
 - **Registers:** `projects/components/stream/regs/stream_regs.rdl`
(PeakRDL specification)
 - **FIFO:** `rtl/amba/gaxi/gaxi_fifo_sync.sv` (underlying FIFO implementation)
-

Last Updated: 2025-10-20

MonBus AXIL Group Specification

Module: monbus_axil_group.sv **Location:** rtl/stream_macro/

Overview

The MonBus AXIL Group provides monitoring and error reporting for STREAM. It receives monitor bus packets from STREAM channels and provides AXIL interfaces for error/interrupt handling and packet logging to memory.

Key Features

- **Single MonBus input:** Receives all monitor packets
 - **Error FIFO:** Buffers error packets for software polling
 - **AXIL slave:** Read error/interrupt packets
 - **AXIL master:** Write monitor packets to system memory
 - **Interrupt output:** Asserted when error FIFO not empty
 - **Protocol filtering:** Per-protocol packet type filtering (AXI, AXIS, CORE)
-

Interface

Parameters

```
parameter int FIFO_DEPTH_ERR      = 64;      // Error/interrupt FIFO depth
parameter int FIFO_DEPTH_WRITE   = 32;      // Master write FIFO depth
parameter int ADDR_WIDTH          = 32;      // AXI address width
parameter int DATA_WIDTH          = 32;      // AXI data width (32 or 64)
parameter int NUM_PROTOCOLS       = 3;        // Number of protocols (AXI,
AXIS, CORE)
```

Port List

Clock and Reset

Signal	Direction	Width	Description
axi_aclk	input	1	AXI clock
axi_aresetn	input	1	AXI active-low reset

Monitor Bus Input

Signal	Direction	Width	Description
monbus_valid	input	1	Monitor bus packet valid
monbus_ready	output	1	Monitor bus ready
monbus_packet	input	64	64-bit monitor bus packet

AXIL Slave Interface (Error/Interrupt FIFO Read)

AR Channel:

Signal	Direction	Width	Description
s_axil_araddr	input	ADDR_WIDTH	Read address
s_axil_arprot	input	3	Protection attributes
s_axil_arvali	input	1	Read address valid
s_axil_arready	output	1	Read address ready

R Channel:

Signal	Direction	Width	Description
s_axil_rdata	output	DATA_WIDTH	Read data
s_axil_rresp	output	2	Read response
s_axil_rvalid	output	1	Read data valid
s_axil_rready	input	1	Read data ready

AXIL Master Interface (Monitor Packet Writes to Memory)

AW Channel:

Signal	Direction	Width	Description
m_axil_awaddr	output	ADDR_WIDTH	Write address
m_axil_awprot	output	3	Protection

Signal	Direction	Width	Description
			attributes
m_axil_awvali_d	output	1	Write address valid
m_axil_awready_y	input	1	Write address ready

W Channel:

Signal	Direction	Width	Description
m_axil_wdata	output	DATA_WIDTH	Write data
m_axil_wstrb	output	DATA_WIDTH/8	Write strobe
m_axil_wvalid	output	1	Write data valid
m_axil_wready	input	1	Write data ready

B Channel:

Signal	Direction	Width	Description
m_axil_bresp	input	2	Write response
m_axil_bvalid	input	1	Write response valid
m_axil_bready	output	1	Write response ready

Interrupt Output

Signal	Direction	Width	Description
irq_out	output	1	Interrupt request (asserted when error FIFO not empty)

Configuration Inputs

Signal	Direction	Width	Description
cfg_base_addr	input	ADDR_WIDTH	Base address for master writes
cfg_limit_ad	input	ADDR_WIDTH	Limit address for master

Signal	Direction	Width	Description
dr		H	writes
cfg_<proto>_pkt_mask	input	16	Per-protocol packet drop mask
cfg_<proto>_err_select	input	16	Per-protocol error FIFO select
cfg_<proto>_<type>_mask	input	16	Per-protocol event-specific masks

Note: <proto> represents AXI, AXIS, or CORE protocols

Ports

Clock and Reset:

```
input logic axi_aclk;
input logic axi_aresetn;
```

Monitor Bus Input (single input):

```
input logic monbus_valid;
output logic monbus_ready;
input logic [63:0] monbus_packet;
```

AXIL Slave (Error/Interrupt FIFO Read):

```
// AR channel
input logic [ADDR_WIDTH-1:0] s_axil_araddr;
input logic [2:0] s_axil_arprot;
input logic s_axil_arvalid;
output logic s_axil_arready;

// R channel
output logic [DATA_WIDTH-1:0] s_axil_rdata;
output logic [1:0] s_axil_rresp;
output logic s_axil_rvalid;
input logic s_axil_rready;
```

AXIL Master (Monitor Packet Writes to Memory):

```
// AW channel
output logic [ADDR_WIDTH-1:0] m_axil_awaddr;
output logic [2:0] m_axil_awprot;
output logic m_axil_awvalid;
input logic m_axil_awready;
```

```

// W channel
output logic [DATA_WIDTH-1:0] m_axil_wdata;
output logic [DATA_WIDTH/8-1:0] m_axil_wstrb;
output logic                  m_axil_wvalid;
input  logic                  m_axil_wready;

// B channel
input  logic [1:0]           m_axil_bresp;
input  logic                  m_axil_bvalid;
output logic                 m_axil_bready;

Interrupt Output:

output logic                   irq_out; // Asserted when error FIFO
not empty

```

Configuration Inputs:

```

master writes
master writes

// Per-protocol configuration (AXI, AXIS, CORE)
FIFO select
specific masks

```

Operation

Monitor Packet Flow (Simplified - No Arbitration)

MonBus Input -> Packet Classifier -> [Error FIFO | Write FIFO]

AXIL Slave	AXIL Master
(CPU read)	(Memory write)

STREAM Advantage: Direct path from input to filter (no arbiter delay)

Packet Classification

Error Packets: - Packet type indicates error (descriptor error, AXI error, timeout, etc.) - Routed to error FIFO - Triggers interrupt if cfg_error_irq_enable asserted

Normal Packets: - Completion, status, performance packets - Routed to log FIFO (if cfg_log_enable asserted) - Written to memory via AXIL master

Error FIFO (AXIL Slave Interface)

Software reads error packets:

```
// Software: Poll error FIFO
uint32_t error_pkt_low, error_pkt_high;

// Read lower 32 bits
error_pkt_low = read_axil(ERROR_FIFO_ADDR_LOW);

// Read upper 32 bits
error_pkt_high = read_axil(ERROR_FIFO_ADDR_HIGH);

// Combine into 64-bit packet
uint64_t error_packet = ((uint64_t)error_pkt_high << 32) |
error_pkt_low;
```

AXIL slave registers:

Address	Name	Access	Description
0x00	ERROR_PKT_L OW	RO	Error packet [31:0], auto-pop on read
0x04	ERROR_PKT_HI GH	RO	Error packet [63:32]
0x08	ERROR_FIFO_S TATUS	RO	FIFO count, empty, full flags
0x0C	IRQ_STATUS	RW1C	Interrupt status (write 1 to clear)

Log FIFO (AXIL Master Interface)

Automatic packet logging:

```
// On normal monitor packet
if (monbus_valid && !is_error_packet) begin
    // Write to memory via AXIL master
    m_axil_awaddr <= cfg_log_base_addr + (log_wr_ptr << 3);
    m_axil_wdata <= monbus_packet[31:0]; // Lower word
    // ... followed by upper word write
```

```
    log_wr_ptr <= log_wr_ptr + 1;  
end
```

Memory layout:

```
cfg_log_base_addr + 0x00: Packet 0 [31:0]  
cfg_log_base_addr + 0x04: Packet 0 [63:32]  
cfg_log_base_addr + 0x08: Packet 1 [31:0]  
cfg_log_base_addr + 0x0C: Packet 1 [63:32]  
...
```

Interrupt Assertion

```
// IRQ asserted when error FIFO not empty (if enabled)  
assign irq_out = cfg_error_irq_enable && !error_fifo_empty;  
  
// Software clears by reading error packets (drains FIFO)  
// Or by writing to IRQ_STATUS register (W1C)
```

MonBus Packet Format

64-bit packet structure:

```
[63:56] - Packet type (error, completion, status, etc.)  
[55:48] - Channel ID  
[47:40] - Reserved / packet-specific  
[39:0] - Packet-specific data
```

Error packet types: - 0xE0: Descriptor error - 0xE1: AXI read error - 0xE2: AXI write error - 0xE3: Timeout error

Normal packet types: - 0x10: Descriptor fetched - 0x20: Transfer complete - 0x30: Performance metrics

Usage in STREAM

Integration Pattern

```
monbus_axil_group #(  
    .FIFO_DEPTH_ERR(64),  
    .FIFO_DEPTH_WRITE(32),  
    .ADDR_WIDTH(32),  
    .DATA_WIDTH(32),  
    .NUM_PROTOCOLS(3)  
) u_monbus (  
    .axi_aclk(aclk),  
    .axi_aresetn(aresetn),
```

```

// Single monitor bus input (from STREAM channel arbiter)
.monbus_valid(stream_mon_valid),
.monbus_ready(stream_mon_ready),
.monbus_packet(stream_mon_packet),

// AXIL slave (CPU access to error FIFO)
.s_axil_araddr(cpu_araddr),
.s_axil_arprot(3'b000),
.s_axil_arvalid(cpu_arvalid),
.s_axil_arready(cpu_arready),
.s_axil_rdata(cpu_rdata),
.s_axil_rresp(cpu_rresp),
.s_axil_rvalid(cpu_rvalid),
.s_axil_rready(cpu_rready),

// AXIL master (log to memory)
.m_axil_awaddr(log_awaddr),
.m_axil_awprot(log_awprot),
.m_axil_awvalid(log_awvalid),
.m_axil_awready(log_awready),
.m_axil_wdata(log_wdata),
.m_axil_wstrb(log_wstrb),
.m_axil_wvalid(log_wvalid),
.m_axil_wready(log_wready),
.m_axil_bresp(log_bresp),
.m_axil_bvalid(log_bvalid),
.m_axil_bready(log_bready),

// Interrupt
.irq_out(stream_irq),

// Configuration
.cfg_base_addr(32'h8000_0000),
.cfg_limit_addr(32'h8FFF_FFFF),
.cfg_axi_pkt_mask(16'h0000),
.cfg_axi_err_select(16'h0001), // Route errors to error FIFO
// ... other protocol configurations
);

```

Error Handling Flow

Example: AXI Read Error

1. AXI Read Engine detects RRESP != OKAY
2. Engine generates MonBus error packet (type=0xE1, ch_id, error details)

3. Packet routed to error FIFO in monbus_axil_group
 4. IRQ asserted (`irq_out = 1`)
 5. Software ISR reads `ERROR_PKT_LOW/HIGH` via AXIL slave
 6. Software logs error, takes recovery action
 7. Error FIFO drains, IRQ deasserts
-

Testing

Test Location: `projects/components/stream/dv/tests/integration_tests/`

Test Scenarios: 1. Normal packet logging to memory 2. Error packet routing to error FIFO 3. Interrupt assertion/deassertion 4. AXIL slave reads (error FIFO) 5. AXIL master writes (memory logging) 6. Multi-channel packet arbitration

Performance

Throughput: 1 packet per cycle (per channel)

Latency: - Error FIFO: 2 cycles (write to AXIL readable) - Memory logging: 4-6 cycles (AXIL master write latency)

Area: ~1000 LUTs + 64 64-bit FIFO

Related Documentation

- **MonBus Protocol:** `rtl/amba/includes/monitor_pkg.sv`
 - **Source:** `rtl/stream_macro/monbus_axil_group.sv`
-

Last Updated: 2025-10-17

Chapter 3: External Interfaces

This chapter documents the external interfaces of the STREAM DMA engine.

