STREAM Specification Index

Generated by md\_to\_docx.py

today

Abstract

Table of Contents

# STREAM Specification Index

**Version:** 0.26 **Date:** 2025-10-17 **Purpose:** Complete technical specification for STREAM subsystem

## Document Organization

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

### Chapter 1: Overview

## Clocks and Reset Specification

**Chapter:** 01 **Version:** 1.0 **Last Updated:** 2025-10-17

### 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) - Dual-flop synchronizers for control signals - Async FIFO for data paths (if needed)

### 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:**

aclk \_\_| |\_| |\_| |\_| |\_| |\_| |\_| |\_| |\_| |\_| |\_| |\_  
pclk \_\_| |\_| |\_| |\_| |\_| |\_| |\_| |\_| |\_| |\_| |\_| |\_  
aresetn \_\_\_\_\_\_\_\_\_\_|   
presetn \_\_\_\_\_\_\_\_\_\_|   
 <--10 cyc-->

#### 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\_axil\_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 for control signals:** - Use dual-flop synchronizers (2-3 stages) - Add ASYNC\_REG attribute for timing tools - Ensure proper constraints in SDC/XDC

**CDC for data paths:** - Use async FIFOs with gray code pointers - Ensure proper full/empty flag synchronization

#### 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

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

### Chapter 2: Functional Blocks

## Descriptor Engine Specification

**Module:** descriptor\_engine.sv **Location:** rtl/stream\_fub/ **Source:** Simplified from RAPIDS

### Overview

The Descriptor Engine fetches descriptors from system memory via AXI and parses them into structured fields for the Scheduler. This module is **simplified from RAPIDS** with a smaller descriptor format (256-bit vs 512-bit).

#### 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 ADDR\_WIDTH = 64; // Address bus width  
parameter int DATA\_WIDTH = 256; // Descriptor width  
parameter int AXI\_ID\_WIDTH = 8; // AXI ID width  
parameter int FIFO\_DEPTH = 16; // Descriptor FIFO depth

#### Ports

**Clock and Reset:**

input logic aclk;  
input logic aresetn;

**Descriptor Request (from Scheduler):**

input logic desc\_req\_valid;  
output logic desc\_req\_ready;  
input logic [ADDR\_WIDTH-1:0] desc\_req\_addr;  
input logic [3:0] desc\_req\_channel\_id; // Channel ID for AXI ID

**Descriptor Output (to Scheduler):**

output logic desc\_valid;  
input logic desc\_ready;  
output descriptor\_t desc\_packet;

**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;  
input logic monbus\_ready;  
output logic [63:0] monbus\_packet;

### Descriptor Format

See 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

#### 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

### Differences from RAPIDS

**Descriptor Size:** - **RAPIDS:** 512-bit descriptor (8 x 64-bit words) - **STREAM:** 256-bit descriptor (4 x 64-bit words) - **Half the size!**

**Key Simplifications:** 1. **Smaller descriptor:** 256 bits vs 512 bits 2. **Simpler fields:** No alignment metadata, no chunk information 3. **Length in beats:** Not chunks (4-byte units) 4. **No circular buffers:** Explicit chain termination only

**Import Change:**

// RAPIDS:  
`include "rapids\_imports.svh"  
  
// STREAM:  
`include "stream\_imports.svh"

**Why Half Size:** - RAPIDS handles complex alignment (requires extra metadata) - RAPIDS uses chunk-based length (4-byte units) - STREAM requires aligned addresses (no fixup metadata needed) - STREAM uses beat-based length (simpler, no conversion needed)

### 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

**Reference:** RAPIDS descriptor\_engine tests can be reused with minimal adaptation.

### Related Documentation

* **RAPIDS Spec:** projects/components/rapids/docs/rapids\_spec/ch02\_blocks/01\_02\_descriptor\_engine.md
* **Package:** rtl/includes/stream\_pkg.sv - Descriptor format
* **Source:** rtl/stream\_fub/descriptor\_engine.sv

**Last Updated:** 2025-10-17 ## Scheduler Specification

**Module:** scheduler.sv **Location:** rtl/stream\_fub/ **Based On:** RAPIDS scheduler (simplified)

### Overview

The Scheduler coordinates descriptor-to-data-transfer flow for a single STREAM channel. It tracks total beats remaining and requests access to data engines via simplified interfaces.

#### Key Simplifications from RAPIDS

* [No] No credit management (removed exponential encoding)
* [No] No control read/write engines (direct APB only)
* [No] No network interfaces (pure memory-to-memory)
* [Done] Simpler FSM (6 states vs RAPIDS 12+ states)
* [Done] Aligned addresses only (no fixup logic)
* [Done] Length in beats (not chunks)

#### Critical Architecture

**[WARNING] RIGID SEPARATION OF CONCERNS:**

* **Scheduler (Coordinator):** Tracks total work, requests access to engines
* **Engines (Autonomous Workers):** Decide burst lengths, execute transactions, report completion

**Interface Contract:** - Scheduler provides: beats\_remaining (total work) - Engines report back: beats\_done (work completed) - Scheduler does **NOT** specify burst lengths

### Interface

#### Parameters

parameter int CHANNEL\_ID = 0; // Channel ID (0-7)  
parameter int ADDR\_WIDTH = 64; // Address bus width  
parameter int DATA\_WIDTH = 512; // Data bus width

#### Ports

**Clock and Reset:**

input logic aclk;  
input logic aresetn;

**Configuration:**

input logic cfg\_enable; // Channel enable  
input logic [31:0] cfg\_timeout; // Timeout threshold

**Descriptor Input (from Descriptor Engine):**

input logic desc\_valid;  
output logic desc\_ready;  
input descriptor\_t desc\_packet;

**Descriptor Request (to Descriptor Engine):**

output logic desc\_req\_valid;  
input logic desc\_req\_ready;  
output logic [ADDR\_WIDTH-1:0] desc\_req\_addr;

**Data Read Interface (to AXI Read Engine via Arbiter):**

output logic datard\_valid; // Request read access  
input logic datard\_ready; // Engine grants access  
output logic [63:0] datard\_addr; // Source address  
output logic [31:0] datard\_beats\_remaining; // Total beats to read  
output logic [3:0] datard\_channel\_id; // Channel ID  
  
// Completion feedback  
input logic datard\_done\_strobe; // Read completed  
input logic [31:0] datard\_beats\_done; // Beats actually moved  
input logic datard\_error; // Read error

**Data Write Interface (to AXI Write Engine via Arbiter):**

output logic datawr\_valid; // Request write access  
input logic datawr\_ready; // Engine grants access  
output logic [63:0] datawr\_addr; // Destination address  
output logic [31:0] datawr\_beats\_remaining; // Total beats to write  
output logic [3:0] datawr\_channel\_id; // Channel ID  
  
// Completion feedback  
input logic datawr\_done\_strobe; // Write completed  
input logic [31:0] datawr\_beats\_done; // Beats actually moved  
input logic datawr\_error; // Write error

**Status Outputs:**

output logic ch\_idle; // Channel idle  
output logic ch\_error; // Channel error  
output logic [31:0] ch\_bytes\_xfered; // Bytes transferred

**MonBus Output:**

output logic monbus\_valid;  
input logic monbus\_ready;  
output logic [63:0] monbus\_packet;

### State Machine

#### States

typedef enum logic [3:0] {  
 CH\_IDLE = 4'h0, // Waiting for descriptor  
 CH\_FETCH\_DESC = 4'h1, // Fetching next descriptor  
 CH\_READ\_DATA = 4'h2, // Reading source data  
 CH\_WRITE\_DATA = 4'h3, // Writing destination data  
 CH\_COMPLETE = 4'h4, // Transfer complete  
 CH\_NEXT\_DESC = 4'h5, // Check for chained descriptor  
 CH\_ERROR = 4'hF // Error state  
} channel\_state\_t;

#### State Transitions

IDLE -> FETCH\_DESC: cfg\_enable && initial descriptor request  
FETCH\_DESC -> READ\_DATA: Descriptor received, valid  
READ\_DATA -> WRITE\_DATA: All reads complete (read\_beats\_remaining == 0)  
WRITE\_DATA -> COMPLETE: All writes complete (write\_beats\_remaining == 0)  
COMPLETE -> NEXT\_DESC: Check next\_descriptor\_ptr  
NEXT\_DESC -> FETCH\_DESC: next\_descriptor\_ptr != 0 (chain)  
NEXT\_DESC -> IDLE: next\_descriptor\_ptr == 0 || last flag set  
ANY -> ERROR: Timeout or engine error  
ERROR -> IDLE: Software reset

### Operation

#### Transfer Flow

1. **IDLE:** Wait for cfg\_enable and initial descriptor
2. **FETCH\_DESC:** Request descriptor via desc\_req\_valid
3. **READ\_DATA:**
   * Assert datard\_valid with datard\_beats\_remaining = descriptor.length
   * Wait for datard\_ready (arbiter grants access)
   * Monitor datard\_done\_strobe and decrement read\_beats\_remaining by datard\_beats\_done
   * Continue until read\_beats\_remaining == 0
4. **WRITE\_DATA:**
   * Assert datawr\_valid with datawr\_beats\_remaining = descriptor.length
   * Wait for datawr\_ready (arbiter grants access)
   * Monitor datawr\_done\_strobe and decrement write\_beats\_remaining by datawr\_beats\_done
   * Continue until write\_beats\_remaining == 0
5. **COMPLETE:** Generate MonBus completion packet
6. **NEXT\_DESC:** Check next\_descriptor\_ptr:
   * If != 0: Loop to FETCH\_DESC with new address
   * If == 0 or last flag: Return to IDLE

#### Beat Tracking

**Critical:** Scheduler tracks **total beats remaining**, NOT burst sizes.

// On descriptor receive  
r\_read\_beats\_remaining <= descriptor.length;  
r\_write\_beats\_remaining <= descriptor.length;  
  
// On read completion strobe  
if (datard\_done\_strobe) begin  
 r\_read\_beats\_remaining <= r\_read\_beats\_remaining - datard\_beats\_done;  
end  
  
// On write completion strobe  
if (datawr\_done\_strobe) begin  
 r\_write\_beats\_remaining <= r\_write\_beats\_remaining - datawr\_beats\_done;  
end

**Engines decide burst size internally!** Scheduler just tracks total progress.

### Key Differences from RAPIDS

| Feature | RAPIDS | STREAM |
| --- | --- | --- |
| **Credit Management** | Exponential encoding | None (removed) |
| **Control Engines** | ctrlrd, ctrlwr | None (direct APB) |
| **Address Fixup** | Complex alignment | Aligned only |
| **Length Units** | Chunks (4 bytes) | Beats (DATA\_WIDTH) |
| **Network** | Network master/slave | None |
| **States** | 12+ states | 6 states |
| **Burst Config** | Via interface signals | Engine-internal only |

### Error Handling

#### Timeout Detection

// Increment timeout counter when waiting for engine  
if ((r\_current\_state == CH\_READ\_DATA && !datard\_ready) ||  
 (r\_current\_state == CH\_WRITE\_DATA && !datawr\_ready)) begin  
 r\_timeout\_counter <= r\_timeout\_counter + 1;  
 if (r\_timeout\_counter >= cfg\_timeout) begin  
 w\_next\_state = CH\_ERROR;  
 end  
end

#### Engine Errors

// Transition to error on engine error  
if (datard\_error || datawr\_error) begin  
 w\_next\_state = CH\_ERROR;  
end

#### MonBus Error Reporting

All errors generate MonBus packets with error codes.

