Table of Contents

1 Rapids Beats Mas Index

Generated: 2026-01-17

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

---

2 Document Information

This document describes the RAPIDS “Beats” architecture, a Phase 1 implementation of the Rapid AXI Programmable In-band Descriptor System. The beats architecture provides network-to-memory and memory-to-network data transfer capabilities using descriptor-based DMA with 8-channel support. It emphasizes “beat-level” tracking for precise flow control and latency management.

2.1 References

2.1.1 Related Documents

Table 0.1: Related Documents and Specifications

Source	Title	Version
RTL Design Sherpa	RAPIDS Product Requirements Document	1.0
RTL Design Sherpa	RAPIDS Original Specification	0.25
RTL Design Sherpa	STREAM MAS (Reference Architecture)	0.90

Source	Title	Version
ARM	AMBA AXI and ACE Protocol Specification	IHI0022H
ARM	AMBA AXI-Stream Protocol Specification	IHI0051A

2.2 Terminology

AXIS

AXI-Stream. ARM's streaming protocol for high-bandwidth, low-latency data transfer without addressing.

AXI4

Version 4 of the AXI protocol, supporting burst transfers up to 256 beats.

Beat

A single data transfer within an AXI burst. For RAPIDS with 512-bit data width, one beat = 64 bytes.

Beats Architecture

The Phase 1 RAPIDS implementation that tracks transfers at beat granularity for precise flow control.

Burst

A group of consecutive data transfers (beats) initiated by a single address phase.

Channel

One of 8 independent DMA channels in RAPIDS. Each channel has its own descriptor chain and SRAM buffer allocation.

Descriptor

A 256-bit data structure containing transfer parameters: address, length, channel ID, and control flags.

Descriptor Chain

A linked list of descriptors where each descriptor points to the next, enabling scatter-gather operations.

DMA

Direct Memory Access. Hardware-controlled data movement between memory locations without CPU intervention.

FUB

Functional Unit Block. A self-contained RTL module with defined interfaces and testbench.

Latency Bridge

A module that provides buffering to compensate for pipeline latency between alloc/drain controllers.

MAC

Macro. An integration-level block that instantiates and connects multiple FUBs.

MonBus

Monitor Bus. A 64-bit internal bus for performance monitoring and debug event reporting.

RAPIDS

Rapid AXI Programmable In-band Descriptor System. A DMA accelerator for network-to-memory transfers.

Sink Path

The data path from network (AXIS slave) to system memory (AXI write master).

Source Path

The data path from system memory (AXI read master) to network (AXIS master).

SRAM

Static Random Access Memory. Used for internal data buffering between network and memory interfaces.

Virtual FIFO

A tracking mechanism (alloc_ctrl/drain_ctrl) that manages SRAM space without moving actual data.

2.3 "Beats" Architecture Concept

The "beats" architecture name reflects the key design decision to track all transfers at **beat granularity** (data-width units) rather than byte or chunk granularity:

1. **Beat-level Allocation:** Space is allocated in beats (e.g., 64-byte units for 512-bit datapath)
2. **Beat-level Drain:** Data availability is tracked in beats
3. **Beat-level Latency:** The latency bridge compensates for pipeline delays in beat counts
4. **Beat-level Completion:** AXI engines report completion in beats

This simplifies the math and aligns naturally with AXI burst semantics where each beat represents one data-width transfer.

2.4 Revision History

Revisions follow the convention x.y, where x is the major version and y is the minor version.