Interface Specifications

STREAM has five 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 Master Interface (Future)**
- Used for MonBus packet logging to memory
- 32-bit data width
- Part of monbus_axil_group integration

[03_apb_interface_spec.md](#)

- **APB Programming Interface**
- Simplified per-channel descriptor kick-off
- Valid/ready handshake protocol
- Not a full APB slave (no PSEL/PENABLE)

[05_monbus_interface_spec.md](#)

- **MonBus Output Protocol**
 - Unified 64-bit monitoring bus
 - Event packet format and encoding
 - Agent ID assignments for STREAM components
 - Integration with downstream consumers
-

STREAM Interface Summary

Interface	Type	Direction	Width	Purpose
APB Programming	Custom	Input	64-bit addr × 8 ch	Descriptor kick-off
AXI4 Descriptor	Master Read	Output	256-bit	Descriptor fetch
AXI4 Data Read	Master Read	Output	512-bit (param)	Source data read

Interface	Type	Direction	Width	Purpose
AXI4 Data Write	Master Write	Output	512-bit (param)	Destination data write
MonBus	Monitor Bus	Output	64-bit	Event reporting

Note: AXIL4 interface is part of the optional monbus_axil_group and not directly exposed at the STREAM Core level.

Last Updated: 2025-11-22

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
Descriptor Fetch	Master Read	256-bit (fixed)	AR, R	Fetch descriptors from memory
Data Read	Master Read	512-bit (param)	AR, R	Read source data to SRAM
Data Write	Master Write	512-bit (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
AXI4 Master Read-Split	AR, R	Flexible	4-byte	Yes	Yes	Data interface with chunk enables
AXI4	AW,	Flexible	4-byte	Yes	Yes	Data

Interface Group	Channels	Transfer Mode	Address Alignment	Monitor	DCG	Notes
Master Write-Split	W, B					a inte rfac es wit h chu nk ena bles
AXI4 Master Read	AR, R	Simplified	Bus-width	No	Yes	Con trol inte rfac es, full y alig ned
AXI4 Master Write	AW, W, B	Simplified	Bus-width	Yes	Yes	Con trol inte rfac es, full y alig ned

Interface Group Parameter Settings

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

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

Interface Configuration Summary

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

Interface Type	Interface Group	Transfer Mode	Alignment	Notes
Flag Sink	AXI4 Master Read	Simplified	32-bit aligned	Control interface
Flag Source	AXI4 Master Read	Simplified	32-bit aligned	Control interface

Transfer Mode Specifications

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

Mode 1: Simplified Transfer Mode (Control Interfaces)

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

Simplified Mode Assumptions

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

Mode 2: Flexible Transfer Mode (Data Interfaces)

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

Flexible Mode Assumptions

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

Aspect	Requirement
	utilization

Mode 1: Simplified Transfer Mode Specification

Assumption 1: Address Alignment to Data Bus Width

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

Assumption 2: Fixed Transfer Size

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

Mode 2: Flexible Transfer Mode Specification

Assumption 1: 4-Byte Address Alignment

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

Assumption 2: Multiple Transfer Sizes

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

Assumption 3: Progressive Alignment Strategy

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

Assumption 4: Chunk Enable Support

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

Common Protocol Assumptions (Both Modes)

Assumption 1: Incrementing Bursts Only

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

Assumption 2: No Address Wraparound

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

Flexible Mode: Address Calculation Examples

Progressive Alignment Examples

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

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

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

Step	Address	Size	AxSIZE	Length	Bytes Transferred	Notes
1	0x1010	16 bytes	3'b100	1 beat	16	Optimal initial size
2	0x1020	32 bytes	3'b101	1 beat	32	Progressive alignment
3	0x1040	64	3'b110	N beats	64×N	Optim

Step	Address	Size	AxSIZE	Length	Bytes Transferred	Notes
		bytes				al transf ers

Chunk Enable Pattern Examples

512-bit Bus with 16×32-bit chunks

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

Master Read Interface Specification

Read Address Channel (AR)

Signal	Width	Direction	Simplified Mode	Flexible Mode	Description
ar_addr	ADDR_WIDTH	Master→Slave	Bus-width aligned	4-byte aligned	Read address
ar_len	8	Master→Slave	0-255	0-255	Burst length - 1
ar_size	3	Master→Slave	Fixed per bus	Variable: 4-64 bytes	Transfer size
ar_burst	2	Master→Slave	2'b01 (INCR only)	2'b01 (INCR only)	Burst type
ar_id	ID_WIDTH	Master→Slave	Any	Any	Transaction ID
ar_lock	1	Master→Slave	1'b0	1'b0	Lock type (normal)
ar_cach	4	Master→Slave	Implementation specific	4'b0011	Cache attributes
ar_prot	3	Master→Slave	Implementation	3'b000	Protection

Signal	Width	Direction	Simplified Mode	Flexible Mode	Description
		ave	specific		attributes
ar_qos	4	Master→Slave	4'b0000 ave	4'b0000	Quality of Service
ar_region	4	Master→Slave	4'b0000 ave	4'b0000	Region identifier
ar_user	USER_WIDTH	Master→Slave	Optional ave	Optional	User-defined
ar_val_id	1	Master→Slave	0 or 1 ave	0 or 1	Address valid
ar_ready	1	Slave→Master	0 or 1	0 or 1	Address ready

Read Data Channel (R)

Signal	Width	Direction	Description
r_data	DATA_WIDTH	Slave→Master	Read data
r_id	ID_WIDTH	Slave→Master	Transaction ID
r_resp	2	Slave→Master	Read response
r_last	1	Slave→Master	Last transfer in burst
r_user	USER_WIDTH	Slave→Master	User-defined (optional)
r_valid	1	Slave→Master	Read data valid
r_ready	1	Master→Slave	Read data ready

Master Write Interface Specification

Write Address Channel (AW)

Signal	Width	Direction	Simplified Mode	Flexible Mode	Description
aw_addr	ADDR_WIDTH	Master→Slave	Bus-width aligned	4-byte aligned	Write address

Signal	Width h	Direction	Simplified Mode	Flexible Mode	Description
aw_len	8	Master→Slave	0-255	0-255	Burst length - 1
aw_size	3	Master→Slave	Fixed per bus	Variable: 4-64 bytes	Transfer size
aw_burst	2	Master→Slave	2'b01 (INCR only)	2'b01 (INCR only)	Burst type
aw_id	ID_WIDTH	Master→Slave	Any	Any	Transaction ID
aw_lock	1	Master→Slave	1'b0	1'b0	Lock type (normal)
aw_cache	4	Master→Slave	Implementation specific	4'b0011	Cache attributes
aw_prot	3	Master→Slave	Implementation specific	3'b000	Protection attributes
aw_qos	4	Master→Slave	4'b0000	4'b0000	Quality of Service
aw_region	4	Master→Slave	4'b0000	4'b0000	Region identifier
aw_user	USER_WIDTH	Master→Slave	Optional	Optional	User-defined
aw_valid	1	Master→Slave	0 or 1	0 or 1	Address valid
aw_ready	1	Slave→Master	0 or 1	0 or 1	Address ready

Write Data Channel (W)

Signal	Width h	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

Signal	Width	Direction	Simplified Mode	Flexible Mode	Description
w_user	USER_WIDTH	Master→Slave	Optional ave	Optional	in burst User-defined
w_val	1	Master→Slave	0 or 1 ave	0 or 1	Write data valid
w_ready	1	Slave→Master	0 or 1	0 or 1	Write data ready

Write Response Channel (B)

Signal	Width	Direction	Description
b_id	ID_WIDTH	Slave→Master	Transaction ID
b_resp	2	Slave→Master	Write response
b_user	USER_WIDTH	Slave→Master	User-defined (optional)
b_valid	1	Slave→Master	Response valid
b_ready	1	Master→Slave	Response ready

Address Calculation Rules

Simplified Mode Address Generation

Parameter	Formula	Description
First Address	Must be bus-width aligned	Starting address
Address N	First_Address + (N × Bus_Width_Bytes)	Address for beat N
Alignment Check	(Address % Bus_Width_Bytes) == 0	Must always be true

Flexible Mode Address Generation

Parameter	Formula	Description
First Address	Must be 4-byte aligned	Starting address

Parameter	Formula	Description
Address N	First_Address + (N × Transfer_Size)	Address for beat N
Alignment Check	(Address % 4) == 0	Must always be true
Progressive Alignment	Choose largest size \leq bytes_to_boundary	Optimization strategy

4KB Boundary Considerations (Both Modes)

Validation Rule	Formula	Description
4KB Boundary	Bursts cannot cross 4KB (0x1000) boundaries	AXI specification
Max Burst Calculation	Max_Beats = (4KB - (Start_Address % 4KB)) / Transfer_Size	Burst limit
Boundary Check	Verify no 4KB crossings in burst	Mandatory validation

Write Strobe Generation

Simplified Mode Strobe Generation

Bus Width	Strobe Pattern	Description
32-bit	4'b1111	All bytes valid
512-bit	64'hFFFFFFFFFFFF FFF	All bytes valid