### Testing

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

**Test Scenarios:** 1. Single descriptor transfer (read -> write) 2. Chained descriptors (2-3 deep) 3. Engine backpressure handling 4. Timeout detection 5. Engine error propagation 6. Beat counter accuracy (variable burst sizes from engines)

### Related Documentation

* **RAPIDS Scheduler:** projects/components/rapids/docs/rapids\_spec/ch02\_blocks/01\_01\_scheduler.md
* **Architecture:** docs/ARCHITECTURAL\_NOTES.md - Separation of concerns
* **Source:** rtl/stream\_fub/scheduler.sv

**Last Updated:** 2025-10-17 ## AXI Read Engine Specification

**Module:** axi\_read\_engine.sv **Location:** rtl/stream\_fub/ **Status:** To be created

### Overview

The AXI Read Engine autonomously executes AXI read transactions to fetch source data from system memory. It accepts requests from the Scheduler, decides burst lengths internally based on configuration and SRAM space, and reports completion back.

#### Key Features

* **Autonomous burst decision:** Engine decides burst length based on internal config
* **Performance modes:** Low, Medium, High (compile-time parameter)
* **SRAM interface:** Writes fetched data to shared SRAM
* **Streaming pipeline:** No FSM in data path (arbiter-based control)
* **Completion feedback:** Reports beats moved via done\_strobe

### Performance Modes

#### Low Performance Mode (PERFORMANCE = "LOW")

**Target:** Area-optimized, tutorial examples

**Characteristics:** - Single outstanding transaction - Minimal logic - Simple sequential operation - ~250 LUTs (estimate)

**Parameters:**

parameter string PERFORMANCE = "LOW";  
parameter int MAX\_BURST\_LEN = 8; // Fixed 8-beat bursts  
parameter int MAX\_OUTSTANDING = 1; // Single transaction  
parameter bit ENABLE\_PIPELINE = 0; // No pipelining

#### Medium Performance Mode (PERFORMANCE = "MEDIUM")

**Target:** Balanced area/performance for typical FPGA

**Characteristics:** - 2-4 outstanding transactions - Basic pipelining - Adaptive burst sizing - ~400 LUTs (estimate)

**Parameters:**

parameter string PERFORMANCE = "MEDIUM";  
parameter int MAX\_BURST\_LEN = 16; // Up to 16-beat bursts  
parameter int MAX\_OUTSTANDING = 4; // 4 outstanding  
parameter bit ENABLE\_PIPELINE = 1; // Basic pipelining

#### High Performance Mode (PERFORMANCE = "HIGH")

**Target:** Maximum throughput for ASIC or high-end FPGA

**Characteristics:** - 8+ outstanding transactions - Full pipelining - Dynamic burst optimization - ~600 LUTs (estimate)

**Parameters:**

parameter string PERFORMANCE = "HIGH";  
parameter int MAX\_BURST\_LEN = 256; // Up to 256-beat bursts  
parameter int MAX\_OUTSTANDING = 16; // 16 outstanding  
parameter bit ENABLE\_PIPELINE = 1; // Full pipelining

### Interface

#### Parameters

parameter string PERFORMANCE = "LOW"; // "LOW", "MEDIUM", "HIGH"  
parameter int ADDR\_WIDTH = 64; // Address bus width  
parameter int DATA\_WIDTH = 512; // Data bus width  
parameter int MAX\_BURST\_LEN = 8; // Max burst length (perf-dependent)  
parameter int MAX\_OUTSTANDING = 1; // Max outstanding transactions  
parameter bit ENABLE\_PIPELINE = 0; // Pipeline enable  
parameter int SRAM\_DEPTH = 1024; // SRAM depth (for space check)

#### Ports

**Clock and Reset:**

input logic aclk;  
input logic aresetn;

**Configuration:**

input logic [7:0] cfg\_burst\_len; // Configured burst length  
input logic cfg\_enable; // Engine enable

**Scheduler Request Interface:**

input logic datard\_valid; // Scheduler requests read  
output logic datard\_ready; // Engine grants access  
input logic [ADDR\_WIDTH-1:0] datard\_addr; // Start address  
input logic [31:0] datard\_beats\_remaining; // Total beats to read  
input logic [3:0] datard\_channel\_id; // Channel ID  
  
// Completion feedback  
output logic datard\_done\_strobe; // Burst completed  
output logic [31:0] datard\_beats\_done; // Beats actually moved  
output logic datard\_error; // Error occurred

**AXI Master Interface:**

// AXI AR (Address Read) Channel  
output logic [ADDR\_WIDTH-1:0] m\_axi\_araddr;  
output logic [7:0] m\_axi\_arlen; // Burst length - 1  
output logic [2:0] m\_axi\_arsize; // Beat size  
output logic [1:0] m\_axi\_arburst; // INCR  
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, datard\_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)

**SRAM Write Interface:**

output logic sram\_wr\_en;  
output logic [ADDR\_WIDTH-1:0] sram\_wr\_addr;  
output logic [DATA\_WIDTH-1:0] sram\_wr\_data;  
input logic [31:0] sram\_wr\_space; // Available space in beats

**MonBus Output:**

output logic monbus\_valid;  
input logic monbus\_ready;  
output logic [63:0] monbus\_packet;

### Operation

#### Burst Decision Logic

**Critical:** Engine decides burst length autonomously, NOT from scheduler interface.

// Determine burst length based on:  
// 1. Performance mode configuration (MAX\_BURST\_LEN)  
// 2. Runtime configuration (cfg\_burst\_len)  
// 3. Remaining beats (datard\_beats\_remaining)  
// 4. SRAM space available (sram\_wr\_space)  
  
function automatic logic [7:0] calculate\_burst\_len();  
 logic [31:0] max\_possible;  
  
 // Start with configured burst length  
 max\_possible = cfg\_burst\_len;  
  
 // Limit to MAX\_BURST\_LEN (performance mode)  
 if (max\_possible > MAX\_BURST\_LEN)  
 max\_possible = MAX\_BURST\_LEN;  
  
 // Limit to remaining beats  
 if (max\_possible > datard\_beats\_remaining)  
 max\_possible = datard\_beats\_remaining;  
  
 // Limit to SRAM space  
 if (max\_possible > sram\_wr\_space)  
 max\_possible = sram\_wr\_space;  
  
 return max\_possible[7:0];  
endfunction

#### Transaction Flow

**Low Performance (Sequential):**

1. Wait for datard\_valid && SRAM space available  
2. Calculate burst length (limited by config, remaining, SRAM)  
3. Issue AXI AR transaction  
4. Wait for all R beats (rlast)  
5. Assert datard\_done\_strobe with beats\_done count  
6. Repeat until datard\_beats\_remaining == 0

**Medium/High Performance (Pipelined):**

1. Accept datard\_valid && track outstanding transactions  
2. Issue multiple AXI AR transactions (up to MAX\_OUTSTANDING)  
3. Pipeline R data into SRAM  
4. Assert datard\_done\_strobe for each completed burst  
5. Dynamically adjust burst sizes based on SRAM space

#### SRAM Write

**All Performance Modes:**

// On AXI R data valid  
always\_ff @(posedge aclk) begin  
 if (m\_axi\_rvalid && m\_axi\_rready) begin  
 sram\_wr\_en <= 1'b1;  
 sram\_wr\_data <= m\_axi\_rdata;  
 sram\_wr\_addr <= r\_sram\_wr\_ptr;  
 r\_sram\_wr\_ptr <= r\_sram\_wr\_ptr + 1;  
 end else begin  
 sram\_wr\_en <= 1'b0;  
 end  
end

### Architecture by Performance Mode

#### Low Performance Implementation

// Simple state machine for transaction control  
typedef enum logic [1:0] {  
 IDLE = 2'b00,  
 ISSUE = 2'b01,  
 WAIT = 2'b10  
} state\_t;  
  
state\_t r\_state;  
  
always\_ff @(posedge aclk or negedge aresetn) begin  
 if (!aresetn) begin  
 r\_state <= IDLE;  
 end else begin  
 case (r\_state)  
 IDLE: begin  
 if (datard\_valid && sram\_wr\_space >= cfg\_burst\_len)  
 r\_state <= ISSUE;  
 end  
  
 ISSUE: begin  
 if (m\_axi\_arvalid && m\_axi\_arready)  
 r\_state <= WAIT;  
 end  
  
 WAIT: begin  
 if (m\_axi\_rvalid && m\_axi\_rlast)  
 r\_state <= (datard\_beats\_remaining > 0) ? ISSUE : IDLE;  
 end  
 endcase  
 end  
end

#### Medium Performance Implementation

* Outstanding transaction counter
* Basic AR/R channel decoupling
* Adaptive burst sizing based on SRAM fullness

#### High Performance Implementation

* Full AR/R channel pipelining
* Transaction ID tracking
* Out-of-order completion handling
* Dynamic burst optimization
* Prefetch lookahead

### Error Handling

#### AXI Errors

// On RRESP != OKAY  
if (m\_axi\_rvalid && m\_axi\_rresp != 2'b00) begin  
 datard\_error <= 1'b1;  
 // Generate MonBus error packet  
end

#### Timeout Detection

// Timeout if no progress for cfg\_timeout cycles  
if (datard\_valid && !transaction\_progress) begin  
 r\_timeout\_counter <= r\_timeout\_counter + 1;  
 if (r\_timeout\_counter >= cfg\_timeout) begin  
 datard\_error <= 1'b1;  
 end  
end

### Testing

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

**Test Scenarios (per performance mode):** 1. Single burst read 2. Multi-burst transfer (beats > MAX\_BURST\_LEN) 3. SRAM backpressure handling 4. Variable burst sizing 5. AXI error response 6. Timeout detection 7. Outstanding transaction limits (Medium/High)

### Performance Comparison

| Metric | Low | Medium | High |
| --- | --- | --- | --- |
| **Area (LUTs)** | ~250 | ~400 | ~600 |
| **Max Throughput** | 50% | 75% | 95% |
| **Outstanding Txns** | 1 | 4 | 16 |
| **Burst Length** | 8 | 16 | 256 |
| **Pipelining** | None | Basic | Full |
| **Use Case** | Tutorial | Typical | High-perf |

### Related Documentation

* **Scheduler:** 02\_scheduler.md - Interface contract
* **Architecture:** docs/ARCHITECTURAL\_NOTES.md - Separation of concerns
* **AXI4 Protocol:** ARM IHI0022E

**Last Updated:** 2025-10-17 ## AXI Write Engine Specification

**Module:** axi\_write\_engine.sv **Location:** rtl/stream\_fub/ **Status:** To be created

### Overview

The AXI Write Engine autonomously executes AXI write transactions to store data to system memory. It accepts requests from the Scheduler, decides burst lengths internally based on configuration and SRAM data availability, and reports completion back.