Table 0.2: RAPIDS Beats MAS Document Revision History

Rev	Date	Author	Notes
0.25	2025-01-10	seang	Initial RAPIDS Beats MAS structure

Last Updated: 2025-01-10

---

3 Architecture Overview

Module: `rapids_core_beats.sv` **Location:** `projects/components/rapids/rtl/macro_beats/` **Status:** Implemented **Last Updated:** 2025-01-10

3.1 Overview

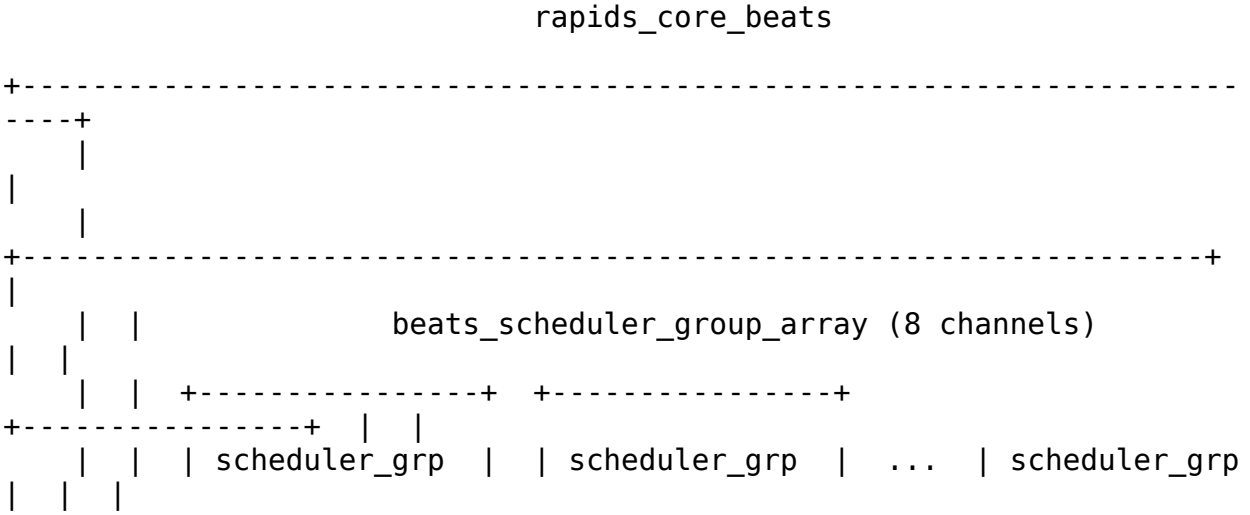
The RAPIDS "Beats" architecture is a Phase 1 implementation providing network-to-memory and memory-to-network data transfer capabilities. The name "beats" reflects the design decision to track all transfers at beat granularity (data-width units) for simplified flow control.