Flexible Mode Strobe Generation

From Chunk Enables (512-bit bus example):

```
// Convert 16×32-bit chunk enables to 64×8-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 000000F	First 4 bytes
16 bytes	16'h000F	64'h0000000000 00000FF	First 16 bytes
32 bytes	16'h00FF	64'h0000000000 0FFFFFF	First 32 bytes
64 bytes	16'hFFFF	64'hFFFFFFFFF FFFFFFF	All 64 bytes

Response Codes

Response Code Specification

Value	Name	Description	Simplified Mode Usage	Flexible Mode Usage
2'b00	OKA Y	Normal access success	Bus-width aligned access	4-byte aligned access
2'b01	EXO KAY	Exclusive access success	Bus-width aligned exclusive	4-byte aligned exclusive
2'b10	SLVE RR	Slave error	Slave-specific error	Slave-specific error
2'b11	DECE RR	Decode error	Bus-width misalignment	4-byte misalignment

Implementation Benefits

Simplified Mode Benefits

Benefit Area	Simplification	Impact
Address Generation	Simple increment by bus width	Minimal logic complexity

Benefit Area	Simplification	Impact
Size Checking	No dynamic size validation	No validation logic needed
Strobe Generation	All strobes always high	Trivial implementation
Timing	Predictable single-size transfers	Optimal timing closure

Flexible Mode Benefits

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

Mode Selection Guidelines

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

Validation Requirements

Simplified Mode Validation

Validation Area	Requirements
Address Alignment	Verify all addresses aligned to full bus width
Fixed Size	Verify AxSIZE always matches DATA_WIDTH
Full Strobes	Verify w_strb is always all 1's
Burst Type	Verify AxBURST is always 2'b01

Flexible Mode Validation

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

Common Validation

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

Performance Characteristics

Simplified Mode Performance

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

Flexible Mode Performance

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

Performance Optimization Strategy

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

This dual-mode approach provides the best of both worlds: simplified, predictable operation for control interfaces and flexible, high-performance operation for data interfaces.

AXI4-Lite Interface Specification and Assumptions

Overview

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

Interface Summary

Number of Interfaces

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

Interface Parameters

Parameter	Description	Valid Values	Default
DATA_WIDTH	AXI data bus width in bits	32, 64	32, 64
ADDR_WIDTH	AXI address bus width in bits	32, 64	37
STRB_WIDTH	Write strobe width	DATA_WIDTH/8	8

Core Protocol Assumptions

Inherent AXI4-Lite Simplifications

AXI4-Lite protocol inherently provides the following constraints:

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

Implementation Assumptions

Assumption 1: Address Alignment to Data Bus Width

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

Assumption 2: Fixed Transfer Size

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

Assumption 3: No Address Wraparound

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

Assumption 4: Standard Protection Attributes

Access Type	AxPROT Value	Description
Normal Access	3'b000	Data, secure, unprivileged
Privileged Access	3'b001	Data, secure, privileged

Access Type	AxPROT Value	Description
Rationale		Covers the majority of control register access patterns

Master Read Interface Specification

Read Address Channel (AR)

Signal	Width	Direction	Required Values	Description
ar_addr	ADDR_WI DTH	Master→Slave	8-byte aligned	Read address
ar_prot	3	Master→Slave	Implementation specific	Protection attributes
ar_valid	1	Master→Slave	0 or 1	Address valid
ar_ready	1	Slave→Master	0 or 1	Address ready

Read Data Channel (R)

Signal	Width	Direction	Description
r_data	64	Slave→Master	Read data
r_resp	2	Slave→Master	Read response
r_valid	1	Slave→Master	Read data valid
r_ready	1	Master→Slave	Read data ready

AXI4-Lite Simplifications (Read)

Removed Signal	AXI4 Usage	AXI4-Lite Reason
ar_id	Transaction ID	Single transfers, no transaction IDs
ar_len	Burst length	Single transfers only
ar_size	Transfer size	Fixed to bus width
ar_burst	Burst type	Single transfers only
ar_lock	Lock type	Simplified access model
ar_cache	Cache attributes	Simplified memory model
ar_qos	Quality of Service	Simplified priority

Removed Signal	AXI4 Usage	AXI4-Lite Reason
		model
ar_region	Region identifier	Simplified address space
ar_user	User-defined	Simplified interface
r_id	Transaction ID	No transaction IDs needed
r_last	Last transfer	Single transfers only
r_user	User-defined	Simplified interface

Master Write Interface Specification

Write Address Channel (AW)

Signal	Width	Direction	Required Values	Description
aw_addr	ADDR_WI DTH	Master→Slave	8-byte aligned	Write address
aw_prot	3	Master→Slave	Implementation specific	Protection attributes
aw_valid	1	Master→Slave	0 or 1	Address valid
aw_ready	1	Slave→Master	0 or 1	Address ready

Write Data Channel (W)

Signal	Width	Direction	Description
w_data	32	Master→Slave	Write data
w_strb	4	Master→Slave	Write strobes (byte enables)
w_valid	1	Master→Slave	Write data valid
w_ready	1	Slave→Master	Write data ready

Write Response Channel (B)

Signal	Width	Direction	Description
b_resp	2	Slave→Master	Write response
b_valid	1	Slave→Master	Response valid
b_ready	1	Master→Slave	Response

Signal	Width	Direction	Description
			ready

AXI4-Lite Simplifications (Write)

Removed Signal	AXI4 Usage	AXI4-Lite Reason
aw_id	Transaction ID	Single transfers, no transaction IDs
aw_len	Burst length	Single transfers only
aw_size	Transfer size	Fixed to bus width
aw_burst	Burst type	Single transfers only
aw_lock	Lock type	Simplified access model
aw_cache	Cache attributes	Simplified memory model
aw_qos	Quality of Service	Simplified priority model
aw_region	Region identifier	Simplified address space
aw_user	User-defined	Simplified interface
w_last	Last transfer	Single transfers only
w_user	User-defined	Simplified interface
b_id	Transaction ID	No transaction IDs needed
b_user	User-defined	Simplified interface

Address Requirements

Address Alignment Rules

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

Address Validation Examples

Address Category	Examples	Status
Valid (8byte aligned)	0x1000, 0x1004, 0x1008, 0x100C	Accepted
Invalid (unaligned)	0x1001, 0x1002, 0x1003	DECERR response

Response Codes

Response Code Specification

Value	Name	Description	Usage in Control Registers
2'b00	OKAY	Normal access success	Successful register access
2'b01	EXOK AY	Exclusive access success	Not used in AXI4-Lite
2'b10	SLVER R	Slave error	Invalid register access
2'b11	DECER R	Decode error	Address decode failure or misalignment

Response Usage Guidelines

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

Protection Signal Usage

Protection Signal Encoding

Bit	Name	Description	Recommended Usage
[0]	Privilege d	0=Normal, 1=Privileged	Set based on processor mode
[1]	Non-	0=Secure, 1=Non-	Set based on security domain

Bit	Name	Description	Recommended Usage
	secure	secure	
[2]	Instruction	0=Data, 1=Instruction on	Always 0 for control registers

Common Protection Patterns

Pattern	AxPROT Value	Description
Normal Data Access	3'b000	Standard register access
Privileged Data Access	3'b001	Privileged register access
Debug Access	3'b010	Debug register access
Privileged Debug	3'b011	Privileged debug access

Implementation Benefits

Simplified Control Register Interface

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

Address Decode Implementation

Implementation Aspect	Method	Benefit
4-byte Alignment Check	$\text{addr}[1:0] == 2'b00$	Simple bit masking
Address Range Check	$\text{addr} \geq \text{base} \&\& \text{addr} \leq \text{end}$	Simple comparisons

Implementation Aspect	Method	Benefit
Combined Check	limit alignment_ok && range_ok	Single decode decision

Error Generation Logic

Error Condition	Check	Response
Address Misalignment	addr[1:0] != 2'b00	Generate DECERR
Address Out of Range	! addr_in_range(addr)	Generate DECERR
Register Error	register_error_condition	Generate SLVERR
Normal Access	All checks pass	Generate OKAY

Timing Requirements

Handshake Protocol

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

Channel Dependencies

Dependency	Requirement	Description
Write Channels	AW and W channels are independent	Can be presented in any order
Write Response	B channel waits for both AW and W completion	Response dependency

Dependency	Requirement	Description
Read Channels	R channel waits for AR channel completion	Response dependency
Transaction Ordering	Multiple outstanding transactions not supported	Inherent AXI4-Lite limitation

Reset Behavior

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

Validation Requirements

Functional Validation

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

Timing Validation

Validation Area	Requirements
Setup/Hold	Verify signal timing requirements
Reset Behavior	Verify proper reset sequence

Validation Area	Requirements
Back-pressure	Verify ready signal behavior under load

Error Injection Testing

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

Example Transactions

64-bit Register Write

Parameter	Value	Description
Bus Width	64 bits (8 bytes)	Data bus configuration
Target Address	0x1000 (8-byte aligned)	Valid aligned address
Write Data	0xDEADBEEFCA FEBABE	64-bit data value
Required Settings	aw_addr=0x100 0, aw_prot=3'b000, w_data=0xDEAD BEEFCAFEBABE, w_strb=8'b1111 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, w_data=0xDEADBEEFCAFEBABE
3	Wait for	aw_ready=1, w_ready=1

Step	Action	Signal States
	handshakes	
4	Wait for response	b_valid=1, b_resp=OKAY
5	Complete transaction	b_ready=1

64-bit Register Read

Parameter	Value	Description
Bus Width	64 bits (8 bytes)	Data bus configuration
Target Address	0x1008 (8-byte aligned)	Valid aligned address
Required Settings	ar_addr=0x1008 , ar_prot=3'b000	Transaction configuration

AR Transaction Flow

Step	Action	Signal States
1	Assert ar_valid with address	ar_valid=1, ar_addr=0x1008
2	Wait for address handshake	ar_ready=1
3	Wait for data response	r_valid=1, r_resp=OKAY
4	Complete transaction	r_ready=1
5	Capture data	r_data (64 bits)

Misaligned Address Example

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

Parameter	Value	Description
	register access	

Common Use Cases

Typical Applications

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

Performance Considerations

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

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
Programming	apb_valid[ch], apb_ready[ch], 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
Alignment Rule	All APB transfers are aligned to natural word boundaries
8-bit transfers	Byte aligned (no restriction)
16-bit transfers	2-byte aligned ($\text{PADDR}[0] = 0$)