#### Key Features

* **Autonomous burst decision:** Engine decides burst length based on internal config
* **Performance modes:** Low, Medium, High (compile-time parameter)
* **SRAM interface:** Reads data from shared SRAM
* **Streaming pipeline:** No FSM in data path (arbiter-based control)
* **Completion feedback:** Reports beats moved via done\_strobe

### Performance Modes

#### Low Performance Mode (PERFORMANCE = "LOW")

**Target:** Area-optimized, tutorial examples

**Characteristics:** - Single outstanding transaction - Minimal logic - Simple sequential operation - ~250 LUTs (estimate)

**Parameters:**

parameter string PERFORMANCE = "LOW";  
parameter int MAX\_BURST\_LEN = 16; // Fixed 16-beat bursts  
parameter int MAX\_OUTSTANDING = 1; // Single transaction  
parameter bit ENABLE\_PIPELINE = 0; // No pipelining

#### Medium Performance Mode (PERFORMANCE = "MEDIUM")

**Target:** Balanced area/performance for typical FPGA

**Characteristics:** - 2-4 outstanding transactions - Basic pipelining - Adaptive burst sizing - ~400 LUTs (estimate)

**Parameters:**

parameter string PERFORMANCE = "MEDIUM";  
parameter int MAX\_BURST\_LEN = 32; // Up to 32-beat bursts  
parameter int MAX\_OUTSTANDING = 4; // 4 outstanding  
parameter bit ENABLE\_PIPELINE = 1; // Basic pipelining

#### High Performance Mode (PERFORMANCE = "HIGH")

**Target:** Maximum throughput for ASIC or high-end FPGA

**Characteristics:** - 8+ outstanding transactions - Full pipelining - Dynamic burst optimization - ~600 LUTs (estimate)

**Parameters:**

parameter string PERFORMANCE = "HIGH";  
parameter int MAX\_BURST\_LEN = 256; // Up to 256-beat bursts  
parameter int MAX\_OUTSTANDING = 16; // 16 outstanding  
parameter bit ENABLE\_PIPELINE = 1; // Full pipelining

### Interface

#### Parameters

parameter string PERFORMANCE = "LOW"; // "LOW", "MEDIUM", "HIGH"  
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 MAX\_BURST\_LEN = 16; // Max burst length (perf-dependent)  
parameter int MAX\_OUTSTANDING = 1; // Max outstanding transactions  
parameter bit ENABLE\_PIPELINE = 0; // Pipeline enable  
parameter int SRAM\_DEPTH = 1024; // SRAM depth (for data check)

#### Ports

**Clock and Reset:**

input logic aclk;  
input logic aresetn;

**Configuration:**

input logic [7:0] cfg\_burst\_len; // Configured burst length  
input logic cfg\_enable; // Engine enable

**Scheduler Request Interface:**

input logic datawr\_valid; // Scheduler requests write  
output logic datawr\_ready; // Engine grants access  
input logic [ADDR\_WIDTH-1:0] datawr\_addr; // Start address  
input logic [31:0] datawr\_beats\_remaining; // Total beats to write  
input logic [3:0] datawr\_channel\_id; // Channel ID  
  
// Completion feedback  
output logic datawr\_done\_strobe; // Burst completed  
output logic [31:0] datawr\_beats\_done; // Beats actually moved  
output logic datawr\_error; // Error occurred

**AXI Master Interface:**

// AXI AW (Address Write) Channel  
output logic [ADDR\_WIDTH-1:0] m\_axi\_awaddr;  
output logic [7:0] m\_axi\_awlen; // Burst length - 1  
output logic [2:0] m\_axi\_awsize; // Beat size  
output logic [1:0] m\_axi\_awburst; // INCR  
output logic [AXI\_ID\_WIDTH-1:0] m\_axi\_awid; // Transaction ID  
output logic m\_axi\_awvalid;  
input logic m\_axi\_awready;  
  
// AXI W (Write Data) Channel  
output logic [DATA\_WIDTH-1:0] m\_axi\_wdata;  
output logic [DATA\_WIDTH/8-1:0] m\_axi\_wstrb;  
output logic m\_axi\_wlast;  
output logic m\_axi\_wvalid;  
input logic m\_axi\_wready;  
  
// AXI B (Write Response) Channel  
input logic [AXI\_ID\_WIDTH-1:0] m\_axi\_bid; // Transaction ID  
input logic [1:0] m\_axi\_bresp;  
input logic m\_axi\_bvalid;  
output logic m\_axi\_bready;

**Critical AXI ID Requirement:**

The lower bits of m\_axi\_awid **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\_awid = {transaction\_counter, datawr\_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)

**SRAM Read Interface:**

output logic sram\_rd\_en;  
output logic [ADDR\_WIDTH-1:0] sram\_rd\_addr;  
input logic [DATA\_WIDTH-1:0] sram\_rd\_data;  
input logic [31:0] sram\_rd\_avail; // Available data in beats

**MonBus Output:**

output logic monbus\_valid;  
input logic monbus\_ready;  
output logic [63:0] monbus\_packet;

### Operation

#### Burst Decision Logic

**Critical:** Engine decides burst length autonomously, NOT from scheduler interface.

// Determine burst length based on:  
// 1. Performance mode configuration (MAX\_BURST\_LEN)  
// 2. Runtime configuration (cfg\_burst\_len)  
// 3. Remaining beats (datawr\_beats\_remaining)  
// 4. SRAM data available (sram\_rd\_avail)  
  
function automatic logic [7:0] calculate\_burst\_len();  
 logic [31:0] max\_possible;  
  
 // Start with configured burst length  
 max\_possible = cfg\_burst\_len;  
  
 // Limit to MAX\_BURST\_LEN (performance mode)  
 if (max\_possible > MAX\_BURST\_LEN)  
 max\_possible = MAX\_BURST\_LEN;  
  
 // Limit to remaining beats  
 if (max\_possible > datawr\_beats\_remaining)  
 max\_possible = datawr\_beats\_remaining;  
  
 // Limit to SRAM data availability  
 if (max\_possible > sram\_rd\_avail)  
 max\_possible = sram\_rd\_avail;  
  
 return max\_possible[7:0];  
endfunction

#### Transaction Flow

**Low Performance (Sequential):**

1. Wait for datawr\_valid && SRAM data available  
2. Calculate burst length (limited by config, remaining, SRAM data)  
3. Issue AXI AW transaction  
4. Stream W data from SRAM (assert wlast on final beat)  
5. Wait for B response  
6. Assert datawr\_done\_strobe with beats\_done count  
7. Repeat until datawr\_beats\_remaining == 0

**Medium/High Performance (Pipelined):**

1. Accept datawr\_valid && track outstanding transactions  
2. Issue multiple AXI AW transactions (up to MAX\_OUTSTANDING)  
3. Pipeline W data from SRAM  
4. Process B responses asynchronously  
5. Assert datawr\_done\_strobe for each completed burst  
6. Dynamically adjust burst sizes based on SRAM data availability

#### SRAM Read

**All Performance Modes:**

// Read data from SRAM for AXI W channel  
always\_ff @(posedge aclk) begin  
 if (m\_axi\_wvalid && m\_axi\_wready) begin  
 sram\_rd\_en <= 1'b1;  
 sram\_rd\_addr <= r\_sram\_rd\_ptr;  
 r\_sram\_rd\_ptr <= r\_sram\_rd\_ptr + 1;  
 end else begin  
 sram\_rd\_en <= 1'b0;  
 end  
end  
  
// Pipeline SRAM data to AXI W  
always\_ff @(posedge aclk) begin  
 if (sram\_rd\_en) begin  
 m\_axi\_wdata <= sram\_rd\_data;  
 m\_axi\_wstrb <= {(DATA\_WIDTH/8){1'b1}}; // Full strobes  
 end  
end

### Architecture by Performance Mode

#### Low Performance Implementation

// Simple state machine for transaction control  
typedef enum logic [2:0] {  
 IDLE = 3'b000,  
 ISSUE\_AW = 3'b001,  
 STREAM\_W = 3'b010,  
 WAIT\_B = 3'b011  
} state\_t;  
  
state\_t r\_state;  
  
always\_ff @(posedge aclk or negedge aresetn) begin  
 if (!aresetn) begin  
 r\_state <= IDLE;  
 end else begin  
 case (r\_state)  
 IDLE: begin  
 if (datawr\_valid && sram\_rd\_avail >= cfg\_burst\_len)  
 r\_state <= ISSUE\_AW;  
 end  
  
 ISSUE\_AW: begin  
 if (m\_axi\_awvalid && m\_axi\_awready)  
 r\_state <= STREAM\_W;  
 end  
  
 STREAM\_W: begin  
 if (m\_axi\_wvalid && m\_axi\_wlast && m\_axi\_wready)  
 r\_state <= WAIT\_B;  
 end  
  
 WAIT\_B: begin  
 if (m\_axi\_bvalid && m\_axi\_bready)  
 r\_state <= (datawr\_beats\_remaining > 0) ? ISSUE\_AW : IDLE;  
 end  
 endcase  
 end  
end

#### Medium Performance Implementation

* Outstanding transaction counter
* AW/W/B channel decoupling
* Adaptive burst sizing based on SRAM data availability

#### High Performance Implementation

* Full AW/W/B channel pipelining
* Transaction ID tracking
* Out-of-order completion handling
* Dynamic burst optimization
* SRAM read prefetch

### Asymmetric Burst Lengths

**Note:** Write engine can use different burst lengths than read engine.

**Example Configuration:**

// Read engine: 8 beats per burst  
axi\_read\_engine #(.MAX\_BURST\_LEN(8)) u\_rd\_engine (...);  
  
// Write engine: 16 beats per burst (2x read)  
axi\_write\_engine #(.MAX\_BURST\_LEN(16)) u\_wr\_engine (...);

**Why This Works:** - SRAM buffer decouples read and write rates - Scheduler doesn’t care about burst sizing differences - Each engine optimizes independently

### Error Handling

#### AXI Errors

// On BRESP != OKAY  
if (m\_axi\_bvalid && m\_axi\_bresp != 2'b00) begin  
 datawr\_error <= 1'b1;  
 // Generate MonBus error packet  
end