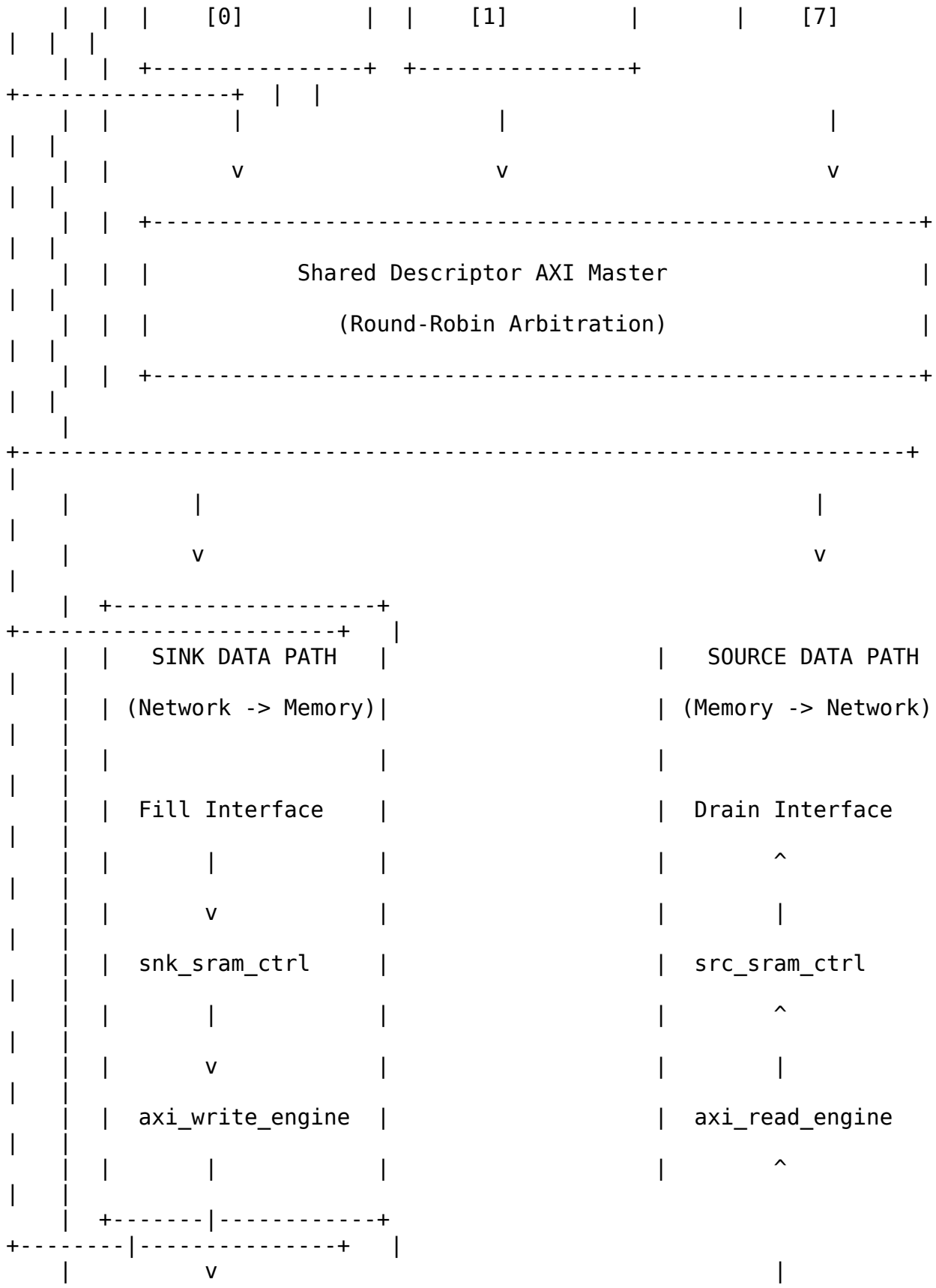
3.1.1 Key Features

- **8 Independent Channels:** Each channel has its own descriptor chain and SRAM allocation
- **Separate Data Paths:** Sink (network-to-memory) and source (memory-to-network) paths
- **Beat-Level Tracking:** All flow control operates at beat granularity
- **SRAM Buffering:** Separate sink and source SRAM buffers for data staging
- **MonBus Integration:** Comprehensive monitoring for all subsystems
- **Streaming Pipelines:** No FSM in data engines - pure streaming for maximum throughput

3.1.2 Block Diagram

3.1.3 Figure 1.1.1: RAPIDS Beats Architecture Block Diagram




[0]
[1]
[7]
Shared Descriptor AXI Master
(Round-Robin Arbitration)
SINK DATA PATH
(Network -> Memory)
SOURCE DATA PATH
(Memory -> Network)
Fill Interface
Drain Interface
snk_sram_ctrl
src_sram_ctrl
axi_write_engine
axi_read_engine

```
|

+----------|-----------------------------------------|--------------
--+
               |                                         |
               v                                         |
        AXI Write Master                          AXI Read Master
        (to System Memory)                        (from System
Memory)
```

Source: 01_architecture_block.mmd

3.2 Data Flow Overview

3.2.1 Sink Path (Network to Memory)

The sink path receives data from an external source (via Fill interface) and writes it to system memory:

3.2.2 Figure 1.1.2: Sink Path Data Flow

```
1. External Fill Valid
        |
        v
2. beats_alloc_ctrl (Space Allocation)
        |
        v
3. snk_sram_controller_unit (SRAM Write)
        |
        v
4. beats_drain_ctrl (Data Available)
        |
        v
5. axi_write_engine (AXI Burst Write)
        |
        v
6. System Memory
```