32-bit transfers	4-byte aligned ($\text{PADDR}[1:0] = 2'b00$)
Rationale	Simplifies peripheral decode logic and ensures efficient access

Assumption 2: Standard Transfer Sizes

Bus Width	Supported Transfer Sizes	Rationale
32-bit bus	8-bit, 16-bit, 32-bit	Covers typical register access patterns
16-bit bus	8-bit, 16-bit	Keeps decode logic simple
8-bit bus	8-bit only	Minimal complexity

Assumption 3: Single-Cycle Default Operation

Aspect	Requirement
Default Behavior	Most peripherals respond in a single cycle (PREADY tied high)

Aspect	Requirement
Optimization	PREADY optimization for simple registers
Exception	Complex peripherals may use PREADY for wait states
Rationale	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	Aligned per transfer size	Peripheral address (2KB space)
src_sel	1	Master→Slave	0 or 1	Peripheral select
src_enab le	1	Master→Slave	0 or 1	Enable signal
src_writ e	1	Master→Slave	0 or 1	Write enable (1=write, 0=read)
snk_addr	11 bits	Master→Slave	Aligned per transfer size	Peripheral address (2KB space)
snk_sel	1	Master→Slave	0 or 1	Peripheral select
snk_enab le	1	Master→Slave	0 or 1	Enable signal
snk_writ e	1	Master→Slave	0 or 1	Write enable (1=write, 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
Address Width	11 bits (src_addr[10:0])	Full address space
Address Space	2KB (2048 bytes)	Total addressable space
Register Alignment	32-bit aligned (4-byte boundaries)	Address constraints
Usable Addresses	0x000 to 0x7FF	Valid address range

Address Decode Implementation

Register	Address Bits	Address Value	Description
reg0	src_addr[10:2] == 9'h000	0x000	First register
reg1	src_addr[10:2] == 9'h001	0x004	Second register
reg2	src_addr[10:2] == 9'h002	0x008	Third register
...	Up to 512 32-bit registers

Transaction Protocol

2-Phase Transaction States

Phase	State	Duration	Signal Requirements
Setup (T1)	SETUP	1 clock cycle	src_sel=1, src_enable=0, address/data stable
Access (T2+)	ACCESS	1+ clock cycles	src_sel=1, src_enable=1, wait for src_ready
Idle	IDLE	Variable	src_sel=0, src_enable=0

State Transitions

Current State	Next State	Condition	Description
IDLE	SETUP	Transaction start	Master drives address and control
SETUP	ACCESS	Clock edge	Master asserts src_enable
ACCESS	IDLE	src_ready=1	Transaction completes
ACCESS	ACCESS	src_ready=0	Wait state continues

Transaction Timing Requirements

Timing Parameter	Requirement	Description
Setup Time	All master signals stable before rising src_clk	Signal stability
Hold Time	All master signals held after rising src_clk	Signal stability
Output Delay	Slave outputs valid after rising src_clk	Response timing

Advanced Features

Dynamic Clock Gating

Feature	Description	Benefit
Automatic Power Reduction	Clocks gated when no activity detected	Significant power savings
Configurable Idle Threshold	Programmable idle count before gating	Tunable power/performance
Multi-Domain Support	Independent gating for APB (PCLK) and backend (ACLK)	Flexible power management
Graceful Handoff	Ready signals forced to zero during gating	Protocol compliance
Activity Detection	Monitors valid signals across interfaces	Intelligent gating control

Clock Domain Crossing (CDC) Handshaking

Feature	Description	Benefit
Arbitrary Frequency Support	Works across any clock frequency ratios	Design flexibility
Robust Reliability	Proven handshaking with proper synchronization	Zero data loss
Command/Response Separation	Independent CDC paths	Maximum throughput
Skid Buffer Integration	Internal buffering prevents backpressure	Smooth operation
Deterministic Latency	Predictable timing characteristics	System predictability

Buffering and Flow Control

Feature	Description	Benefit
Internal Skid Buffers	Configurable depth buffering	Improved performance
Backpressure Handling	Proper ready/valid handshaking	Flow control
Command Pipelining	Backend processes while APB handles responses	Efficiency

Feature	Description	Benefit
Response Queuing	Responses buffered for varying latencies	Latency tolerance

Error Handling

Error Response Specification

Condition	src_slverr	Description
Successful access	0	Normal completion
Address decode error	1	Invalid address
Protection violation	1	Insufficient privilege
Backend error	1	Downstream error condition

Error Response Timing

Timing Requirement	Description
src_slverr Validity	Must be valid when src_ready is asserted
Error Completion	Error response completes transaction normally
Master Responsibility	Master must check src_slverr status

Reset Behavior

Reset Requirements

Reset Phase	Signal Requirements	Description
Reset Assertion	All outputs to known states	Deterministic reset
Reset Deassertion	src_sel=0, src_enable=0 in first cycle	Clean startup
Reset Values	src_ready=0, src_slverr=0, src_rdata=0	Default states

Implementation Variants

Available Implementations

Variant	Features	Use Case
Basic APB Slave	Single clock domain, internal buffering	Simple peripherals
CDC-Enabled APB Slave	Dual clock domain (PCLK/ACLK)	Mixed-frequency systems
Power-Optimized APB Slave	Dynamic clock gating	Power-sensitive designs

Performance Characteristics

Latency Specifications

Metric	Value	Description
Minimum Read Latency	2 clock cycles	Setup + access phases
Minimum Write Latency	2 clock cycles	Setup + access phases
CDC Additional Latency	2-6 clock cycles	Depends on clock relationships
Buffer Latency	Minimal	Skid buffer design

Throughput Specifications

Metric	Value	Description
Single Transaction	2 clocks minimum	Basic transaction time
Back-to-back Transactions	Limited by src_ready	Depends on peripheral
CDC Throughput	Maintained across domains	Proper buffering

Power Consumption

Power Category	Characteristics	Description
Active Power	Standard CMOS logic	Normal operation
Idle Power	Significantly reduced with clock gating	Power optimization
Gating Overhead	Minimal additional logic	Efficient

Power Category	Characteristics	Description
		implementation

Validation Requirements

Functional Validation

Validation Area	Requirements
Protocol Compliance	Verify 2-phase setup/access sequence
Address Decode	Verify 11-bit address handling and alignment
Error Response	Verify src_slver generation and handling
Wait States	Verify src_ready functionality
CDC Operation	Verify cross-clock domain transfers
Clock Gating	Verify power management behavior
Buffer Operation	Verify skid buffer and flow control

Timing Validation

Validation Area	Requirements
Setup/Hold	Verify signal timing requirements
Reset Sequence	Verify proper reset behavior
Multi-Clock	Verify CDC timing across frequency ranges
Gating Transitions	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	Setu

Clock Cycle	src_sel	src_enabl e	src_writ e	src_add r	src_wda ta	src_read y	Phase
2	1	1	1	0x100	0xABCD	1	p 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	Phase
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 identifier width	8	8
EVENT_DATA_WIDTH	Event data width	35	35

Core Design Assumptions

Assumption 1: Hierarchical Event Organization

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

Assumption 2: Protocol Isolation

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

Assumption 3: Two-Tier Memory Architecture

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

Assumption 4: Configurable Packet Routing

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

Interface Signal Specification

Monitor Bus Output Interface

Signal	Width	Direction	Description
mon_packet	64	Monitor→System	Monitor packet data
mon_valid	1	Monitor→System	Packet valid signal
mon_ready	1	System→Monitor	Ready to accept packet
mon_error	1	Monitor→System	Monitor error condition

Protocol Input Interfaces

Signal	Width	Direction	Description
axi_event	64	AXI Monitor→Bus	AXI event packet
axi_event_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 Monitor→Bus	MNOC event packet
mnoc_event_val_id	1	MNOC Monitor→Bus	MNOC event valid
mnoc_event_ready	1	Bus→MNOC Monitor	Ready for MNOC event
arb_event	64	ARB Monitor→Bus	ARB event packet
arb_event_val	1	ARB	ARB event

Signal	Width	Direction	Description
id		Monitor→Bus	valid
arb_event_ready	1	Bus→ARB Monitor	Ready for ARB event
core_event	64	CORE Monitor→Bus	CORE event packet
core_event_valid	1	CORE Monitor→Bus	CORE event valid
core_event_ready	1	Bus→CORE Monitor	Ready for CORE event

Control and Status Signals

Signal	Width	Direction	Description
clk	1	Input	System clock
resetn	1	Input	Active-low reset
monitor_enable	1	Input	Global monitor enable
packet_type_enables	16	Input	Per-type enable bits
local_memory_full	1	Output	Local memory full flag
external_memory_error	1	Output	External memory error

Packet Format and Field Allocation

64-bit Monitor Bus Packet Structure

Field	Bits	Width	Description
Packet Type	[63:60]	4	Event category (Error, Completion, etc.)
Protocol	[59:57]	3	Bus protocol (AXI=0, MNOC=1, APB=2, ARB=3, CORE=4)
Event Code	[56:53]	4	Specific events within category
Channel ID	[52:47]	6	Transaction/channel

Field	Bits	Width	Description
			identifier
Unit ID	[46:43]	4	Subsystem identifier
Agent ID	[42:35]	8	Module identifier
Event Data	[34:0]	35	Event-specific payload

Packet Type Definitions

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

Protocol-Specific Event Codes

AXI Protocol Events

Error Events (*PktTypeError* + *PROTOCOL_AXI*)

Code	Event Name	Description
0x0	AXI_ERR_RESP_SLVE RR	Slave error response
0x1	AXI_ERR_RESP_DECE RR	Decode error response
0x2	AXI_ERR_DATA_ORP HAN	Data without command
0x3	AXI_ERR_RESP_ORP HAN	Response without transaction
0x4	AXI_ERR_PROTOCOL	Protocol violation
0x5	AXI_ERR_BURST_LE NGTH	Invalid burst length
0x6	AXI_ERR_BURST_SIZ E	Invalid burst size
0x7	AXI_ERR_BURST_TYP E	Invalid burst type
0x8	AXI_ERR_ID_COLLISI ON	ID collision detected
0x9	AXI_ERR_WRITE_BE FORE_ADDR	Write data before address
0xA	AXI_ERR_RESP_BEFO RE_DATA	Response before data complete
0xB	AXI_ERR_LAST_MISS ING	Missing LAST signal
0xC	AXI_ERR_STROBE_E RROR	Write strobe error
0xD	AXI_ERR_RESERVED _D	Reserved
0xE	AXI_ERR_RESERVED _E	Reserved
0xF	AXI_ERR_USER_DEFI	User-defined error

Code	Event Name	Description
	NED	

Timeout Events (PktTypeTimeout + PROTOCOL_AXI)

Code	Event Name	Description
0x0	AXI_TIMEOUT_CMD	Command/Address timeout
0x1	AXI_TIMEOUT_DATA	Data timeout
0x2	AXI_TIMEOUT RESP	Response timeout
0x3	AXI_TIMEOUT_HAN DSHAKE	Handshake timeout
0x4	AXI_TIMEOUT_BURS T	Burst completion timeout
0x5	AXI_TIMEOUT_EXCL USIVE	Exclusive access timeout
0x6-0xE	Reserved	Future expansion
0xF	AXI_TIMEOUT_USER _DEFINED	User-defined timeout

Performance Events (PktTypePerf + PROTOCOL_AXI)

Code	Event Name	Description
0x0	AXI_PERF_ADDR_LA TENCY	Address phase latency
0x1	AXI_PERF_DATA_LA TENCY	Data phase latency
0x2	AXI_PERF_RESP_LAT ENCY	Response phase latency
0x3	AXI_PERF_TOTAL_L ATENCY	Total transaction latency
0x4	AXI_PERF_THROUG HPUT	Transaction throughput
0x5	AXI_PERF_ERROR_R ATE	Error rate
0x6	AXI_PERF_ACTIVE_C OUNT	Active transaction count
