STREAM Specification Index

Generated by md_to_docx.py

today

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

```python
## 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`
- **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_schedu`
- **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 back-pressure 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 back-pressure 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_g` (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

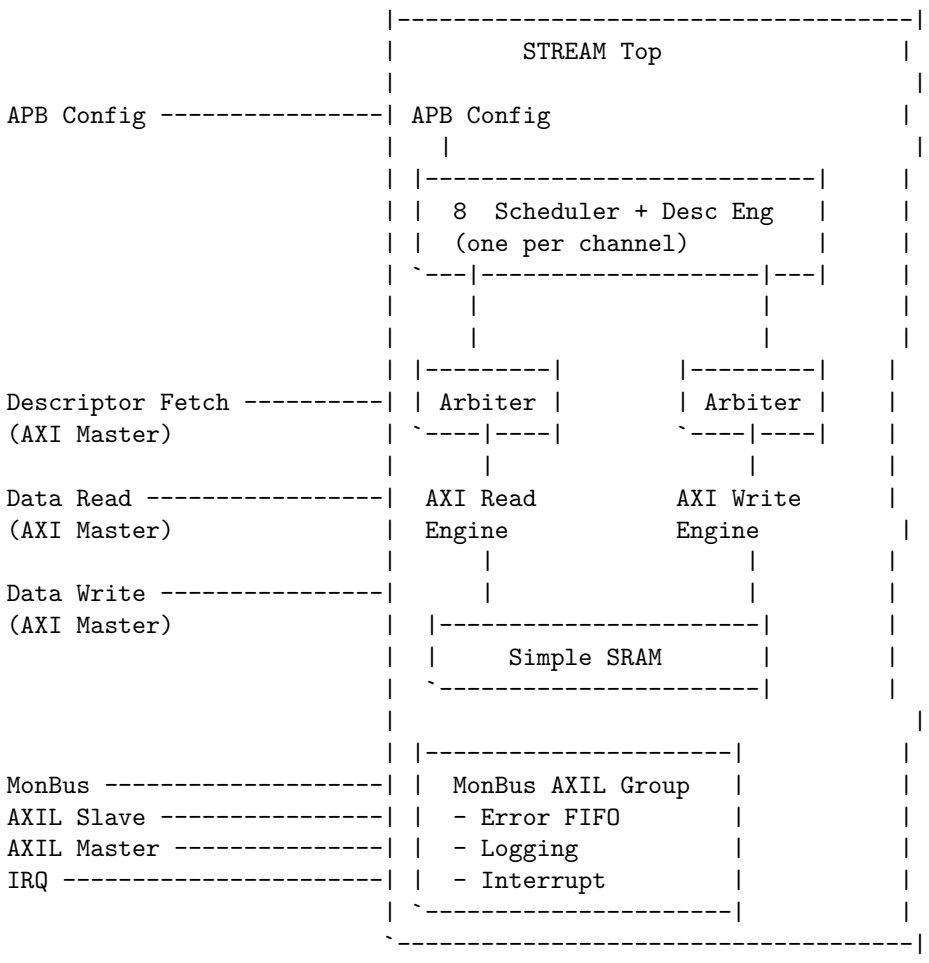
Overview

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



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	Channel 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	Page/control 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	CSR registers	~350	[Done] Placeholder
monbus_axil_group	stream_macro/monbus_axil_group.sv	MonBus → AXIL packetizer	~300	[Done] Copied from RAPIDS
stream_top	stream_macro/stream_top.sv	Full-stream integration	~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 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

Component	Instances	Area/Instance	Total
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** - Product requirements and overview
- **ARCHITECTURAL_NOTES.md** - Critical design decisions
- **CLAUDE.md** - AI development guide
- **Register Generation** - PeakRDL workflow

Specification Conventions

Signal Naming

- **Clock:** `aclk`, `pclk`
- **Reset:** `aresetn`, `presetn` (active-low)
- **Valid/Ready:** Standard AXI/custom handshake
- **Registers:** `r_` prefix (e.g., `r_state`, `r_counter`)
- **Wires:** `w_` prefix (e.g., `w_next_state`, `w_grant`)

Parameter Naming

- **Uppercase:** `NUM_CHANNELS`, `DATA_WIDTH`, `ADDR_WIDTH`
- **String parameters:** `PERFORMANCE` ("LOW", "MEDIUM", "HIGH")

State Machine Naming

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

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

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