#### Timeout Detection

// Timeout if no progress for cfg\_timeout cycles  
if (datawr\_valid && !transaction\_progress) begin  
 r\_timeout\_counter <= r\_timeout\_counter + 1;  
 if (r\_timeout\_counter >= cfg\_timeout) begin  
 datawr\_error <= 1'b1;  
 end  
end

### Testing

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

**Test Scenarios (per performance mode):** 1. Single burst write 2. Multi-burst transfer (beats > MAX\_BURST\_LEN) 3. SRAM data availability backpressure 4. Variable burst sizing 5. AXI error response 6. Timeout detection 7. Outstanding transaction limits (Medium/High) 8. Asymmetric burst lengths with read engine

### Performance Comparison

| Metric | Low | Medium | High |
| --- | --- | --- | --- |
| **Area (LUTs)** | ~250 | ~400 | ~600 |
| **Max Throughput** | 50% | 75% | 95% |
| **Outstanding Txns** | 1 | 4 | 16 |
| **Burst Length** | 16 | 32 | 256 |
| **Pipelining** | None | Basic | Full |
| **Use Case** | Tutorial | Typical | High-perf |

### Related Documentation

* **Scheduler:** 02\_scheduler.md - Interface contract
* **Read Engine:** 03\_axi\_read\_engine.md - Companion read engine
* **Architecture:** docs/ARCHITECTURAL\_NOTES.md - Separation of concerns
* **AXI4 Protocol:** ARM IHI0022E

**Last Updated:** 2025-10-17 ## SRAM Controller Specification

**Module:** sram\_controller.sv **Location:** rtl/stream\_fub/ **Status:** To be created

### Overview

The SRAM Controller provides a monolithic buffer interface that is internally partitioned across 8 independent channels. Each channel gets its own address space, write/read pointers, and space/availability tracking, while the physical SRAM implementation is abstracted.

#### Key Features

* **Monolithic interface:** Single SRAM controller at top level
* **Per-channel partitioning:** 8 independent channel buffers
* **Dedicated pointers:** Each channel has own write/read pointers
* **Space tracking:** Write interface reports free lines available
* **Availability tracking:** Read interface reports ready lines to drain
* **Overflow protection:** Per-channel full/empty detection
* **Physical abstraction:** May use one large SRAM or multiple discrete SRAMs internally

### Architecture

#### Conceptual Partitioning

Monolithic SRAM Controller (64 KB total)  
|  
|-- Channel 0: Base 0x0000, Size 8 KB (128 lines x 64B)  
|-- Channel 1: Base 0x2000, Size 8 KB (128 lines x 64B)  
|-- Channel 2: Base 0x4000, Size 8 KB (128 lines x 64B)  
|-- Channel 3: Base 0x6000, Size 8 KB (128 lines x 64B)  
|-- Channel 4: Base 0x8000, Size 8 KB (128 lines x 64B)  
|-- Channel 5: Base 0xA000, Size 8 KB (128 lines x 64B)  
|-- Channel 6: Base 0xC000, Size 8 KB (128 lines x 64B)  
`-- Channel 7: Base 0xE000, Size 8 KB (128 lines x 64B)

#### Physical Implementation Options

**Option 1: Single Large SRAM** - One 1024-line x 512-bit SRAM - Address decode: {channel\_id[2:0], line\_offset[6:0]} - Simple, single clock domain

**Option 2: Per-Channel SRAMs** - Eight 128-line x 512-bit SRAMs - Independent instances for each channel - Better for banking/power gating

### Interface

#### Parameters

parameter int NUM\_CHANNELS = 8; // Fixed at 8 for STREAM  
parameter int DATA\_WIDTH = 512; // Data width in bits  
parameter int LINES\_PER\_CHANNEL = 128; // Buffer depth per channel  
parameter int ADDR\_WIDTH = $clog2(LINES\_PER\_CHANNEL); // 7 bits  
localparam int TOTAL\_LINES = NUM\_CHANNELS \* LINES\_PER\_CHANNEL; // 1024

#### Ports

**Clock and Reset:**

input logic aclk;  
input logic aresetn;

**Per-Channel Write Interface:**

// Channel 0-7 write ports  
input logic [NUM\_CHANNELS-1:0] ch\_wr\_en;  
input logic [NUM\_CHANNELS-1:0][DATA\_WIDTH-1:0]  
 ch\_wr\_data;  
output logic [NUM\_CHANNELS-1:0][ADDR\_WIDTH:0]  
 ch\_wr\_free; // Free lines available

**Per-Channel Read Interface:**

// Channel 0-7 read ports  
input logic [NUM\_CHANNELS-1:0] ch\_rd\_en;  
output logic [NUM\_CHANNELS-1:0][DATA\_WIDTH-1:0]  
 ch\_rd\_data;  
output logic [NUM\_CHANNELS-1:0][ADDR\_WIDTH:0]  
 ch\_rd\_avail; // Ready lines to drain

**Status Outputs (per channel):**

output logic [NUM\_CHANNELS-1:0] ch\_full;  
output logic [NUM\_CHANNELS-1:0] ch\_empty;  
output logic [NUM\_CHANNELS-1:0] ch\_overflow; // Overflow error  
output logic [NUM\_CHANNELS-1:0] ch\_underflow; // Underflow error

### Operation

#### Per-Channel Pointer Management

Each channel maintains independent write and read pointers:

// Per-channel state (replicated 8 times)  
logic [ADDR\_WIDTH-1:0] r\_wr\_ptr[NUM\_CHANNELS]; // Write pointer  
logic [ADDR\_WIDTH-1:0] r\_rd\_ptr[NUM\_CHANNELS]; // Read pointer  
logic [ADDR\_WIDTH:0] r\_count[NUM\_CHANNELS]; // Occupancy counter

#### Write Operation

**For each channel independently:**

// Channel i write  
always\_ff @(posedge aclk or negedge aresetn) begin  
 if (!aresetn) begin  
 r\_wr\_ptr[i] <= '0;  
 end else if (ch\_wr\_en[i]) begin  
 if (!ch\_full[i]) begin  
 // Write to SRAM at channel's partition  
 sram\_wr\_addr = {i[2:0], r\_wr\_ptr[i]}; // Channel base + offset  
 sram\_wr\_data = ch\_wr\_data[i];  
 sram\_wr\_en = 1'b1;  
  
 // Advance write pointer (wraps within channel)  
 r\_wr\_ptr[i] <= r\_wr\_ptr[i] + 1;  
 r\_count[i] <= r\_count[i] + 1;  
 end else begin  
 ch\_overflow[i] <= 1'b1; // Overflow error  
 end  
 end  
end

#### Read Operation

**For each channel independently:**

// Channel i read  
always\_ff @(posedge aclk or negedge aresetn) begin  
 if (!aresetn) begin  
 r\_rd\_ptr[i] <= '0;  
 end else if (ch\_rd\_en[i]) begin  
 if (!ch\_empty[i]) begin  
 // Read from SRAM at channel's partition  
 sram\_rd\_addr = {i[2:0], r\_rd\_ptr[i]};  
 sram\_rd\_en = 1'b1;  
  
 // Advance read pointer (wraps within channel)  
 r\_rd\_ptr[i] <= r\_rd\_ptr[i] + 1;  
 r\_count[i] <= r\_count[i] - 1;  
 end else begin  
 ch\_underflow[i] <= 1'b1; // Underflow error  
 end  
 end  
end  
  
// Read data available next cycle (SRAM read latency = 1)  
assign ch\_rd\_data[i] = sram\_rd\_data\_q;

#### Space Tracking

**Free lines for writes:**

// Free space available for writes  
assign ch\_wr\_free[i] = LINES\_PER\_CHANNEL - r\_count[i];

**Available lines for reads:**

// Data available for reads  
assign ch\_rd\_avail[i] = r\_count[i];

#### Full/Empty Detection

// Per-channel status  
assign ch\_full[i] = (r\_count[i] == LINES\_PER\_CHANNEL);  
assign ch\_empty[i] = (r\_count[i] == 0);

### Physical SRAM Instantiation

#### Option 1: Single Monolithic SRAM

// Single 1024-line SRAM shared across all channels  
simple\_sram #(  
 .DATA\_WIDTH(512),  
 .ADDR\_WIDTH(10), // 1024 lines = 8 channels x 128 lines  
 .CHUNK\_WIDTH(64)  
) u\_sram (  
 .aclk(aclk),  
 .aresetn(aresetn),  
  
 // Write port (arbitrated across channels)  
 .wr\_en(w\_sram\_wr\_en),  
 .wr\_addr(w\_sram\_wr\_addr), // {ch\_id[2:0], line\_offset[6:0]}  
 .wr\_data(w\_sram\_wr\_data),  
 .wr\_chunk\_en({8{1'b1}}), // All chunks enabled  
  
 // Read port (arbitrated across channels)  
 .rd\_en(w\_sram\_rd\_en),  
 .rd\_addr(w\_sram\_rd\_addr), // {ch\_id[2:0], line\_offset[6:0]}  
 .rd\_data(w\_sram\_rd\_data)  
);

#### Option 2: Per-Channel SRAMs

// Replicate 8 times (one per channel)  
generate  
 for (genvar ch = 0; ch < NUM\_CHANNELS; ch++) begin : gen\_sram  
 simple\_sram #(  
 .DATA\_WIDTH(512),  
 .ADDR\_WIDTH(7), // 128 lines per channel  
 .CHUNK\_WIDTH(64)  
 ) u\_ch\_sram (  
 .aclk(aclk),  
 .aresetn(aresetn),  
  
 // Write port (dedicated to this channel)  
 .wr\_en(ch\_wr\_en[ch] && !ch\_full[ch]),  
 .wr\_addr(r\_wr\_ptr[ch]),  
 .wr\_data(ch\_wr\_data[ch]),  
 .wr\_chunk\_en({8{1'b1}}),  
  
 // Read port (dedicated to this channel)  
 .rd\_en(ch\_rd\_en[ch] && !ch\_empty[ch]),  
 .rd\_addr(r\_rd\_ptr[ch]),  
 .rd\_data(ch\_rd\_data[ch])  
 );  
 end  
endgenerate

### Integration with AXI Engines

#### AXI Read Engine -> SRAM Write

// Read engine writes fetched data to SRAM  
axi\_read\_engine u\_rd\_engine (  
 // ... AXI master interface  
  
 // SRAM controller interface (channel selected by arbiter)  
 .sram\_wr\_en(ch\_wr\_en[granted\_ch\_id]),  
 .sram\_wr\_data(ch\_wr\_data[granted\_ch\_id]),  
 .sram\_wr\_free(ch\_wr\_free[granted\_ch\_id]) // Backpressure signal  
);

#### SRAM Read -> AXI Write Engine

// Write engine reads data from SRAM  
axi\_write\_engine u\_wr\_engine (  
 // ... AXI master interface  
  