0x7	AXI_PERF_BANDWI	Bandwidth

Code	Event Name	Description
	DTH_UTIL	utilization
0x8	AXI_PERF_QUEUE_DEPTH	Average queue depth
0x9	AXI_PERF_BURST_EFFICIENCY	Burst efficiency metric
0xA-0xE	Reserved	Future expansion
0xF	AXI_PERF_USER_DEFINED	User-defined performance

APB Protocol Events

Error Events (PktTypeError + PROTOCOL_APB)

Code	Event Name	Description
0x0	APB_ERR_PSLVERR	Peripheral slave error
0x1	APB_ERR_SETUP_VIOLATION	Setup phase protocol violation
0x2	APB_ERR_ACCESS_VIOLATION	Access phase protocol violation
0x3	APB_ERR_STROBE_ERROR	Write strobe error
0x4	APB_ERR_ADDR_DECODE	Address decode error
0x5	APB_ERR_PROT_VIOLATION	Protection violation (PPROT)
0x6	APB_ERR_ENABLE_ERROR	Enable phase error
0x7	APB_ERR_READY_ERROR	PREADY protocol error
0x8-0xE	Reserved	Future expansion
0xF	APB_ERR_USER_DEFINED	User-defined error

Timeout Events (PktTypeTimeout + PROTOCOL_APB)

Code	Event Name	Description
0x0	APB_TIMEOUT_SETUP	Setup phase timeout
0x1	APB_TIMEOUT_ACCESS	Access phase timeout
0x2	APB_TIMEOUT_ENABLE_BLE	Enable phase timeout (PREADY stuck)
0x3	APB_TIMEOUT_PREADYSTUCK	PREADY stuck low
0x4	APB_TIMEOUT_TRANSFER	Overall transfer timeout
0x5-0xE	Reserved	Future expansion
0xF	APB_TIMEOUT_USER_DEFINED	User-defined timeout

Completion Events (PktTypeCompletion + PROTOCOL_APB)

Code	Event Name	Description
0x0	APB_COMPL_TRANSACTION_COMPLETE	Transaction completed
0x1	APB_COMPL_READ_COMPLETE	Read transaction complete
0x2	APB_COMPL_WRITE_COMPLETE	Write transaction complete
0x3-0xE	Reserved	Future expansion
0xF	APB_COMPL_USER_DEFINED	User-defined completion

Threshold Events (PktTypeThreshold + PROTOCOL_APB)

Code	Event Name	Description
0x0	APB_THRESH_LATE_NCY	APB latency threshold
0x1	APB_THRESH_ERROR_RATE	APB error rate threshold
0x2	APB_THRESH_ACCESS	Access count

Code	Event Name	Description
0x3	S_COUNT	threshold
0x4-0xE	APB_THRESH_BAND WIDTH	Bandwidth threshold
0xF	Reserved	Future expansion
	APB_THRESH_USER_ DEFINED	User-defined threshold

Performance Events (PktTypePerf + PROTOCOL_APB)

Code	Event Name	Description
0x0	APB_PERF_READ_LA TENCY	Read transaction latency
0x1	APB_PERF_WRITE_L ATENCY	Write transaction latency
0x2	APB_PERF_THROUG HPUT	Transaction throughput
0x3	APB_PERF_ERROR_R ATE	Error rate
0x4	APB_PERF_ACTIVE_C OUNT	Active transaction count
0x5	APB_PERF_COMPLE TED_COUNT	Completed transaction count
0x6-0xE	Reserved	Future expansion
0xF	APB_PERF_USER_DE FINED	User-defined performance

Debug Events (PktTypeDebug + PROTOCOL_APB)

Code	Event Name	Description
0x0	APB_DEBUG_STATE_ CHANGE	APB state changed
0x1	APB_DEBUG_SETUP_ PHASE	Setup phase event
0x2	APB_DEBUG_ACCESS _PHASE	Access phase event
0x3	APB_DEBUG_ENABL E_PHASE	Enable phase event

Code	Event Name	Description
0x4	APB_DEBUG_PSEL_T RACE	PSEL trace
0x5	APB_DEBUG_PENAB LE_TRACE	PENABLE trace
0x6	APB_DEBUG_PREAD Y_TRACE	PREADY trace
0x7	APB_DEBUG_PPROT_ TRACE	PPROT trace
0x8	APB_DEBUG_PSTRB_ TRACE	PSTRB trace
0x9-0xE	Reserved	Future expansion
0xF	APB_DEBUG_USER_D	User-defined debug DEFINED

MNOC Protocol Events

Error Events (PktTypeError + PROTOCOL_MNOC)

Code	Event Name	Description
0x0	MNOC_ERR_PARITY	Parity error
0x1	MNOC_ERR_PROTOC OL	Protocol violation
0x2	MNOC_ERR_OVERFL OW	Buffer/Credit overflow
0x3	MNOC_ERR_UNDERF LOW	Buffer/Credit underflow
0x4	MNOC_ERR_ORPNA N	Orphaned packet/ACK
0x5	MNOC_ERR_INVALI D	Invalid type/channel/paylo d
0x6	MNOC_ERR_HEADER _CRC	Header CRC error
0x7	MNOC_ERR_PAYLOA D_CRC	Payload CRC error
0x8	MNOC_ERR_SEQUEN	Sequence number

Code	Event Name	Description
	CE	error
0x9	MNOC_ERR_ROUTE	Routing error
0xA	MNOC_ERR_DEADLOCK	Deadlock detected
	CK	
0xB-0xE	Reserved	Future expansion
0xF	MNOC_ERR_USER_DEFINED	User-defined error

Credit Events (PktTypeCredit + PROTOCOL_MNOC)

Code	Event Name	Description
0x0	MNOC_CREDIT_ALLLOCATED	Credits allocated
0x1	MNOC_CREDIT_CONSUMED	Credits consumed
0x2	MNOC_CREDIT_RETURNED	Credits returned
0x3	MNOC_CREDIT_OVERFLOW	Credit overflow detected
0x4	MNOC_CREDIT_UNDERFLOW	Credit underflow detected
0x5	MNOC_CREDIT_EXHAUSTED	All credits exhausted
0x6	MNOC_CREDIT_RESTORED	Credits restored
0x7	MNOC_CREDIT_EFFICIENCY	Credit efficiency metric
0x8	MNOC_CREDIT_LEAK	Credit leak detected
0x9-0xE	Reserved	Future expansion
0xF	MNOC_CREDIT_USER_DEFINED	User-defined credit event

Channel Events (PktTypeChannel + PROTOCOL_MNOC)

Code	Event Name	Description
0x0	MNOC_CHANNEL_OPENED	Channel opened

Code	Event Name	Description
	PEN	
0x1	MNOC_CHANNEL_CL OSE	Channel closed
0x2	MNOC_CHANNEL_ST ALL	Channel stalled
0x3	MNOC_CHANNEL_R ESUME	Channel resumed
0x4	MNOC_CHANNEL_C ONGESTION	Channel congestion detected
0x5	MNOC_CHANNEL_P RIORITY	Channel priority change
0x6-0xE	Reserved	Future expansion
0xF	MNOC_CHANNEL_U SER_DEFINED	User-defined channel event

Stream Events (PktTypeStream + PROTOCOL_MNOC)

Code	Event Name	Description
0x0	MNOC_STREAM_STA RT	Stream started
0x1	MNOC_STREAM_EN D	Stream ended (EOS)
0x2	MNOC_STREAM_PAU SE	Stream paused
0x3	MNOC_STREAM_RES UME	Stream resumed
0x4	MNOC_STREAM_OVE RFLOW	Stream buffer overflow
0x5	MNOC_STREAM_UN DERFLOW	Stream buffer underflow
0x6-0xE	Reserved	Future expansion
0xF	MNOC_STREAM_USE R_DEFINED	User-defined stream event

ARB Protocol Events

Error Events (PktTypeError + PROTOCOL_ARB)

Code	Event Name	Description
0x0	ARB_ERR_STARVATION ON	Client request starvation
0x1	ARB_ERR_ACK_TIME OUT	Grant ACK timeout
0x2	ARB_ERR_PROTOCOL_VI OLATION	ACK protocol violation
0x3	ARB_ERR_CREDIT_VI OLATION	Credit system violation
0x4	ARB_ERR_FAIRNESS _VIOLATION	Weighted fairness violation
0x5	ARB_ERR_WEIGHT_ UNDERFLOW	Weight credit underflow
0x6	ARB_ERR_CONCURR ENT_GRANTS	Multiple simultaneous grants
0x7	ARB_ERR_INVALID_ GRANT_ID	Invalid grant ID detected
0x8	ARB_ERR_ORPHAN_ ACK	ACK without pending grant
0x9	ARB_ERR_GRANT_O VERLAP	Overlapping grant periods
0xA	ARB_ERR_MASK_ER ROR	Round-robin mask error
0xB	ARB_ERR_STATE_MA CHINE	FSM state error
0xC	ARB_ERR_CONFIGUR ATION	Invalid configuration
0xD-0xE	Reserved	Future expansion
0xF	ARB_ERR_USER_DEF INED	User-defined error

Timeout Events (PktTypeTimeout + PROTOCOL_ARB)

Code	Event Name	Description
0x0	ARB_TIMEOUT_GRA NT_ACK	Grant ACK timeout
0x1	ARB_TIMEOUT_REQ UEST_HOLD	Request held too long
0x2	ARB_TIMEOUT_WEI GHT_UPDATE	Weight update timeout
0x3	ARB_TIMEOUT_BLO CK_RELEASE	Block release timeout
0x4	ARB_TIMEOUT_CRE DIT_UPDATE	Credit update timeout
0x5	ARB_TIMEOUT_STAT E_CHANGE	State machine timeout
0x6-0xE	Reserved	Future expansion
0xF	ARB_TIMEOUT_USER _DEFINED	User-defined timeout

Completion Events (PktTypeCompletion + PROTOCOL_ARB)

Code	Event Name	Description
0x0	ARB_COMPL_GRANT _ISSUED	Grant successfully issued
0x1	ARB_COMPL_ACK_R ECEIVED	ACK successfully received
0x2	ARB_COMPL_TRANS ACTION	Complete transaction (grant+ack)
0x3	ARB_COMPL_WEIGHT _UPDATE	Weight update completed
0x4	ARB_COMPL_CREDIT _CYCLE	Credit cycle completed
0x5	ARB_COMPL_FAIRNESS _PERIOD	Fairness analysis period
0x6	ARB_COMPL_BLOCK _PERIOD	Block period completed
0x7-0xE	Reserved	Future expansion

Code	Event Name	Description
0xF	ARB_COMPL_USER_DEFINED	User-defined completion

Threshold Events (PktTypeThreshold + PROTOCOL_ARB)

Code	Event Name	Description
0x0	ARB_THRESH_REQUEST_LATENCY	Request-to-grant latency threshold
0x1	ARB_THRESH_ACK_LATENCY	Grant-to-ACK latency threshold
0x2	ARB_THRESH_FAIRNESS_DEV	Fairness deviation threshold
0x3	ARB_THRESH_ACTIVE_REQUESTS	Active request count threshold
0x4	ARB_THRESH_GRANT_RATE	Grant rate threshold
0x5	ARB_THRESH EFFICIENCY	Grant efficiency threshold
0x6	ARB_THRESH_CREDIT_LOW	Low credit threshold
0x7	ARB_THRESH_WEIGHT_IMBALANCE	Weight imbalance threshold
0x8	ARB_THRESH_STARVATION_TIME	Starvation time threshold
0x9-0xE	Reserved	Future expansion
0xF	ARB_THRESH_USER_DEFINED	User-defined threshold

Performance Events (PktTypePerf + PROTOCOL_ARB)

Code	Event Name	Description
0x0	ARB_PERF_GRANT_ISSUED	Grant issued event
0x1	ARB_PERF_ACK RECEIVED	ACK received event
0x2	ARB_PERF_GRANT EFFICIENCY	Grant completion efficiency
0x3	ARB_PERF_FAIRNESS_METRIC	Fairness compliance metric
0x4	ARB_PERF_THROUGHPUT	Arbitration throughput
0x5	ARB_PERF_LATENCY_AVG	Average latency measurement

Code	Event Name	Description
0x6	ARB_PERF_WEIGHT_COMPLIANCE	Weight compliance metric
0x7	ARB_PERF_CREDIT_UTILIZATION	Credit utilization efficiency
0x8	ARB_PERF_CLIENT_ACTIVITY	Per-client activity metric
0x9	ARB_PERF_STARVATION_COUNT	Starvation event count
0xA	ARB_PERF_BLOCK EFFICIENCY	Block/unblock efficiency
0xB-0xE	Reserved	Future expansion
0xF	ARB_PERF_USER_DEFINED	User-defined performance

Debug Events (PktTypeDebug + PROTOCOL_ARB)

Code	Event Name	Description
0x0	ARB_DEBUG_STATE_CHANGE	Arbiter state machine change
0x1	ARB_DEBUG_MASK_UPDATE	Round-robin mask update
0x2	ARB_DEBUG_WEIGHT_CHANGE	Weight configuration change
0x3	ARB_DEBUG_CREDIT_UPDATE	Credit level update
0x4	ARB_DEBUG_CLIENT_MASK	Client enable/disable mask
0x5	ARB_DEBUG_PRIORITY_CHANGE	Priority level change
0x6	ARB_DEBUG_BLOCK_EVENT	Block/unblock event
0x7	ARB_DEBUG_QUEUE_STATUS	Request queue status
0x8	ARB_DEBUG_COUNT_ER_SNAPSHOT	Counter values snapshot
0x9	ARB_DEBUG_FIFO_S	FIFO status change

Code	Event Name	Description
	TATUS	
0xA	ARB_DEBUG_FAIRNESS_STATE	Fairness tracking state
0xB	ARB_DEBUG_ACK_STATE	ACK protocol state
0xC-0xE	Reserved	Future expansion
0xF	ARB_DEBUG_USER_DEFINED	User-defined debug

CORE Protocol Events

Error Events (PktTypeError + PROTOCOL_CORE)

Code	Event Name	Description
0x0	CORE_ERR_DESCRIPTOR_MALFORMED	Missing magic number (0x900dc0de)
0x1	CORE_ERR_DESCRIPTOR_BAD_ADDR	Invalid descriptor address
0x2	CORE_ERR_DATA_BAD_ADDR	Invalid data address (fetch or runtime)
0x3	CORE_ERR_FLAG_COMPARE_MISMATCH	Flag mask/compare mismatch
0x4	CORE_ERR_CREDIT_UNDERFLOW	Credit system violation
0x5	CORE_ERR_STATE_MACHINE	Invalid FSM state transition
0x6	CORE_ERR_DESCRIPTOR_ENGINE	Descriptor engine FSM error
0x7	CORE_ERR_FLAG_ENGINE	Flag engine FSM error
0x8	CORE_ERR_PROGRAM_ENGINE	Program engine FSM error
0x9	CORE_ERR_DATA_ENGINE	Data engine error
0xA	CORE_ERR_CHANNEL_INVALID_ID	Invalid channel ID
0xB	CORE_ERR_CONTROL_VIOLATION	Control register violation
0xC-0xE	Reserved	Future expansion

Code	Event Name	Description
------	------------	-------------

0xF	CORE_ERR_USER_DEFINED	User-defined error
------------	-----------------------	--------------------

Timeout Events (PktTypeTimeout + PROTOCOL_CORE)

Code	Event Name	Description
0x0	CORE_TIMEOUT_DESCRIPTOR_R_FETCH	Descriptor fetch timeout
0x1	CORE_TIMEOUT_FLAG_RETRY	Flag comparison retry timeout
0x2	CORE_TIMEOUT_PROGRAM_WRITE	Program write timeout
0x3	CORE_TIMEOUT_DATA_TRANSFER	Data transfer timeout
0x4	CORE_TIMEOUT_CREDIT_WAIT	Credit wait timeout
0x5	CORE_TIMEOUT_CONTROL_WAIT	Control enable wait timeout
0x6	CORE_TIMEOUT_ENGINE_RESPONSE	Sub-engine response timeout
0x7	CORE_TIMEOUT_STATE_TRANSITION	FSM state transition timeout
0x8-0xE	Reserved	Future expansion