Source: 01_sink_path_flow.mmd

3.2.3 Source Path (Memory to Network)

The source path reads data from system memory and sends it to an external destination (via Drain interface):

3.2.4 Figure 1.1.3: Source Path Data Flow

```
1. Scheduler Request
        |
```

```
        v
2. beats_alloc_ctrl (Space Allocation)
        |
        v
3. axi_read_engine (AXI Burst Read)
        |
        v
4. src_sram_controller_unit (SRAM Write)
        |
        v
5. beats_drain_ctrl (Data Available)
        |
        v
6. External Drain Ready
```

Source: 01_source_path_flow.mmd

3.3 Key Architectural Decisions

3.3.1 Beat-Level Tracking

All flow control uses beat granularity:

Table 1.1.1: Beat Granularity Examples

Operation	Granularity	Example (512-bit DW)
Space allocation	Beats	Request 8 beats = 512 bytes
Data availability	Beats	16 beats ready = 1024 bytes
AXI burst length	Beats	ARLEN=15 = 16 beats
Latency compensation	Beats	2-beat pipeline delay

3.3.2 Concurrent Read/Write

To prevent deadlock with large transfers:

```
Example: 100MB transfer with 2KB SRAM buffer

Sequential operation (WRONG):
1. Read 100MB -> DEADLOCK at 2KB (SRAM full, can't complete read)
```

```
Concurrent operation (CORRECT):
1. Read starts filling SRAM -> SRAM becomes full (2KB)
2. Read pauses (natural backpressure)
3. Write drains SRAM -> SRAM has free space
4. Read resumes -> Both continue until 100MB complete
```

3.3.3 Virtual FIFOs (Alloc/Drain)

The alloc_ctrl and drain_ctrl modules are “virtual FIFOs” that track space/data without storing actual data:

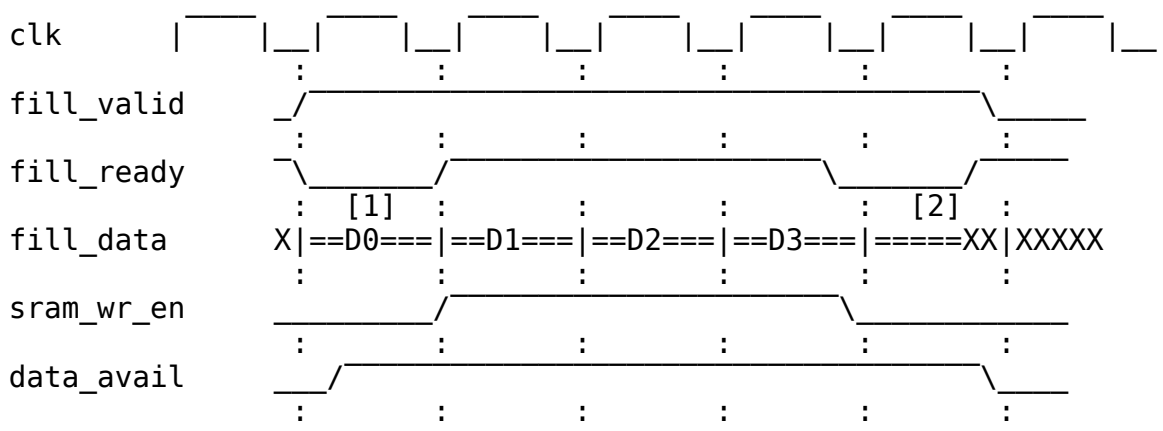
- **beats_alloc_ctrl:** Tracks allocated space (write pointer advances on allocation, read pointer on actual write)
- **beats_drain_ctrl:** Tracks available data (write pointer advances on data arrival, read pointer on drain)