 // SRAM controller interface (channel selected by arbiter)  
 .sram\_rd\_en(ch\_rd\_en[granted\_ch\_id]),  
 .sram\_rd\_data(ch\_rd\_data[granted\_ch\_id]),  
 .sram\_rd\_avail(ch\_rd\_avail[granted\_ch\_id]) // Data available  
);

### Arbiter Integration

The SRAM controller accepts per-channel signals, but the AXI engines are shared resources. Arbiters select which channel has access:

// Write arbiter: Grants one channel access to AXI read engine  
channel\_arbiter u\_write\_arbiter (  
 .requests(ch\_datard\_req),  
 .grant\_id(wr\_grant\_ch\_id),  
 .grant\_valid(wr\_grant\_valid)  
);  
  
// Read arbiter: Grants one channel access to AXI write engine  
channel\_arbiter u\_read\_arbiter (  
 .requests(ch\_datawr\_req),  
 .grant\_id(rd\_grant\_ch\_id),  
 .grant\_valid(rd\_grant\_valid)  
);  
  
// Mux channel signals to engines based on grant  
assign engine\_sram\_wr\_en = ch\_wr\_en[wr\_grant\_ch\_id] && wr\_grant\_valid;  
assign engine\_sram\_rd\_en = ch\_rd\_en[rd\_grant\_ch\_id] && rd\_grant\_valid;

### Differences from RAPIDS

| Feature | RAPIDS | STREAM |
| --- | --- | --- |
| **Partitioning** | Dynamic (credit-based) | Fixed per-channel |
| **Address Space** | Shared with complex allocation | Fixed base per channel |
| **Chunk Support** | Full partial write support | Aligned only (all chunks) |
| **Overflow Handling** | Credit system prevents | Error flag on overflow |
| **Configuration** | Runtime configurable sizes | Compile-time fixed sizes |

### Buffer Sizing

**Per-Channel Allocation:** - 128 lines x 512 bits = 128 lines x 64 bytes = 8 KB per channel - Total: 8 channels x 8 KB = 64 KB

**Typical Transfer:** - Descriptor length: 64 beats - Channel buffer: 128 lines - Can hold 2x typical descriptor (allows pipelining)

**Overflow Condition:** - If read engine fills buffer faster than write engine drains - ch\_wr\_free goes to 0 - Read engine must stall (backpressure)

**Underflow Condition:** - If write engine tries to read before data available - ch\_rd\_avail is 0 - Write engine must wait

### Error Handling

#### Overflow Detection

// Write when full  
always\_ff @(posedge aclk) begin  
 if (ch\_wr\_en[i] && ch\_full[i]) begin  
 ch\_overflow[i] <= 1'b1;  
 // Generate MonBus error packet  
 end  
end

#### Underflow Detection

// Read when empty  
always\_ff @(posedge aclk) begin  
 if (ch\_rd\_en[i] && ch\_empty[i]) begin  
 ch\_underflow[i] <= 1'b1;  
 // Generate MonBus error packet  
 end  
end

### Testing

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

**Test Scenarios:** 1. Single channel write/read (independent operation) 2. All 8 channels active simultaneously 3. Overflow detection (write to full buffer) 4. Underflow detection (read from empty buffer) 5. Wrap-around pointer behavior 6. Space/availability tracking accuracy 7. Concurrent multi-channel operations

### Performance

**Throughput:** 1 write + 1 read per cycle (per channel, if using per-channel SRAMs)

**Latency:** - Write to SRAM: 1 cycle - Read from SRAM: 1 cycle (registered output) - Space/availability updates: Combinational (same cycle)

**Area Estimate:** - Controller logic: ~200 LUTs per channel x 8 = ~1,600 LUTs - SRAM: 1024 lines x 512 bits = 64 KB - Total: ~1,600 LUTs + 64 KB

### Related Documentation

* **Simple SRAM:** fub\_07\_simple\_sram.md - Physical SRAM primitive
* **AXI Read Engine:** fub\_03\_axi\_read\_engine.md - SRAM write interface consumer
* **AXI Write Engine:** fub\_04\_axi\_write\_engine.md - SRAM read interface consumer
* **RAPIDS SRAM Controllers:** projects/components/rapids/ - Reference implementation

**Last Updated:** 2025-10-17 ## Simple SRAM Specification

**Module:** simple\_sram.sv **Location:** rtl/stream\_fub/ **Source:** Copied from RAPIDS (no changes)

### Overview

The Simple SRAM provides dual-port buffering between read and write engines. This module is **identical to RAPIDS** with no modifications.

#### Key Features

* Dual-port synchronous SRAM
* Independent read and write ports
* Configurable depth and width
* Chunk enable support (for partial writes)
* Single clock domain

### Interface

#### Parameters

parameter int DATA\_WIDTH = 512; // Data width in bits  
parameter int ADDR\_WIDTH = 10; // Address width (depth = 2^ADDR\_WIDTH)  
parameter int CHUNK\_WIDTH = 64; // Chunk width for enables  
localparam int NUM\_CHUNKS = DATA\_WIDTH / CHUNK\_WIDTH;

#### Ports

**Clock and Reset:**

input logic aclk;  
input logic aresetn;

**Write Port:**

input logic wr\_en;  
input logic [ADDR\_WIDTH-1:0] wr\_addr;  
input logic [DATA\_WIDTH-1:0] wr\_data;  
input logic [NUM\_CHUNKS-1:0] wr\_chunk\_en; // Per-chunk write enable

**Read Port:**

input logic rd\_en;  
input logic [ADDR\_WIDTH-1:0] rd\_addr;  
output logic [DATA\_WIDTH-1:0] rd\_data;

### Operation

#### Write Operation

// Synchronous write with chunk enables  
always\_ff @(posedge aclk) begin  
 if (wr\_en) begin  
 for (int i = 0; i < NUM\_CHUNKS; i++) begin  
 if (wr\_chunk\_en[i]) begin  
 mem[wr\_addr][(i+1)\*CHUNK\_WIDTH-1 -: CHUNK\_WIDTH] <=  
 wr\_data[(i+1)\*CHUNK\_WIDTH-1 -: CHUNK\_WIDTH];  
 end  
 end  
 end  
end

#### Read Operation

// Synchronous read with registered output  
always\_ff @(posedge aclk) begin  
 if (rd\_en) begin  
 rd\_data <= mem[rd\_addr];  
 end  
end

**Read Latency:** 1 cycle (registered output)

### Usage in STREAM

#### Buffer Decoupling

SRAM decouples read and write engine timing:

AXI Read Engine -> SRAM Write Port  
 |  
 [Buffer]  
 |  
 SRAM Read Port -> AXI Write Engine

**Benefits:** - Read and write engines operate at different rates - Absorbs backpressure from either direction - Enables asymmetric burst lengths (read=8, write=16)

#### Pointer Management

**External logic tracks pointers:**

// Write pointer (managed by read engine)  
logic [ADDR\_WIDTH-1:0] wr\_ptr;  
always\_ff @(posedge aclk) begin  
 if (axi\_read\_complete) begin  
 wr\_ptr <= wr\_ptr + beats\_read;  
 end  
end  
  
// Read pointer (managed by write engine)  
logic [ADDR\_WIDTH-1:0] rd\_ptr;  
always\_ff @(posedge aclk) begin  
 if (axi\_write\_complete) begin  
 rd\_ptr <= rd\_ptr + beats\_written;  
 end  
end  
  
// Space calculation  
assign sram\_wr\_space = SRAM\_DEPTH - (wr\_ptr - rd\_ptr);  
assign sram\_rd\_avail = wr\_ptr - rd\_ptr;

### Chunk Enable Usage

**Purpose:** Support partial writes for unaligned transfers.

**Note:** STREAM requires aligned addresses, so chunk enables are typically all ’1.

// STREAM: All chunks enabled (aligned addresses)  
assign wr\_chunk\_en = {NUM\_CHUNKS{1'b1}};  
  
// RAPIDS: May have partial chunk enables for alignment  
assign wr\_chunk\_en = alignment\_logic(...);

### Differences from RAPIDS

**None.** This module is identical to RAPIDS simple\_sram.sv.

### Typical Configuration

**For 512-bit data width:**

simple\_sram #(  
 .DATA\_WIDTH(512), // 64 bytes per beat  
 .ADDR\_WIDTH(10), // 1024 entries  
 .CHUNK\_WIDTH(64) // 8-byte chunks  
) u\_sram (  
 .aclk(aclk),  
 .aresetn(aresetn),  
 // ... ports  
);

**Memory size:** 1024 entries 64 bytes = 64 KB

### Testing

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

**Test Scenarios:** 1. Basic write -> read 2. Concurrent read/write (different addresses) 3. Full buffer fill/drain 4. Chunk enable combinations (if used)

**Reference:** RAPIDS simple\_sram tests can be reused directly.

### Related Documentation

* **RAPIDS Spec:** projects/components/rapids/docs/rapids\_spec/ (if available)
* **Source:** rtl/stream\_fub/simple\_sram.sv

**Last Updated:** 2025-10-17 ## Channel Arbiter Specification

**Module:** channel\_arbiter.sv **Location:** rtl/stream\_macro/ **Status:** To be created

### Overview

The Channel Arbiter manages access to shared resources (descriptor fetch, data read, data write AXI masters) across 8 independent STREAM channels. It implements priority-based arbitration with round-robin fairness within the same priority level.

#### Key Features

* **8 channels:** Fixed maximum (configurable via parameter)
* **Priority-based:** Uses descriptor priority field (8-bit)
* **Round-robin:** Within same priority level
* **Timeout protection:** Prevents starvation
* **Separate arbiters:** Independent for descriptor, read, write paths

### Arbitration Scheme

#### Priority Levels

**Descriptor priority field:** 8-bit value from descriptor - **Higher value = higher priority** - **Range:** 0 (lowest) to 255 (highest)

#### Round-Robin Within Priority

Channels with same priority rotate fairly:  
 Priority 7: CH0 -> CH3 -> CH0 -> CH3 -> ...  
 Priority 5: CH1 -> CH2 -> CH1 -> CH2 -> ...  
  