0xF	CORE_TIMEOUT_USER_DEFINED	User-defined timeout

Completion Events (PktTypeCompletion + PROTOCOL_CORE)

Code	Event Name	Description
0x0	CORE_COMPL_DESCRIPTOR_LOADED	Descriptor successfully loaded
0x1	CORE_COMPL_DESCRIPTOR_CHAIN	Descriptor chain completed
0x2	CORE_COMPL_FLAG_MATCHED	Flag comparison successful
0x3	CORE_COMPL_PROGRAM_COMPLETED	Post-programming completed

Code	Event Name	Description
0x4	CORE_COMPL_DATA_TRANSFER	Data transfer completed
0x5	CORE_COMPL_CREDIT_CYCLE	Credit cycle completed
0x6	CORE_COMPL_CHANNEL_COMPLETE	Channel processing complete
0x7	CORE_COMPL_ENGINE_READY	Sub-engine ready
0x8-0xE	Reserved	Future expansion
0xF	CORE_COMPL_USER_DEFINED	User-defined completion

Threshold Events (PktTypeThreshold + PROTOCOL_CORE)

Code	Event Name	Description
0x0	CORE_THRESH_DESCRIPTOR_QUEUE	Descriptor queue depth threshold
0x1	CORE_THRESH_CREDIT_LOW	Credit low threshold
0x2	CORE_THRESH_FLAG_RETRY_COUNT	Flag retry count threshold
0x3	CORE_THRESH_LATENCY	Processing latency threshold
0x4	CORE_THRESH_ERROR_RATE	Error rate threshold
0x5	CORE_THRESH_THROUGHPUT	Throughput threshold
0x6	CORE_THRESH_ACTIVE_CHANNELS	Active channel count threshold
0x7	CORE_THRESH_PROGRAM_LATENCY	Program write latency threshold
0x8	CORE_THRESH_DATA_RATE	Data transfer rate threshold
0x9-0xE	Reserved	Future expansion
0xF	CORE_THRESH_USER_DEFINED	User-defined threshold

Performance Events (PktTypePerf + PROTOCOL_CORE)

Code	Event Name	Description
0x0	CORE_PERF_END_OF _DATA	Stream continuation signal
0x1	CORE_PERF_END_OF _STREAM	Stream termination signal
0x2	CORE_PERF_ENTERI NG_IDLE	FSM returning to idle
0x3	CORE_PERF_CREDIT_ INCREMENTED	Credit added by software
0x4	CORE_PERF_CREDIT_ EXHAUSTED	Credit blocking execution
0x5	CORE_PERF_STATE_ TRANSITION	FSM state change
0x6	CORE_PERF_DESCRI PTOR_ACTIVE	Data processing started
0x7	CORE_PERF_FLAG_R ETRY	Flag comparison retry
0x8	CORE_PERF_CHANN EL_ENABLE	Channel enabled by software
0x9	CORE_PERF_CHANN EL_DISABLE	Channel disabled by software
0xA	CORE_PERF_CREDIT_ UTILIZATION	Credit utilization metric
0xB	CORE_PERF_PROCES SING_LATENCY	Total processing latency
0xC	CORE_PERF_QUEUE_ DEPTH	Current queue depth
0xD-0xE	Reserved	Future expansion
0xF	CORE_PERF_USER_D EFINED	User-defined performance

Debug Events (PktTypeDebug + PROTOCOL_CORE)

Code	Event Name	Description
0x0	CORE_DEBUG_FSM_STATE_C	Descriptor FSM state change

Code	Event Name	Description
	HANGE	
0x1	CORE_DEBUG_DESCRIPTOR_CONTENT	Descriptor content trace
0x2	CORE_DEBUG_FLAG_ENGINE_STATE	Flag engine state trace
0x3	CORE_DEBUG_PROGRAM_ENGINE_STATE	Program engine state trace
0x4	CORE_DEBUG_CREDIT_OPERATION	Credit system operation
0x5	CORE_DEBUG_CONTROL_REGISTER	Control register access
0x6	CORE_DEBUG_ENGINE_HANDSHAKE	Engine handshake trace
0x7	CORE_DEBUG_QUEUE_STATUS	Queue status change
0x8	CORE_DEBUG_COUNTER_SNAPSHOT	Counter values snapshot
0x9	CORE_DEBUG_ADDRESS_TRACE	Address progression trace
0xA	CORE_DEBUG_PAYLOAD_TRACE	Payload content trace
0xB-0xE	Reserved	Future expansion
0xF	CORE_DEBUG_USER_DEFINED	User-defined debug

Memory Architecture and Packet Routing

Two-Tier Memory Architecture

Local Error/Interrupt Memory

Characteristic	Description
Storage Types	Error Packets (Type 0x0) and Timeout Packets (Type 0x3)
Access Method	Immediate CPU access without memory subsystem delays

Characteristic	Description
Capacity	Large enough to prevent overflow during error bursts
Priority	Critical events requiring immediate attention
Indexing	Fast search and retrieval mechanisms

Configurable External Memory

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

Routing Configuration

Base and Limit Registers

Register Set	Purpose	Configuration
Completion Config	Type 0x1 routing	base_addr, limit_addr, enable, priority
Threshold Config	Type 0x2 routing	base_addr, limit_addr, enable, priority
Performance Config	Type 0x4 routing	base_addr, limit_addr, enable, priority
Debug Config	Type 0xF routing	base_addr, limit_addr, enable, priority

Routing Decision Logic

Packet Type	Destination	Address Calculation
Error (0x0)	Local Memory	local_error_write_pointer

Packet Type	Destination	Address Calculation
Timeout (0x3)	Local Memory	local_error_write_pointer
Completion (0x1)	External Memory	completion_config.base_addr + offset
Performance (0x4)	External Memory	performance_config.base_addr + offset
Debug (0xF)	External Memory	debug_config.base_addr + offset

Address Space Management

Memory Layout Example

Address Range	Usage	Description
0x1000_0000 - 0x1000_FFFF	Local Error Memory	Immediate access storage
0x2000_0000 - 0x2001_FFFF	Performance Packets	External bulk storage
0x2010_0000 - 0x2011_FFFF	Completion Packets	External bulk storage
0x2020_0000 - 0x202F_FFFF	Debug Packets	External bulk storage

Transaction State and Bus Transaction Structure

Transaction State Enumeration

State	Value	Description	Usage
TRANS_EMPT_Y	3'b000	Unused entry	Available slot
TRANS_ADDR_PHASE	3'b001	Address phase active (AXI) / Packet sent (MNOC) / Setup phase (APB)	Initial phase
TRANS_DATA_PHASE	3'b010	Data phase active (AXI) / Waiting for ACK (MNOC) / Access phase (APB)	Data transfer
TRANS_RESP_PHASE	3'b011	Response phase active (AXI) / ACK received (MNOC) / Enable phase (APB)	Response handling
TRANS_COMPLETE	3'b100	Transaction complete	Successful completion

State	Value	Description	Usage
TRANS_ERRO	3'b101	Transaction has error	Error condition
R			
TRANS_ORPH	3'b110	Orphaned transaction	Missing components
ANED			
TRANS_CRED	3'b111	Credit stall (MNOC only)	MNOC-specific stall
IT_STALL			

Enhanced Transaction Structure

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

Phase Completion Flags

Flag	Description	Protocol Usage
cmd_receiv	Address phase received / ed	All protocols
	Packet sent / Setup phase	
data_starte	Data phase started / ACK	All protocols
d	expected / Access phase	

Flag	Description	Protocol Usage
data_comp_leted	Data phase completed / ACK received / Enable phase	All protocols
resp_received	Response received / Final ACK / PREADY asserted	All protocols

Protocol-Specific Tracking Fields

Field	Width	Description	Protocol
channel	6	Channel ID (AXI ID / MNOC channel / PSEL bit position)	All protocols
eos_seen	1	EOS marker seen	MNOC only
parity_error	1	Parity error detected	MNOC only
credit_at_start	8	Credits available at start	MNOC only
retry_count	3	Number of retries	MNOC only
desc_addr_match	1	Descriptor address match detected	AXI only
data_addr_match	1	Data address match detected	AXI only
apb_phase	2	Current APB phase	APB only
pslverr_seen	1	PSLVERR detected	APB only
pprot_value	3	PPROT value	APB only
pstrb_value	4	PSTRB value for writes	APB only
arb_grant_id	8	Current grant ID	ARB only
arb_weight	8	Current weight value	ARB only
core_fsm_state	3	Current CORE FSM state	CORE only
core_channel_id	6	CORE channel identifier	CORE only

APB Transaction Phases

Phase	Value	Description
APB_PHASE_IDLE	2'b00	Bus idle
APB_PHASE_SETUP	2'b01	Setup phase (PSEL)

Phase	Value	Description
		asserted)
APB_PHASE_ACCE S	2'b10	Access phase (PENABLE asserted)
APB_PHASE_ENABL E	2'b11	Enable phase (waiting for PREADY)

APB Protection Types

Protection	Value	Description
APB_PROT_NORMA L	3'b000	Normal access
APB_PROT_PRIVILE GED	3'b001	Privileged access
APB_PROT_SECURE	3'b010	Secure access
APB_PROT_INSTRU CTION	3'b100	Instruction access

MNOC Payload Types

Payload	Value	Description
MNOC_PAYLOAD_C ONFIG	2'b00	CONFIG_PKT
MNOC_PAYLOAD_T S	2'b01	TS_PKT
MNOC_PAYLOAD_R DA	2'b10	RDA_PKT
MNOC_PAYLOAD_R AW	2'b11	RAW_PKT

MNOC ACK Types

ACK Type	Value	Description
MNOC_ACK_STOP	2'b00	MSAP_STOP
MNOC_ACK_START	2'b01	MSAP_START
MNOC_ACK_CREDI T_ON	2'b10	MSAP_CREDIT_ON
MNOC_ACK_STOP_	2'b11	MSAP_STOP_AT_EOS

ACK Type	Value	Description
AT_EOS		

ARB State Types

State	Value	Description
ARB_STATE_IDLE	3'b000	Idle state
ARB_STATE_ARBITRATE	3'b001	Performing arbitration
ARB_STATE_GRANT	3'b010	Grant issued, waiting for ACK
ARB_STATE_BLOCKED	3'b011	Arbitration blocked
ARB_STATE_WEIGHT_UPD	3'b100	Weight update in progress
ARB_STATE_ERROR	3'b101	Error state

CORE State Types

State	Value	Description
CORE_STATE_IDLE	3'b000	Idle state
CORE_STATE_DESC_FETCH	3'b001	Fetching descriptor
CORE_STATE_FLAG_CHECK	3'b010	Checking flag condition
CORE_STATE_PROG_RAM_WRITE	3'b011	Writing program RAM_WRITE
CORE_STATE_DATA_TRANSFER	3'b100	Transferring data
CORE_STATE_CRED_WAIT	3'b101	Waiting for credits
CORE_STATE_ERROR	3'b110	Error state

Configuration and Control

Monitor Configuration Registers

Global Configuration

Field	Width	Description
monitor_enable	1	Global monitor enable
error_local_enable	1	Enable local error storage
external_route_ena ble	1	Enable external routing
unit_id	4	Unit identifier
agent_id	8	Agent identifier
packet_type_enable s	16	Per-type enable bits

Packet Type Enable Mapping

Bit	Enable	Description
0	PKT_ENABLE_ERRO R	Enable error packets
1	PKT_ENABLE_COMP LETION	Enable completion packets
2	PKT_ENABLE_THRES HOLD	Enable threshold packets
3	PKT_ENABLE_TIME OUT	Enable timeout packets
4	PKT_ENABLE_PERF	Enable performance packets
5	PKT_ENABLE_CREDI T	Enable credit packets (MNOC)
6	PKT_ENABLE_CHAN NEL	Enable channel packets (MNOC)
7	PKT_ENABLE_STREA M	Enable stream packets (MNOC)
8	PKT_ENABLE_ADDR_	Enable address

Bit	Enable	Description
	MATCH	match (AXI)
9	PKT_ENABLE_APB	Enable APB packets
15	PKT_ENABLE_DEBU G	Enable debug packets

Protocol-Specific Configuration

AXI Monitor Configuration

Field	Width	Description
active_trans_thresh	16	Active transaction threshold
old		
latency_threshold	32	Latency threshold (cycles)
addr_timeout_cnt	4	Address timeout count
data_timeout_cnt	4	Data timeout count
resp_timeout_cnt	4	Response timeout count
burst_boundary_ch	1	Enable burst boundary checking
eck		
address_match_ena	1	Enable address matching
ble		
desc_addr_match_b	64	Descriptor address match base
ase		
desc_addr_match_	64	Descriptor address match mask
mask		
data_addr_match_b	64	Data address match base
ase		
data_addr_match_	64	Data address match mask
mask		

MNOC Monitor Configuration

Field	Width	Description
credit_low_threshold	8	Credit low threshold
d		

Field	Width	Description
packet_rate_thresh	16	Packet rate threshold
old		
max_route_hops	8	Maximum routing hops
enable_credit_tracking	1	Enable credit tracking
enable_deadlock_detection	1	Enable deadlock detection
deadlock_timeout	4	Deadlock detection timeout

ARB Monitor Configuration

Field	Width	Description
grant_timeout_cnt	16	Grant ACK timeout count
fairness_window	32	Fairness analysis window
weight_update_enable	1	Enable weight tracking
starvation_threshold	16	Starvation detection threshold
efficiency_threshold	8	Grant efficiency threshold

CORE Monitor Configuration

Field	Width	Description
descriptor_timeout_cnt	16	Descriptor fetch timeout count
flag_retry_limit	8	Maximum flag retry count
credit_low_threshold	8	Credit low threshold
processing_timeout_cnt	32	Processing timeout count