3.4 Module Hierarchy

```
rapids_core_beats
├── beats_scheduler_group_array
│   ├── beats_scheduler_group [0..7]
│   │   ├── descriptor_engine
│   │   └── scheduler
│   └── Shared Descriptor AXI Master
│
├── sink_data_path
│   ├── snk_sram_controller
│   │   └── snk_sram_controller_unit [0..7]
│   │       ├── beats_alloc_ctrl
│   │       ├── simple_sram
│   │       └── beats_drain_ctrl
│   └── axi_write_engine
│       └── beats_latency_bridge
│
└── source_data_path
    ├── axi_read_engine
    │   └── beats_latency_bridge
    └── src_sram_controller
        └── src_sram_controller_unit [0..7]
            ├── beats_alloc_ctrl
            ├── simple_sram
            └── beats_drain_ctrl
```

3.5 Timing Diagram: Basic Transfer

3.5.1 Figure 1.1.4: Basic Sink Path Transfer Timing




```
[1] = fill_ready deasserted (SRAM full or allocation pending)
[2] = fill_ready deasserted (end of burst)
```

TODO: Replace with simulation-generated waveform showing actual signals

---

3.6 Related Documentation

- **Top-Level Port List** - Complete port specification
- **Clocks and Reset** - Timing requirements
- **Beats Scheduler Group** - Scheduler integration
- **Sink Data Path** - Sink path details

Last Updated: 2025-01-10

---

4 Top-Level Port List

Module: `rapids_core_beats.sv` **Location:** `projects/components/rapids/rtl/macro_beats/` **Status:** Implemented **Last Updated:** 2025-01-10

4.1 Parameters

4.1.1 Primary Parameters

```
parameter int NUM_CHANNELS = 8;                // Number of DMA
channels
parameter int CHAN_WIDTH = $clog2(NUM_CHANNELS);
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 SRAM_DEPTH = 512;                // SRAM depth per
channel
parameter int SEG_COUNT_WIDTH = $clog2(SRAM_DEPTH) + 1;
parameter int PIPELINE = 0;                    // Pipeline stages
parameter int AR_MAX_OUTSTANDING = 8;          // Max outstanding AR
transactions
parameter int AW_MAX_OUTSTANDING = 8;          // Max outstanding AW
transactions
parameter int W_PHASE_FIFO_DEPTH = 64;         // Write data FIFO depth
parameter int B_PHASE_FIFO_DEPTH = 16;         // Write response FIFO
depth
```

: Table 1.2.1: Primary Parameters

4.1.2 Monitor Bus Parameters

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

: Table 1.2.2: Monitor Bus Parameters

4.2 Clock and Reset

Table 1.2.3: Clock and Reset Signals

Signal	Direction	Width	Description
`clk`	input	1	System clock (100-500 MHz)
`rst_n`	input	1	Active-low asynchronous

Signal	Direction	Width	Description
			reset

4.3 APB Programming Interface

Table 1.2.4: APB Programming Interface

Signal	Direction	Width	Description
`apb_valid`	input	NC	Per-channel kick-off valid
`apb_ready`	output	NC	Per-channel ready
`apb_addr`	input	NC x AW	Per-channel descriptor start address

4.4 Configuration Interface

4.4.1 Per-Channel Configuration

Table 1.2.5: Per-Channel Configuration

Signal	Direction	Width	Description
`cfg_channel_enable`	input	NC	Enable each channel
`cfg_channel_reset`	input	NC	Per-channel soft reset

4.4.2 Scheduler Configuration

Table 1.2.6: Scheduler Configuration

Signal	Direction	Width	Description
`cfg_sched_enable`	input	1	Global scheduler enable
`cfg_sched_timeout_cycles`	input	16	Timeout threshold in cycles
`cfg_sched_timeout_enable`	input	1	Enable timeout detection
`cfg_sched_err_enable`	input	1	Enable error reporting