Between priorities: Higher always wins  
 Priority 7 CH0 > Priority 5 CH1

#### Timeout/Starvation Prevention

// If a channel waits too long, boost priority temporarily  
if (r\_wait\_cycles[ch\_id] > cfg\_timeout\_threshold) begin  
 effective\_priority = MAX\_PRIORITY;  
end

### Interface

#### Parameters

parameter int NUM\_CHANNELS = 8; // Fixed at 8 for STREAM  
parameter int PRIORITY\_WIDTH = 8; // Priority field width

#### Ports

**Clock and Reset:**

input logic aclk;  
input logic aresetn;

**Configuration:**

input logic [31:0] cfg\_timeout\_threshold;

**Channel Requests (Descriptor Path):**

input logic [NUM\_CHANNELS-1:0] desc\_req; // Request signals  
input logic [NUM\_CHANNELS-1:0][PRIORITY\_WIDTH-1:0]  
 desc\_priority; // Priority per channel  
output logic [NUM\_CHANNELS-1:0] desc\_grant; // Grant signals  
output logic [$clog2(NUM\_CHANNELS)-1:0]  
 desc\_grant\_id; // Granted channel ID  
output logic desc\_grant\_valid; // Grant valid

**Channel Requests (Data Read Path):**

input logic [NUM\_CHANNELS-1:0] datard\_req;  
input logic [NUM\_CHANNELS-1:0][PRIORITY\_WIDTH-1:0]  
 datard\_priority;  
output logic [NUM\_CHANNELS-1:0] datard\_grant;  
output logic [$clog2(NUM\_CHANNELS)-1:0]  
 datard\_grant\_id;  
output logic datard\_grant\_valid;

**Channel Requests (Data Write Path):**

input logic [NUM\_CHANNELS-1:0] datawr\_req;  
input logic [NUM\_CHANNELS-1:0][PRIORITY\_WIDTH-1:0]  
 datawr\_priority;  
output logic [NUM\_CHANNELS-1:0] datawr\_grant;  
output logic [$clog2(NUM\_CHANNELS)-1:0]  
 datawr\_grant\_id;  
output logic datawr\_grant\_valid;

### Arbitration Algorithm

#### Priority Encoder with Round-Robin

function automatic logic [$clog2(NUM\_CHANNELS)-1:0]  
 arbitrate(  
 logic [NUM\_CHANNELS-1:0] requests,  
 logic [NUM\_CHANNELS-1:0][PRIORITY\_WIDTH-1:0] priorities,  
 logic [$clog2(NUM\_CHANNELS)-1:0] last\_grant  
 );  
  
 logic [PRIORITY\_WIDTH-1:0] max\_priority;  
 logic [NUM\_CHANNELS-1:0] max\_priority\_mask;  
 logic [$clog2(NUM\_CHANNELS)-1:0] grant\_id;  
  
 // Find maximum priority among requesters  
 max\_priority = 0;  
 for (int i = 0; i < NUM\_CHANNELS; i++) begin  
 if (requests[i] && priorities[i] > max\_priority) begin  
 max\_priority = priorities[i];  
 end  
 end  
  
 // Mask channels with max priority  
 for (int i = 0; i < NUM\_CHANNELS; i++) begin  
 max\_priority\_mask[i] = requests[i] && (priorities[i] == max\_priority);  
 end  
  
 // Round-robin among max priority channels  
 grant\_id = round\_robin\_select(max\_priority\_mask, last\_grant);  
  
 return grant\_id;  
endfunction

#### Round-Robin Selection

function automatic logic [$clog2(NUM\_CHANNELS)-1:0]  
 round\_robin\_select(  
 logic [NUM\_CHANNELS-1:0] candidates,  
 logic [$clog2(NUM\_CHANNELS)-1:0] last\_grant  
 );  
  
 // Start searching from last\_grant + 1  
 for (int offset = 1; offset <= NUM\_CHANNELS; offset++) begin  
 int idx = (last\_grant + offset) % NUM\_CHANNELS;  
 if (candidates[idx]) begin  
 return idx;  
 end  
 end  
  
 // Fallback (shouldn't reach here if candidates != 0)  
 return 0;  
endfunction

### Operation

#### Grant Cycle

1. All channels assert request signals with priority  
2. Arbiter determines winner based on:  
 a. Highest priority wins  
 b. Among same priority: round-robin from last grant  
3. Arbiter asserts grant for one cycle  
4. Winning channel captures grant and proceeds  
5. Arbiter ready for next arbitration

#### Timeout Boost

// Track wait time per channel  
always\_ff @(posedge aclk) begin  
 for (int i = 0; i < NUM\_CHANNELS; i++) begin  
 if (desc\_req[i] && !desc\_grant[i]) begin  
 r\_desc\_wait\_cycles[i] <= r\_desc\_wait\_cycles[i] + 1;  
 end else begin  
 r\_desc\_wait\_cycles[i] <= 0;  
 end  
 end  
end  
  
// Boost priority if timeout  
logic [NUM\_CHANNELS-1:0][PRIORITY\_WIDTH-1:0] effective\_priority;  
always\_comb begin  
 for (int i = 0; i < NUM\_CHANNELS; i++) begin  
 if (r\_desc\_wait\_cycles[i] > cfg\_timeout\_threshold) begin  
 effective\_priority[i] = {PRIORITY\_WIDTH{1'b1}}; // Max priority  
 end else begin  
 effective\_priority[i] = desc\_priority[i];  
 end  
 end  
end

### Example Scenarios

#### Scenario 1: Simple Priority

Requests:  
 CH0: priority=7, waiting  
 CH1: priority=5, waiting  
 CH2: priority=5, waiting  
  
Result: CH0 granted (highest priority)

#### Scenario 2: Round-Robin

Requests (all priority=5):  
 CH1: waiting  
 CH2: waiting  
 CH4: waiting  
  
Last grant: CH4  
Result: CH1 granted (round-robin from CH4+1)

#### Scenario 3: Timeout Boost

Initial:  
 CH0: priority=7, just requested  
 CH3: priority=3, waiting 1000 cycles (> timeout)  
  
Timeout boost: CH3 effective priority = 255  
Result: CH3 granted (timeout boost to max priority)

### Integration Pattern

#### Connecting to Schedulers

// Instantiate arbiter  
channel\_arbiter #(  
 .NUM\_CHANNELS(8)  
) u\_arbiter (  
 .aclk(aclk),  
 .aresetn(aresetn),  
  
 // Descriptor path  
 .desc\_req({ch7\_desc\_req, ch6\_desc\_req, ..., ch0\_desc\_req}),  
 .desc\_priority({ch7\_priority, ch6\_priority, ..., ch0\_priority}),  
 .desc\_grant({ch7\_desc\_grant, ch6\_desc\_grant, ..., ch0\_desc\_grant}),  
  
 // Data read path  
 .datard\_req({ch7\_datard\_valid, ..., ch0\_datard\_valid}),  
 .datard\_priority({ch7\_priority, ..., ch0\_priority}),  
 .datard\_grant({ch7\_datard\_ready, ..., ch0\_datard\_ready}),  
  
 // Data write path  
 .datawr\_req({ch7\_datawr\_valid, ..., ch0\_datawr\_valid}),  
 .datawr\_priority({ch7\_priority, ..., ch0\_priority}),  
 .datawr\_grant({ch7\_datawr\_ready, ..., ch0\_datawr\_ready})  
);

#### Multiplexing Granted Channel

// Descriptor fetch address mux  
always\_comb begin  
 case (desc\_grant\_id)  
 3'd0: desc\_fetch\_addr = ch0\_desc\_addr;  
 3'd1: desc\_fetch\_addr = ch1\_desc\_addr;  
 // ...  
 3'd7: desc\_fetch\_addr = ch7\_desc\_addr;  
 endcase  
end

### Testing

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

**Test Scenarios:** 1. Single channel (no arbitration needed) 2. Two channels, different priorities 3. Multiple channels, same priority (verify round-robin) 4. Timeout boost triggering 5. All 8 channels active simultaneously 6. Priority changes during operation

### Performance

**Arbitration Latency:** 1 cycle (registered output)

**Area Estimate:** ~150 LUTs per arbiter 3 arbiters = ~450 LUTs

### Related Documentation

* **Scheduler:** 02\_scheduler.md - Requesters
* **Top-Level:** 09\_stream\_top.md - Integration

**Last Updated:** 2025-10-17 ## APB Config Specification

**Module:** apb\_config.sv **Location:** rtl/stream\_macro/ **Status:** Placeholder (PeakRDL generation planned)

### Overview

The APB Config module provides the APB slave interface for STREAM configuration and control. It wraps PeakRDL-generated registers and optionally includes clock domain crossing (CDC) logic.

#### Current Status

**Phase 1 (Current):** Manual placeholder implementation **Phase 2 (Future):** PeakRDL-generated registers with wrapper

#### 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\_CTRL | 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\_BURST | RW | 32 | Read burst length config |
| +0x0C | CHx\_WR\_BURST | RW | 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;

#### Ports

**APB Slave Interface:**

input logic pclk;  
input logic presetn;  
  
input logic [ADDR\_WIDTH-1:0] paddr;  
input logic psel;  
input logic penable;  
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 Integration (Future)

#### Generation Workflow

**See:** regs/README.md for complete workflow

1. **Create:** regs/stream\_regs.rdl
2. **Generate:** peakrdl regblock stream\_regs.rdl -o regs/generated/
3. **Update:** apb\_config.sv to instantiate generated registers

#### Wrapper Pattern

// Future: apb\_config.sv becomes wrapper  
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\_hpet/ - Reference implementation
* **Scheduler:** 02\_scheduler.md - Consumer of configuration

**Last Updated:** 2025-10-17 ## MonBus AXIL Group Specification

**Module:** monbus\_axil\_group.sv **Location:** rtl/stream\_macro/ **Source:** Copied from RAPIDS (identical)

### Overview

The MonBus AXIL Group provides monitoring and error reporting for STREAM. It aggregates monitor bus packets from all channels and provides AXIL interfaces for error/interrupt handling and packet logging to memory.