enable_descriptor_t	1	Enable descriptor

Field	Width	Description
race		content tracing
enable_fsm_trace	1	Enable FSM state tracing

Validation Requirements

Functional Validation

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

Performance Validation

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

Error Handling Validation

Validation Area	Requirements
Error Injection	Verify error detection and reporting
Overflow Handling	Verify behavior when memories

Validation Area	Requirements
Configuration Errors	fill Verify invalid configuration detection
Recovery Mechanisms	Verify error recovery procedures

Usage Examples

Creating Monitor Packets

Packet Type	Example Usage
AXI Error	Protocol=AXI, Type=Error, Code=AXI_ERR_RESP_SLVERR
MNOC Credit	Protocol=MNOC, Type=Credit, Code=MNOC_CREDIT_EXHAUSTED
APB Performance	Protocol=APB, Type=Performance, Code=APB_PERF_TOTAL_LATENCY
ARB Threshold	Protocol=ARB, Type=Threshold, Code=ARB_THRESH_FAIRNESS_DEV
CORE Completion	Protocol=CORE, Type=Completion, Code=CORE_COMPL_DESCRIPTOR_LOADED

Packet Decoding

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

Monitor Bus Packet Helper Functions

Packet Field Extraction

Function	Return Type	Description
get_packet_type(pk t)	logic [3:0]	Extract packet type [63:60]

Function	Return Type	Description
get_protocol_type(pkt)	protocol_type_t	Extract protocol [59:57]
get_event_code(pkt)	logic [3:0]	Extract event code [56:53]
get_channel_id(pkt)	logic [5:0]	Extract channel ID [52:47]
get_unit_id(pkt)	logic [3:0]	Extract unit ID [46:43]
get_agent_id(pkt)	logic [7:0]	Extract agent ID [42:35]
get_event_data(pkt)	logic [34:0]	Extract event data [34:0]

Packet Creation Function

Function	Parameters	Description
create_monitor_packet()	packet_type, protocol, event_code, channel_id, unit_id, agent_id, event_data	Create complete 64-bit packet

Event Code Creation Functions

Function	Parameter	Description
create_axi_error_event()	axi_error_code_t	Create AXI error event code
create_axi_timeout_event()	axi_timeout_code_t	Create AXI timeout event code
create_axi_completion_event()	axi_completion_code_t	Create AXI completion event code
create_axi_threshold_event()	axi_threshold_code_t	Create AXI threshold event code
create_axi_performance_event()	axi_performance_code_t	Create AXI performance event code
create_axi_addr_match_event()	axi_addr_match_code_t	Create AXI address match event code
create_axi_debug_event()	axi_debug_code_t	Create AXI debug event code
create_apb_error_event()	apb_error_code_t	Create APB error event code

Function	Parameter	Description
ent()		
create_apb_timeout_event()	apb_timeout_code_t	Create APB timeout event code
create_apb_completion_event()	apb_completion_code_t	Create APB completion event code
create_mnoc_error_event()	mnoc_error_code_t	Create MNOC error event code
create_mnoc_timeout_event()	mnoc_timeout_code_t	Create MNOC timeout event code
create_mnoc_completion_event()	mnoc_completion_code_t	Create MNOC completion event code
create_mnoc_credit_event()	mnoc_credit_code_t	Create MNOC credit event code
create_mnoc_channel_event()	mnoc_channel_code_t	Create MNOC channel event code
create_mnoc_stream_event()	mnoc_stream_code_t	Create MNOC stream event code
create_arb_error_event()	arb_error_code_t	Create ARB error event code
create_arb_timeout_event()	arb_timeout_code_t	Create ARB timeout event code
create_arb_completion_event()	arb_completion_code_t	Create ARB completion event code
create_arb_threshold_event()	arb_threshold_code_t	Create ARB threshold event code
create_arb_performance_event()	arb_performance_code_t	Create ARB performance event code
create_arb_debug_event()	arb_debug_code_t	Create ARB debug event code
create_core_error_event()	core_error_code_t	Create CORE error event code
create_core_timeout_event()	core_timeout_code_t	Create CORE timeout event code
create_core_completion_event()	core_completion_code_t	Create CORE completion

Function	Parameter	Description
ion_event()		event code
create_core_threshold_event()	core_threshold_code_t	Create CORE threshold event code
create_core_performance_event()	core_performance_code_t	Create CORE performance event code
create_core_debug_event()	core_debug_code_t	Create CORE debug event code

Validation Functions

Function	Parameters	Description
is_valid_event_for_packet_type()	packet_type, protocol, event_code	Validate event code for packet type and protocol

String Functions for Debugging

Function	Parameter	Description
get_axi_error_name()	axi_error_code_t	Get human-readable AXI error name
get_arb_error_name()	arb_error_code_t	Get human-readable ARB error name
get_core_error_name()	core_error_code_t	Get human-readable CORE error name
get_packet_type_name()	logic [3:0]	Get packet type name string
get_protocol_name()	protocol_type_t	Get protocol name string
get_event_name()	packet_type, protocol, event_code	Get comprehensive event name

Debug and Monitoring Signals

Essential Debug Signals

Signal	Width	Purpose
debug_packet_counts	32×16	Packet count per type
debug_protocol_counts	32×5	Packet count per protocol
debug_error_count	32	Total error packet

Signal	Width	Purpose
s		count
debug_local_memory_level	16	Local memory usage level
debug_external_memory_level	16	External memory usage level

Performance Counters

Counter	Width	Purpose
total_packets_processed	32	Total packets processed
packets_dropped	32	Packets dropped due to overflow
routing_errors	32	Routing configuration errors
memory_full_events	32	Memory full occurrences

Protocol Coverage Summary

Complete Protocol Event Matrix

Protocol	Error	Timeout	Completion	Threshold	Performance	Debug	Protocol-Specific
AXI	YES 16	YES 16	YES 16	YES 16	YES 16	YES 16	AddrMatch YES 16
MNOC	YES 16	YES 16	YES 16	NO 0	NO 0	NO 0	Credit/ Channel/ Stream YES 48
APB	YES 16	YES 16	YES 16	YES 16	YES 16	YES 16	None
ARB	YES 16	YES 16	YES 16	YES 16	YES 16	YES 16	None
CORE	YES 16	YES 16	YES 16	YES 16	YES 16	YES 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)

0x000 - 0x03F: Channel Kick-off (apbtodescr.sv routing)
0x000: CH0_CTRL_LOW - Channel 0 descriptor address [31:0]
0x004: CH0_CTRL_HIGH - Channel 0 descriptor address [63:32]
0x008: CH1_CTRL_LOW - Channel 1 descriptor address [31:0]
0x00C: CH1_CTRL_HIGH - Channel 1 descriptor address [63:32]
0x010: CH2_CTRL_LOW - Channel 2 descriptor address [31:0]
0x014: CH2_CTRL_HIGH - Channel 2 descriptor address [63:32]
0x018: CH3_CTRL_LOW - Channel 3 descriptor address [31:0]
0x01C: CH3_CTRL_HIGH - Channel 3 descriptor address [63:32]
0x020: CH4_CTRL_LOW - Channel 4 descriptor address [31:0]
0x024: CH4_CTRL_HIGH - Channel 4 descriptor address [63:32]
0x028: CH5_CTRL_LOW - Channel 5 descriptor address [31:0]
0x02C: CH5_CTRL_HIGH - Channel 5 descriptor address [63:32]
0x030: CH6_CTRL_LOW - Channel 6 descriptor address [31:0]
0x034: CH6_CTRL_HIGH - Channel 6 descriptor address [63:32]
0x038: CH7_CTRL_LOW - Channel 7 descriptor address [31:0]
0x03C: CH7_CTRL_HIGH - Channel 7 descriptor address [63:32]
0x040 - 0x0FF: Reserved
0x100 - 0xFFFF: Configuration and Status Registers
0x100 - 0x11F: Global Control and Status
0x120 - 0x17F: Per-Channel Control and Status
0x200 - 0x21F: Scheduler Configuration
0x220 - 0x23F: Descriptor Engine Configuration
0x240 - 0x27F: Monitor 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			
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]
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]

Offset		Type	Reset	Description
t	Register	e	t	
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	RW	0	Global reset (self-clearing) T
0	GLOBAL_E	RW	0	Global enable (1=enabled, 0=disabled) N

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	Typ			Description
		e	Reset		
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	Typ		Description
		e		
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	Typ		Description
		e		
31:8	Reserved	RO		Reserved

Bits	Field	Type	Description
7:0	SCHED_IDL_E	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_E	31:7	Reserved	RO	Reserved
		6:0	STATE	RO	Channel 0 scheduler state (one-hot)
0x154	CH1_STAT_E	6:0	STATE	RO	Channel 1 scheduler state (one-hot)
0x158	CH2_STAT_E	6:0	STATE	RO	Channel 2 scheduler state (one-hot)
0x15C	CH3_STAT_E	6:0	STATE	RO	Channel 3 scheduler state (one-hot)
0x160	CH4_STAT_E	6:0	STATE	RO	Channel 4 scheduler state (one-hot)
0x164	CH5_STAT_E	6:0	STATE	RO	Channel 5 scheduler state (one-hot)
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.

Scheduler Configuration (0x200 - 0x21F)

SCHED_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_ES	RW	1000	Timeout in clock cycles

SCHED_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_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

Bits	Field	Type	Reset	Description
5:2	FIFO_THRES_H	RW	0x8	Prefetch FIFO threshold (4 bits)
1	PREFETCH_E_N	RW	0	Prefetch enable
0	DESCENG_EN	RW	1	Descriptor engine enable

DESCENG_ADDR0_BASE (0x224)

Base address for descriptor address range 0 (lower 32 bits).

Bits	Field	Type	Reset	Description
31:0	ADDR0_BA_SE	RW	0x00000000 0	Address range 0 base

DESCENG_ADDR0_LIMIT (0x228)

Limit address for descriptor address range 0 (lower 32 bits).

Bits	Field	Type	Reset	Description
31:0	ADDR0_LIM_IT	RW	0xFFFFFFF FF	Address range 0 limit

DESCENG_ADDR1_BASE (0x22C)

Base address for descriptor address range 1 (lower 32 bits).

Bits	Field	Type	Reset	Description
31:0	ADDR1_BA_SE	RW	0x00000000 0	Address range 1 base

DESCENG_ADDR1_LIMIT (0x230)

Limit address for descriptor address range 1 (lower 32 bits).

Bits	Field	Type	Reset	Description
31:0	ADDR1_LIM	RW	0xFFFFFFF	Address range 1 limit

Bits	Field	Type	Reset	Description
	IT		FF	

Monitor Configuration (0x240 - 0x27F)

DAXMON_CONFIG (0x240)

Descriptor AXI master monitor configuration.

Bits	Field	Type	Reset	Description
31:5	Reserved	RO	0x0	Reserved
4	DEBUG_EN	RW	0	Debug packet enable
3	PERF_EN	RW	0	Performance packet enable
2	TIMEOUT_E	RW	1	Timeout detection enable N
1	COMPL_EN	RW	0	Completion packet enable
0	ERR_EN	RW	1	Error detection enable

RDMON_CONFIG (0x244)

Data read AXI master monitor configuration.

Bits	Field	Type	Reset	Description
31:5	Reserved	RO	0x0	Reserved
4	DEBUG_EN	RW	0	Debug packet enable
3	PERF_EN	RW	0	Performance packet enable
2	TIMEOUT_E	RW	1	Timeout detection enable N
1	COMPL_EN	RW	0	Completion packet enable
0	ERR_EN	RW	1	Error detection enable

RDMON_TIMEOUT (0x248)

Data read monitor timeout threshold.

Bits	Field	Typ e	Reset	Description
31:16	Reserved	RO	0x0	Reserved
15:0	TIMEOUT_CYCL ES	RW	5000	Timeout in clock cycles

[WRMON_CONFIG \(0x24C\)](#)

Data write AXI master monitor configuration.

Bits	Field	Typ e	Reset	Description
31:5	Reserved	RO	0x0	Reserved
4	DEBUG_EN	RW	0	Debug packet enable
3	PERF_EN	RW	0	Performance packet enable
2	TIMEOUT_E N	RW	1	Timeout detection enable
1	COMPL_EN	RW	0	Completion packet enable
0	ERR_EN	RW	1	Error detection enable

[WRMON_TIMEOUT \(0x250\)](#)