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

4.4.3 Descriptor Engine Configuration

Table 1.2.7: Descriptor Engine Configuration

Signal	Direction	Width	Description
`cfg_desceng_enable`	input	1	Global descriptor engine enable
`cfg_desceng_prefetch`	input	1	Enable descriptor prefetching
`cfg_desceng_fifo_thresh`	input	4	Prefetch FIFO threshold
`cfg_desceng_addr0_base`	input	AW	Address range 0 base
`cfg_desceng_addr0_limit`	input	AW	Address range 0 limit
`cfg_desceng_addr1_base`	input	AW	Address range 1 base
`cfg_desceng_addr1_limit`	input	AW	Address range 1 limit

4.4.4 AXI Transfer Configuration

Table 1.2.8: AXI Transfer Configuration

Signal	Direction	Width	Description
`cfg_axi_rd_xfer_beats`	input	8	AXI read burst length (beats)
`cfg_axi_wr_xfer_beats`	input	8	AXI write burst length (beats)

4.5 Status Interface

Table 1.2.9: Status Interface

Signal	Direction	Width	Description
system_idle	output	1	All components idle
descriptor_engine_idle	output	NC	Per-channel descriptor engine idle
scheduler_idle	output	NC	Per-channel scheduler idle
scheduler_state	output	NC x 7	Per-channel FSM state (one-hot)
sched_error	output	NC	Per-channel error flag

4.6 Sink Path Interface (Network to Memory)

4.6.1 Fill Interface (External to SRAM)

Table 1.2.10: Sink Fill Interface

Signal	Direction	Width	Description
snk_fill_alloc_req	input	1	Allocation request
snk_fill_alloc_size	input	8	Size in beats to allocate
snk_fill_alloc_id	input	CIW	Channel ID for allocation
snk_fill_space_free	output	NC x SCW	Available space per channel
snk_fill_valid	input	1	Fill data valid
snk_fill_ready	output	1	Ready to accept fill data
snk_fill_data	input	DW	Fill data
snk_fill_last	input	1	Last beat of fill burst
snk_fill_id	input	CIW	Fill channel ID

4.6.2 Sink AXI Write Master

Table 1.2.11: Sink AXI Write Master Interface

Signal	Direction	Width	Description
`m_snk_axi_awid`	output	IW	Write address ID
`m_snk_axi_awaddr`	output	AW	Write address
`m_snk_axi_awlen`	output	8	Burst length (beats - 1)
`m_snk_axi_awsize`	output	3	Burst size
`m_snk_axi_awburst`	output	2	Burst type
`m_snk_axi_awvalid`	output	1	Write address valid
`m_snk_axi_awready`	input	1	Write address ready
`m_snk_axi_wdata`	output	DW	Write data
`m_snk_axi_wstrb`	output	DW/8	Write strobes
`m_snk_axi_wlast`	output	1	Last write beat
`m_snk_axi_wvalid`	output	1	Write data valid
`m_snk_axi_wready`	input	1	Write data ready
`m_snk_axi_bid`	input	IW	Response ID
`m_snk_axi_bresp`	input	2	Write response
`m_snk_axi_bvalid`	input	1	Response valid
`m_snk_axi_bready`	output	1	Response ready

4.7 Source Path Interface (Memory to Network)

4.7.1 Source AXI Read Master

Table 1.2.12: Source AXI Read Master Interface

Signal	Direction	Width	Description
`m_src_axi_arid`	output	IW	Read address ID
`m_src_axi_araddr`	output	AW	Read address
`m_src_axi_arlen`	output	8	Burst length (beats - 1)
`m_src_axi_arsize`	output	3	Burst size
`m_src_axi_arburst`	output	2	Burst type
`m_src_axi_arvalid`	output	1	Read address valid
`m_src_axi_arready`	input	1	Read address ready
`m_src_axi_rid`	input	IW	Read data ID
`m_src_axi_rdata`	input	DW	Read data
`m_src_axi_rresp`	input	2	Read response
`m_src_axi_rlast`	input	1	Last read beat
`m_src_axi_rvalid`	input	1	Read data valid
`m_src_axi_rready`	output	1	Read data ready

4.7.2 Drain Interface (SRAM to External)

Table 1.2.13: Source Drain Interface

Signal	Direction	Width	Description
`src_drain_valid`	output	1	Drain data valid
`src_drain_ready`	input	1	External ready for drain