#### Key Features

* **Multiple MonBus inputs:** One per STREAM channel
* **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
* **Identical to RAPIDS:** Proven design, no modifications

### Differences from RAPIDS

**None.** This module is functionally identical to RAPIDS monbus\_axil\_group.sv.

**Only Change:** - Header comment updated to mention STREAM - Functionally unchanged

**Why Identical:** - MonBus protocol standardized across all projects - Error/interrupt handling proven in RAPIDS - AXIL interface patterns reused

### Interface

#### Parameters

parameter int NUM\_CHANNELS = 8; // Number of monitor bus inputs  
parameter int MONBUS\_WIDTH = 64; // Monitor bus packet width  
parameter int AXIL\_ADDR\_WIDTH = 32; // AXIL address width  
parameter int AXIL\_DATA\_WIDTH = 32; // AXIL data width  
parameter int ERROR\_FIFO\_DEPTH = 64; // Error packet FIFO depth

#### Ports

**Clock and Reset:**

input logic aclk;  
input logic aresetn;

**Monitor Bus Inputs (per channel):**

input logic [NUM\_CHANNELS-1:0] ch\_monbus\_valid;  
output logic [NUM\_CHANNELS-1:0] ch\_monbus\_ready;  
input logic [NUM\_CHANNELS-1:0][MONBUS\_WIDTH-1:0]  
 ch\_monbus\_packet;

**AXIL Slave (Error/Interrupt FIFO Read):**

// AR channel  
input logic [AXIL\_ADDR\_WIDTH-1:0] s\_axil\_araddr;  
input logic s\_axil\_arvalid;  
output logic s\_axil\_arready;  
  
// R channel  
output logic [AXIL\_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 [AXIL\_ADDR\_WIDTH-1:0] m\_axil\_awaddr;  
output logic m\_axil\_awvalid;  
input logic m\_axil\_awready;  
  
// W channel  
output logic [AXIL\_DATA\_WIDTH-1:0] m\_axil\_wdata;  
output logic [3:0] 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;

**Configuration:**

input logic [AXIL\_ADDR\_WIDTH-1:0] cfg\_log\_base\_addr; // Base addr for logging  
input logic cfg\_log\_enable; // Enable memory logging  
input logic cfg\_error\_irq\_enable; // Enable error interrupts

### Operation

#### Monitor Packet Flow

Channel MonBus -> Packet Classifier -> [Error FIFO | Log FIFO]  
 | |  
 AXIL Slave AXIL Master  
 (CPU read) (Memory write)

#### 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\_LOW | RO | Error packet [31:0], auto-pop on read |
| 0x04 | ERROR\_PKT\_HIGH | RO | Error packet [63:32] |
| 0x08 | ERROR\_FIFO\_STATUS | 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 #(  
 .NUM\_CHANNELS(8)  
) u\_monbus (  
 .aclk(aclk),  
 .aresetn(aresetn),  
  
 // MonBus inputs from channels  
 .ch\_monbus\_valid({ch7\_mon\_valid, ..., ch0\_mon\_valid}),  
 .ch\_monbus\_ready({ch7\_mon\_ready, ..., ch0\_mon\_ready}),  
 .ch\_monbus\_packet({ch7\_mon\_pkt, ..., ch0\_mon\_pkt}),  
  
 // AXIL slave (CPU access to error FIFO)  
 .s\_axil\_araddr(cpu\_araddr),  
 .s\_axil\_arvalid(cpu\_arvalid),  
 // ... AXIL slave signals  
  
 // AXIL master (log to memory)  
 .m\_axil\_awaddr(log\_awaddr),  
 .m\_axil\_awvalid(log\_awvalid),  
 // ... AXIL master signals  
  
 // Interrupt  
 .irq\_out(stream\_irq),  
  
 // Configuration  
 .cfg\_log\_base\_addr(32'h8000\_0000),  
 .cfg\_log\_enable(1'b1),  
 .cfg\_error\_irq\_enable(1'b1)  
);

### 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

**Reference:** RAPIDS monbus\_axil\_group tests can be reused directly.

### 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

* **RAPIDS Spec:** projects/components/rapids/docs/rapids\_spec/ch02\_blocks/04\_monbus\_axil\_group.md (if available)
* **MonBus Protocol:** rtl/amba/includes/monitor\_pkg.sv
* **Source:** rtl/stream\_macro/monbus\_axil\_group.sv

**Last Updated:** 2025-10-17 ## STREAM Top-Level Integration Specification

**Module:** stream\_top.sv **Location:** rtl/stream\_macro/ **Status:** To be created

### Overview

The STREAM Top-Level module integrates all STREAM components into a complete scatter-gather DMA engine. It provides the external interfaces for APB configuration, AXI memory access, and MonBus monitoring.

#### Key Features

* **8 independent channels** with shared resource arbitration
* **APB slave** for configuration
* **AXI masters** for descriptor fetch, data read, data write
* **AXIL interfaces** for MonBus error/logging
* **MonBus output** for monitoring and debugging
* **Parameterizable** performance modes for AXI engines

### Block Diagram

|-------------------------------------|  
 | STREAM Top |  
 | |  
APB Config ----------------| APB Config |  
 | | |  
 | |----------------------------| |  
 | | 8 Scheduler + Desc Eng | |  
 | | (one per channel) | |  
 | `---|--------------------|---| |  
 | | | |  
 | | | |  
 | |---------| |---------| |  
Descriptor Fetch ----------| | Arbiter | | Arbiter | |  
(AXI Master) | `----|----| `----|----| |  
 | | | |  
Data Read -----------------| AXI Read AXI Write |  
(AXI Master) | Engine Engine |  
 | | | |  
Data Write ----------------| | | |  
(AXI Master) | |-----------------------| |  
 | | Simple SRAM | |  
 | `-----------------------| |  
 | |  
 | |----------------------| |  
MonBus --------------------| | MonBus AXIL Group | |  
AXIL Slave ----------------| | - Error FIFO | |  
AXIL Master ---------------| | - Logging | |  
IRQ -----------------------| | - Interrupt | |  
 | `----------------------| |  
 `-------------------------------------|

### Interface

#### Parameters

parameter int NUM\_CHANNELS = 8; // Fixed at 8  
parameter int ADDR\_WIDTH = 64; // Address bus width  
parameter int DATA\_WIDTH = 512; // Data bus width  
parameter int APB\_ADDR\_WIDTH = 32; // APB address width  
parameter int APB\_DATA\_WIDTH = 32; // APB data width  
parameter int AXIL\_ADDR\_WIDTH = 32; // AXIL address width  
parameter int AXIL\_DATA\_WIDTH = 32; // AXIL data width  
parameter int SRAM\_DEPTH = 1024; // SRAM depth  
  
// Performance modes for AXI engines  
parameter string RD\_PERFORMANCE = "LOW"; // "LOW", "MEDIUM", "HIGH"  
parameter string WR\_PERFORMANCE = "LOW"; // "LOW", "MEDIUM", "HIGH"  
parameter int RD\_MAX\_BURST\_LEN = 8; // Read engine max burst  
parameter int WR\_MAX\_BURST\_LEN = 16; // Write engine max burst

#### Ports

**Clock and Reset:**

input logic aclk;  
input logic aresetn;

**APB Configuration Interface:**

input logic [APB\_ADDR\_WIDTH-1:0] s\_apb\_paddr;  
input logic s\_apb\_psel;  
input logic s\_apb\_penable;  
input logic s\_apb\_pwrite;  
input logic [APB\_DATA\_WIDTH-1:0] s\_apb\_pwdata;  
input logic [3:0] s\_apb\_pstrb;  
output logic s\_apb\_pready;  
output logic [APB\_DATA\_WIDTH-1:0] s\_apb\_prdata;  
output logic s\_apb\_pslverr;

**AXI Master - Descriptor Fetch (256-bit):**

// AR channel  
output logic [ADDR\_WIDTH-1:0] m\_axi\_desc\_araddr;  
output logic [7:0] m\_axi\_desc\_arlen;  
output logic [2:0] m\_axi\_desc\_arsize;  
output logic [1:0] m\_axi\_desc\_arburst;  
output logic m\_axi\_desc\_arvalid;  
input logic m\_axi\_desc\_arready;  
  
// R channel  
input logic [255:0] m\_axi\_desc\_rdata;  
input logic [1:0] m\_axi\_desc\_rresp;  
input logic m\_axi\_desc\_rlast;  
input logic m\_axi\_desc\_rvalid;  
output logic m\_axi\_desc\_rready;

**AXI Master - Data Read (parameterizable width):**

// AR channel  
output logic [ADDR\_WIDTH-1:0] m\_axi\_rd\_araddr;  
output logic [7:0] m\_axi\_rd\_arlen;  
output logic [2:0] m\_axi\_rd\_arsize;  
output logic [1:0] m\_axi\_rd\_arburst;  
output logic m\_axi\_rd\_arvalid;  
input logic m\_axi\_rd\_arready;  
  
// R channel  
input logic [DATA\_WIDTH-1:0] m\_axi\_rd\_rdata;  
input logic [1:0] m\_axi\_rd\_rresp;  
input logic m\_axi\_rd\_rlast;  
input logic m\_axi\_rd\_rvalid;  
output logic m\_axi\_rd\_rready;

**AXI Master - Data Write (parameterizable width):**

// AW channel  
output logic [ADDR\_WIDTH-1:0] m\_axi\_wr\_awaddr;  
output logic [7:0] m\_axi\_wr\_awlen;  
output logic [2:0] m\_axi\_wr\_awsize;  
output logic [1:0] m\_axi\_wr\_awburst;  
output logic m\_axi\_wr\_awvalid;  
input logic m\_axi\_wr\_awready;  
  
// W channel  
output logic [DATA\_WIDTH-1:0] m\_axi\_wr\_wdata;  
output logic [DATA\_WIDTH/8-1:0] m\_axi\_wr\_wstrb;  
output logic m\_axi\_wr\_wlast;  
output logic m\_axi\_wr\_wvalid;  
input logic m\_axi\_wr\_wready;  
  
// B channel  
input logic [1:0] m\_axi\_wr\_bresp;  
input logic m\_axi\_wr\_bvalid;  
output logic m\_axi\_wr\_bready;

**AXIL Slave (MonBus Error/Interrupt Access):**

// AR channel  
input logic [AXIL\_ADDR\_WIDTH-1:0] s\_axil\_araddr;  
input logic s\_axil\_arvalid;  
output logic s\_axil\_arready;  
  
// R channel  
output logic [AXIL\_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 (MonBus Packet Logging to Memory):**

// AW channel  
output logic [AXIL\_ADDR\_WIDTH-1:0] m\_axil\_awaddr;  
output logic m\_axil\_awvalid;  
input logic m\_axil\_awready;  
  
// W channel  
output logic [AXIL\_DATA\_WIDTH-1:0] m\_axil\_wdata;  
output logic [3:0] 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;

**MonBus Output:**

output logic monbus\_valid;  
input logic monbus\_ready;  
output logic [63:0] monbus\_packet;

**Interrupt:**

output logic irq\_out;

### Internal Instances

#### Per-Channel Blocks (8 instances)

// Channel 0-7: Scheduler + Descriptor Engine  
for (genvar ch = 0; ch < NUM\_CHANNELS; ch++) begin : gen\_channels  
  
 // Descriptor Engine  
 descriptor\_engine #(  
 .ADDR\_WIDTH(ADDR\_WIDTH),  
 .DATA\_WIDTH(256)  
 ) u\_desc\_engine (  
 .aclk(aclk),  
 .aresetn(aresetn),  
 .desc\_req\_valid(ch\_desc\_req[ch]),  
 .desc\_req\_ready(ch\_desc\_req\_ready[ch]),  
 .desc\_req\_addr(ch\_desc\_addr[ch]),  
 .desc\_valid(ch\_desc\_valid[ch]),  
 .desc\_ready(ch\_desc\_ready[ch]),  
 .desc\_packet(ch\_desc\_packet[ch]),  
 // ... AXI and MonBus  
 );  
  