Data write monitor timeout threshold.

Bits	Field	Typ e	Reset	Description
31:16	Reserved	RO	0x0	Reserved
15:0	TIMEOUT_CYCL ES	RW	5000	Timeout in clock cycles

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
0x120-0x17F	Per-channel control and status	12	RW/RO
0x200-0x21F	Scheduler configuration	2	RW
0x220-0x23F	Descriptor engine configuration	6	RW
0x240-0x27F	Monitor configuration	5	RW

Total: 44 registers (16 kick-off + 28 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

Chapter 5: Programming Models

This chapter provides software developer guidance for using the STREAM DMA engine.

Planned Contents

01_initialization.md

- Power-on initialization sequence
- Register configuration

- Channel setup
- Interrupt configuration

02_single_transfer.md

- Simple single-descriptor transfer
- Example: Memory copy operation
- C code examples
- Expected timing

03_chained_transfers.md

- Multi-descriptor chains
- Descriptor linking
- Chain termination
- Error handling in chains

04_multi_channel.md

- Concurrent channel operations
- Priority management
- Resource sharing
- Performance optimization

05_error_handling.md

- Error detection
- Error reporting via registers
- Recovery procedures
- Timeout handling

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: TBD - To be created during detailed documentation phase

Target Audience: - Software engineers integrating STREAM - Driver developers - System architects - Application developers

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
Channel Control	2	Enable/reset individual channels
Scheduler	6	Transfer scheduling and timeouts
Descriptor Engine	7	Descriptor fetch behavior
AXI Monitors	45 (15 per monitor × 3)	Debug/trace filtering
AXI Transfer	2	Burst configuration
Performance	3	Profiling and metrics
Total	65 configuration signals	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_enable

Type: Global scheduler enable **Width:** 1 bit **Default:** 1'b1 (enabled) **Register:** SCHED_CTRL @ 0x200[0]

Description: Master enable for all schedulers. When disabled, all channels stop scheduling new operations.

cfg_sched_timeout_cycles[15:0]

Type: Timeout threshold **Width:** 16 bits **Default:** 16'd10000 (10,000 cycles)

Register: SCHED_TIMEOUT @ 0x204[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:

Timeout cycles = (Expected latency × 10) + margin

Example for DDR4 @ 200 MHz:

- Expected read latency: 100 cycles (500 ns)
 - Safety margin: 10x
 - Timeout = $100 \times 10 = 1000$ cycles
-

cfg_sched_timeout_enable

Type: Timeout detection enable **Width:** 1 bit **Default:** 1'b1 (enabled) **Register:** SCHED_CTRL @ 0x200[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_CTRL @ 0x200[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'b0 (disabled by default) **Register:** SCHED_CTRL @ 0x200[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_CTRL @ 0x200[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[ADDR_WIDTH-1:0]

Type: Base address for descriptor region 0 **Width:** 64 bits (parameterizable)
Default: 64'h0000_0000_0000_0000 **Register:** DESCENG_ADDR0_BASE @ 0x224-0x228

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[ADDR_WIDTH-1:0]

Type: Limit address for descriptor region 0 **Width:** 64 bits **Default:** 64'hFFFF_FFFF_FFFF_FFFF (no limit) **Register:** DESCENG_ADDR0_LIMIT @ 0x22C-0x230

Description: Upper limit of first descriptor region. Descriptors beyond this address use region 1.

cfg_desceng_addr1_base[ADDR_WIDTH-1:0]

Type: Base address for descriptor region 1 **Width:** 64 bits **Default:** 64'h0000_0000_0000_0000 **Register:** DESCENG_ADDR1_BASE @ 0x234-0x238

Description: Base address of second descriptor memory region (optional).

cfg_desceng_addr1_limit[ADDR_WIDTH-1:0]

Type: Limit address for descriptor region 1 **Width:** 64 bits **Default:** 64'hFFFF_FFFF_FFFF **Register:** DESCENG_ADDR1_LIMIT @ 0x23C-0x240

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:

$$\text{Transfer size (bytes)} = \text{beats} \times (\text{DATA_WIDTH} / 8)$$

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:

- Start with errors only:** Enable only `cfg_*_mon_err_enable` initially. Add other packets as needed.
 - Completion packets are expensive:** Only enable `cfg_*_mon_compl_enable` for specific channels during debug.
 - Performance packets flood MonBus:** Enable `cfg_*_mon_perf_enable` sparingly (1-2 channels maximum).
 - 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, global_RST	-
0x120	CHANNEL_ENABLE	ch_en[7:0]	1
0x124	CHANNEL_RESET	ch_RST[7:0]	1
0x200	SCHED_CTRL	enable, timeout_en, err_en, compl_en, perf_en	2
0x204	SCHED_TIMEOUT	timeout_cycle s[15:0]	2
0x220	DESCENG_CTRL	enable, prefetch, fifo_thresh	3
0x224	DESCENG_ADDR0_BASE	addr0_base[6 3:0]	3
0x22C	DESCENG_ADDR0_LIMIT	addr0_limit[6 3:0]	3
0x234	DESCENG_ADDR1_BASE	addr1_base[6 3:0]	3
0x23C	DESCENG_ADDR1_LIMIT	addr1_limit[6 3:0]	3
0x240-0x25F	DAXMON_*	Descriptor monitor (15 signals)	4.1
0x260-0x27F	RDMON_*	Read engine monitor (15 signals)	4.2
0x280-0x29F	WRMON_*	Write engine monitor (15	4.3

Address	Register Name	Fields	Section
0x2A0	AXI_XFER_CFG	signals) rd_xfer_beats ,	5
0x2B0	PERF_CTRL	wr_xfer_beat s perf_enable, perf_mode, 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] = 0x00000007; // enable | timeout_en | err_en
    base_addr[0x204/4] = 500;        // timeout_cycles (SRAM)

    // 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] = 0x00000007; // enable | timeout_en | err_en
    base_addr[0x204/4] = 5000;       // timeout_cycles (DDR4)

    // 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: 2025-11-21 **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
Network Interfaces	Yes (Network master/slave)	✗ No (pure memory-to-memory)
Address Alignment	Complex fixup logic	✓ Aligned addresses only
Credit Management	Exponential encoding	✓ Simple transaction limits
Control Engines	Control read/write engines	✗ No (direct APB config)
Descriptor Length	Chunks (4-byte)	✓ Beats (data width)
Program Engine	Complex alignment FSM	✓ Simplified coordination

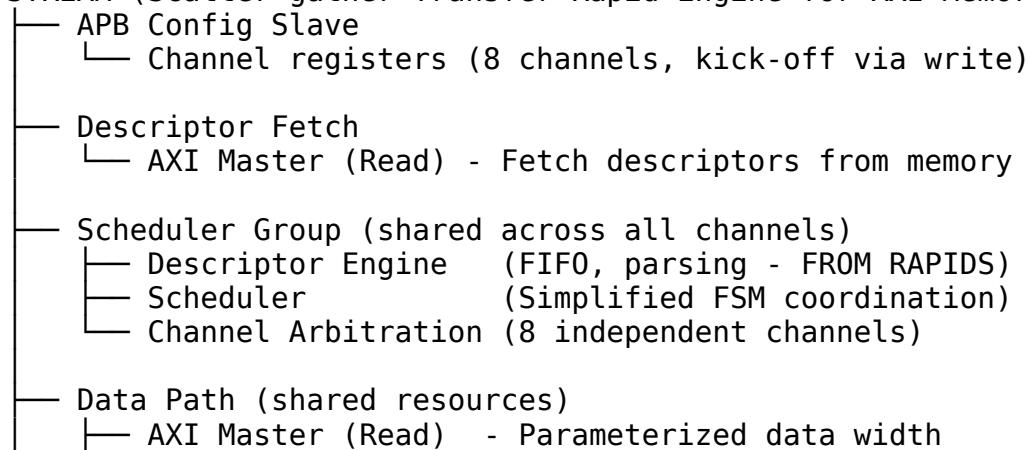
2.2 Tutorial-Friendly Features

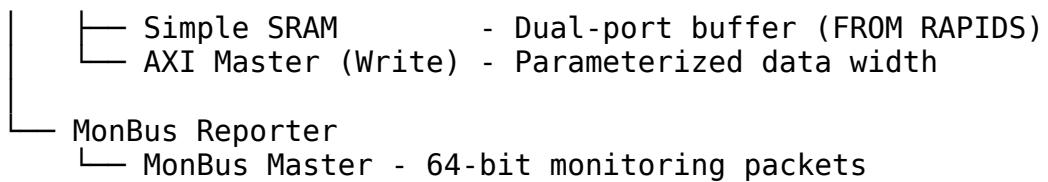
- **Aligned Addresses:** Source and destination addresses must be aligned to data width
 - **Length in Beats:** Descriptor length specified in data beats (not bytes/chunks)
 - **Single APB Write:** One APB register write kicks off entire descriptor chain
 - **No Circular Buffers:** Chained descriptors with explicit termination
 - **Parameterized Engines:** Multiple AXI engine versions (compile-time selection)
-

3. Architecture Overview

3.1 Top-Level Block Diagram

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





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
APB Slave	Slave	32-bit	Configuration, channel kick-off	Write to channel register starts transfer
AXI Master (Descriptor)	Master	256-bit	Fetch descriptors from memory	Dedicated descriptor fetch path
AXI Master (Data)	Master	Parameteriz	Read source	Multiple

Interface	Type	Width	Purpose	Notes
Read)		able	data	engine versions (compile-time)
AXI Master (Data Write)	Master	Parameterizable	Write destination data	Multiple engine versions (compile-time)
MonBus Master	Master	64-bit	Monitor packet output	Standard AMBA format

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 BEATS (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	Gates APB interface

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
 datard_addr;           // Source address (aligned)
 datard_beats_remaining; // Beats left to read
 datard_burst_len;        // Preferred burst length
 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
 datawr_valid;           // Channel has write
request
 datawr_ready;          // Engine ready for
request
 [63:0] datawr_addr;      // Destination address
(aligned)
 [31:0] datawr_beats_remaining; // Beats left to write
 [7:0] datawr_burst_len;       // Preferred burst length
 [3:0] datawr_channel_id;     // 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	Channel 0 kick-off (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
Invalid descriptor	Valid bit = 0	Skip, move to next
Alignment error	Address not aligned	Set error flag, halt channel
AXI SLVERR	AXI response	Set error flag, halt channel

Error Type	Detection	Response
AXI DECERR	AXI response	Set error flag, halt channel
Timeout	Transaction timeout	Set error flag, halt channel
SRAM overflow	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

Code	Event	Description
		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 complexity
Length in beats	Avoid byte/chunk conversion math
No circular buffers	Explicit termination easier to understand
Single APB kick-off	Simple software model
No credit management	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 (8×): ~2,400 LUTs - Scheduler (8×): ~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`
- **RAPIDS Simple SRAM:**
`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
    area!)
      ├── 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(TBBase):
    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)

- APB Config
 - └ apb_config_slave.sv (To be created)
- Scheduler Group
 - └ descriptor_engine.sv (FROM RAPIDS - adapt imports)
 - └ scheduler.sv (Simplified from RAPIDS)
 - └ channel_arbiter.sv (To be created)
- Data Path
 - └ axi_read_engine_v1.sv (Version 1: Basic)
 - └ axi_read_engine_v2.sv (Version 2: Pipelined)
 - └ axi_write_engine_v1.sv (Version 1: Basic)
 - └ axi_write_engine_v2.sv (Version 2: Pipelined)
 - └ simple_sram.sv (FROM RAPIDS - no changes)
- MonBus Reporter
 - └ monbus_reporter.sv (To be created)

Module Status

Module	Source	Status	Notes
descriptor_engin e.sv	RAPIDS (copy)	✓ Copied	Adapt #include only

Module	Source	Status	Notes
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
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_app(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 × 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 x 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 x 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, reset
0x04	GLOBAL_STAT	RO US	Global status, channel idle/error
0x08	GLOBAL_CONF	RW IG	Global configuration
0x0C	Reserved	-	-

Channel Registers ($8 \times 0x10$ 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 address (write to kick off)
+0x04	CHx_STATUS	RO	Channel status, idle, error
+0x08	CHx_RD_BURS T	RW	Read burst length (for engine config)
+0x0C	CHx_WR_BURS T	RW	Write burst length (for 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/apb_config.sv
module apb_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_hpet/peakrdl/hpet_regs.rdl](#) - Register definition -
[projects/components/apb_hpet/peakrdl/generated/](#) - Generated outputs -
[projects/components/apb_hpet/rtl/apb_hpet.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