 // Scheduler  
 scheduler #(  
 .CHANNEL\_ID(ch),  
 .ADDR\_WIDTH(ADDR\_WIDTH),  
 .DATA\_WIDTH(DATA\_WIDTH)  
 ) u\_scheduler (  
 .aclk(aclk),  
 .aresetn(aresetn),  
 .cfg\_enable(ch\_enable[ch]),  
 .desc\_valid(ch\_desc\_valid[ch]),  
 .desc\_ready(ch\_desc\_ready[ch]),  
 .desc\_packet(ch\_desc\_packet[ch]),  
 .datard\_valid(ch\_datard\_valid[ch]),  
 .datard\_ready(ch\_datard\_ready[ch]),  
 // ... more signals  
 );  
  
end

#### Shared Resources

// Channel Arbiter  
channel\_arbiter #(  
 .NUM\_CHANNELS(NUM\_CHANNELS)  
) u\_arbiter (  
 .aclk(aclk),  
 .aresetn(aresetn),  
 .desc\_req(ch\_desc\_req),  
 .desc\_priority(ch\_desc\_priority),  
 .desc\_grant(ch\_desc\_grant),  
 .datard\_req(ch\_datard\_valid),  
 .datard\_grant(ch\_datard\_ready),  
 .datawr\_req(ch\_datawr\_valid),  
 .datawr\_grant(ch\_datawr\_ready)  
);  
  
// AXI Read Engine  
axi\_read\_engine #(  
 .PERFORMANCE(RD\_PERFORMANCE),  
 .MAX\_BURST\_LEN(RD\_MAX\_BURST\_LEN),  
 .ADDR\_WIDTH(ADDR\_WIDTH),  
 .DATA\_WIDTH(DATA\_WIDTH)  
) u\_rd\_engine (  
 .aclk(aclk),  
 .aresetn(aresetn),  
 .datard\_valid(datard\_valid\_mux), // From arbiter mux  
 .datard\_ready(datard\_ready\_mux),  
 .datard\_addr(datard\_addr\_mux),  
 .m\_axi\_araddr(m\_axi\_rd\_araddr),  
 // ... more signals  
);  
  
// AXI Write Engine  
axi\_write\_engine #(  
 .PERFORMANCE(WR\_PERFORMANCE),  
 .MAX\_BURST\_LEN(WR\_MAX\_BURST\_LEN),  
 .ADDR\_WIDTH(ADDR\_WIDTH),  
 .DATA\_WIDTH(DATA\_WIDTH)  
) u\_wr\_engine (  
 .aclk(aclk),  
 .aresetn(aresetn),  
 .datawr\_valid(datawr\_valid\_mux), // From arbiter mux  
 .datawr\_ready(datawr\_ready\_mux),  
 .datawr\_addr(datawr\_addr\_mux),  
 .m\_axi\_awaddr(m\_axi\_wr\_awaddr),  
 // ... more signals  
);  
  
// Simple SRAM  
simple\_sram #(  
 .DATA\_WIDTH(DATA\_WIDTH),  
 .ADDR\_WIDTH($clog2(SRAM\_DEPTH))  
) u\_sram (  
 .aclk(aclk),  
 .aresetn(aresetn),  
 .wr\_en(sram\_wr\_en),  
 .wr\_addr(sram\_wr\_addr),  
 .wr\_data(sram\_wr\_data),  
 .rd\_en(sram\_rd\_en),  
 .rd\_addr(sram\_rd\_addr),  
 .rd\_data(sram\_rd\_data)  
);  
  
// MonBus AXIL Group  
monbus\_axil\_group #(  
 .NUM\_CHANNELS(NUM\_CHANNELS)  
) u\_monbus (  
 .aclk(aclk),  
 .aresetn(aresetn),  
 .ch\_monbus\_valid(ch\_monbus\_valid),  
 .ch\_monbus\_ready(ch\_monbus\_ready),  
 .ch\_monbus\_packet(ch\_monbus\_packet),  
 .s\_axil\_araddr(s\_axil\_araddr),  
 .m\_axil\_awaddr(m\_axil\_awaddr),  
 .irq\_out(irq\_out),  
 // ... more signals  
);  
  
// APB Config  
apb\_config #(  
 .NUM\_CHANNELS(NUM\_CHANNELS)  
) u\_config (  
 .pclk(aclk),  
 .presetn(aresetn),  
 .paddr(s\_apb\_paddr),  
 .psel(s\_apb\_psel),  
 .ch\_enable(ch\_enable),  
 .ch\_desc\_addr(ch\_desc\_addr),  
 .ch\_read\_burst\_len(ch\_read\_burst\_len),  
 .ch\_write\_burst\_len(ch\_write\_burst\_len),  
 // ... more signals  
);

### Resource Multiplexing

#### Descriptor Fetch Arbiter

// Multiplex descriptor requests to single AXI master  
always\_comb begin  
 case (desc\_grant\_id)  
 3'd0: m\_axi\_desc\_araddr = ch0\_desc\_araddr;  
 3'd1: m\_axi\_desc\_araddr = ch1\_desc\_araddr;  
 // ... all 8 channels  
 endcase  
end  
  
// Demultiplex descriptor responses  
assign ch0\_desc\_rvalid = m\_axi\_desc\_rvalid && (desc\_grant\_id == 3'd0);  
assign ch1\_desc\_rvalid = m\_axi\_desc\_rvalid && (desc\_grant\_id == 3'd1);  
// ... all 8 channels

#### Data Read/Write Arbiters

Similar multiplexing for datard\_\* and datawr\_\* interfaces.

### Configuration Example

**Software initialization:**

// 1. Configure global settings  
write\_apb(ADDR\_GLOBAL\_CTRL, GLOBAL\_ENABLE);  
  
// 2. Configure channel 0  
write\_apb(ADDR\_CH0\_RD\_BURST, 8); // 8-beat read bursts  
write\_apb(ADDR\_CH0\_WR\_BURST, 16); // 16-beat write bursts  
  
// 3. Kick off transfer (write descriptor address)  
write\_apb(ADDR\_CH0\_CTRL, 0x1000\_0000);  
  
// 4. (Optional) Configure multiple channels  
write\_apb(ADDR\_CH1\_CTRL, 0x2000\_0000);  
write\_apb(ADDR\_CH2\_CTRL, 0x3000\_0000);

### Performance Modes

**Example configurations:**

#### Tutorial Mode (Low Performance)

stream\_top #(  
 .RD\_PERFORMANCE("LOW"),  
 .WR\_PERFORMANCE("LOW"),  
 .RD\_MAX\_BURST\_LEN(8),  
 .WR\_MAX\_BURST\_LEN(16)  
) u\_stream (...);

**Characteristics:** Simple, clear, educational

#### Production Mode (High Performance)

stream\_top #(  
 .RD\_PERFORMANCE("HIGH"),  
 .WR\_PERFORMANCE("HIGH"),  
 .RD\_MAX\_BURST\_LEN(256),  
 .WR\_MAX\_BURST\_LEN(256)  
) u\_stream (...);

**Characteristics:** Maximum throughput, pipelined

### Testing

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

**Test Scenarios:** 1. Single channel transfer (end-to-end) 2. Multi-channel concurrent transfers 3. Chained descriptors 4. Error handling and recovery 5. Performance validation (different modes) 6. MonBus packet generation 7. Interrupt handling

### Area Estimates

| Component | Instances | Area/Instance | Total |
| --- | --- | --- | --- |
| Descriptor Engine | 8 | ~300 LUTs | ~2400 LUTs |
| Scheduler | 8 | ~400 LUTs | ~3200 LUTs |
| AXI Read Engine (Low) | 1 | ~250 LUTs | ~250 LUTs |
| AXI Write Engine (Low) | 1 | ~250 LUTs | ~250 LUTs |
| Simple SRAM | 1 | 1024 64B | 64 KB |
| Channel Arbiter | 3 | ~150 LUTs | ~450 LUTs |
| APB Config | 1 | ~350 LUTs | ~350 LUTs |
| MonBus AXIL Group | 1 | ~1000 LUTs | ~1000 LUTs |
| **Total (Low Perf)** | - | - | **~7900 LUTs + 64KB SRAM** |

**High Performance:** ~12000 LUTs + 64KB SRAM (due to AXI engine pipelining)

### Related Documentation

* **All Components:** 00\_index.md - Specification index
* **RAPIDS Comparison:** ../../ARCHITECTURAL\_NOTES.md
* **Source:** rtl/stream\_macro/stream\_top.sv (to be created)

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

## 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] Simplified from RAPIDS |
| scheduler | stream\_fub/scheduler.sv | Transfer coordinator | ~400 | [Done] Created (corrected) |
| axi\_read\_engine | stream\_fub/axi\_read\_engine.sv | AXI read master | ~250 | [Pending] To be created |
| axi\_write\_engine | stream\_fub/axi\_write\_engine.sv | AXI write master | ~250 | [Pending] To be created |
| sram\_controller | stream\_fub/sram\_controller.sv | Per-channel buffer management | ~400 | [Pending] To be created |
| simple\_sram | stream\_fub/simple\_sram.sv | Dual-port SRAM primitive | ~150 | [Done] Copied from RAPIDS |

### Integration Blocks (MAC)

| Module | File | Purpose | Lines | Status |
| --- | --- | --- | --- | --- |
| channel\_arbiter | stream\_macro/channel\_arbiter.sv | Priority arbitration | ~200 | [Pending] To be created |
| apb\_config | stream\_macro/apb\_config.sv | Config wrapper | ~350 | [Done] Placeholder |
| monbus\_axil\_group | stream\_macro/monbus\_axil\_group.sv | MonBus + AXIL | ~800 | [Done] Copied from RAPIDS |
| 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:

### Low Performance Mode

* **Target:** Area-optimized, low throughput
* **Features:** Minimal logic, single outstanding transaction
* **Use Case:** Tutorial examples, area-constrained designs
* **Area:** ~250 LUTs per engine

### Medium Performance Mode

* **Target:** Balanced area/performance
* **Features:** Basic pipelining, 2-4 outstanding transactions
* **Use Case:** Typical FPGA implementations
* **Area:** ~400 LUTs per engine

### High Performance Mode

* **Target:** Maximum throughput
* **Features:** Full pipelining, 8+ outstanding transactions
* **Use Case:** High-bandwidth ASIC implementations
* **Area:** ~600 LUTs per engine

## Clock and Reset Summary

### Clock Domains

| Clock | Frequency | Usage |
| --- | --- | --- |
| aclk | 100-500 MHz | Primary - all STREAM logic, AXI/AXIL interfaces |
| 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](ch01_overview/03_clocks_and_reset.md) 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 implementations |
| 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 |
| MonBus AXIL Group | 1 | ~1,000 LUTs | ~1,000 LUTs |
| **Total** | - | - | **~9,500 LUTs + 64KB** |

## Related Documentation

* [**PRD.md**](../../PRD.md) - Product requirements and overview
* [**ARCHITECTURAL\_NOTES.md**](../ARCHITECTURAL_NOTES.md) - Critical design decisions
* [**CLAUDE.md**](../../CLAUDE.md) - AI development guide
* [**Register Generation**](../../regs/README.md) - 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