Signal	Direction	Width	Description
src_drain_data	output	DW	Drain data
src_drain_last	output	1	Last beat of drain burst
src_drain_id	output	CIW	Drain channel ID

4.8 Descriptor AXI Master

Table 1.2.14: Descriptor AXI Master Interface

Signal	Direction	Width	Description
m_desc_axi_arid	output	IW	Read address ID
m_desc_axi_araddr	output	AW	Read address
m_desc_axi_arlen	output	8	Burst length
m_desc_axi_arsize	output	3	Burst size
m_desc_axi_arburst	output	2	Burst type
m_desc_axi_arvalid	output	1	Read address valid
m_desc_axi_arready	input	1	Read address ready
m_desc_axi_rid	input	IW	Read data ID
m_desc_axi_rdata	input	256	Descriptor data (256-bit)
m_desc_axi_rresp	input	2	Read response
m_desc_axi_rlast	input	1	Last read beat
m_desc_axi_rvalid	input	1	Read data valid
m_desc_axi_rready	output	1	Read data ready

4.9 MonBus Interface

Table 1.2.15: MonBus Interface

Signal	Direction	Width	Description
`monbus_valid`	output	1	Monitor packet valid
`monbus_ready`	input	1	Monitor consumer ready
`monbus_data`	output	64	Monitor packet data

Last Updated: 2025-01-10

---

5 Clocks and Reset

Module: `rapids_core_beats.sv` **Location:** `projects/components/rapids/rtl/macro_beats/` **Status:** Implemented **Last Updated:** 2025-01-10

5.1 Clock Domain

RAPIDS Beats operates in a single clock domain for simplified design and maximum throughput.

5.1.1 Clock Specifications

Table 1.3.1: Clock Specifications

Parameter	Specification
Clock Signal	`clk` (also `axi_aclk` in some

Parameter	Specification
	submodules)
Frequency Range	100 - 500 MHz
Duty Cycle	45% - 55%
Clock Jitter	< 100 ps peak-to-peak

5.1.2 Clock Distribution

```
                    clk (system clock)
                          |
    +---------------------+---------------------+
    |                     |                     |
    v                     v                     v
beats_scheduler    sink_data_path    source_data_path
_group_array            |                     |
    |               +----+----+          +----+----+
    |               |         |          |         |
    v               v         v          v         v
descriptor    snk_sram   axi_write  axi_read  src_sram
_engine       _ctrl      _engine    _engine   _ctrl
```

5.2 Reset Specification

RAPIDS uses asynchronous assert, synchronous deassert reset methodology.

5.2.1 Reset Signal

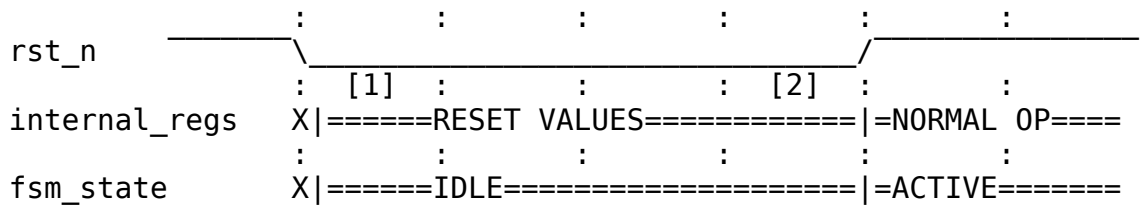
Table 1.3.2: Reset Specifications

Parameter	Specification
Reset Signal	`rst_n` (also `axi_aresetn` in some submodules)
Polarity	Active-low
Assert	Asynchronous (immediate)
Deassert	Synchronous (on rising clock edge)
Minimum Duration	10 clock cycles

5.2.2 Reset Timing Diagram

5.2.3 Figure 1.3.1: Reset Timing

```
    clk      |‾‾‾‾|__|‾‾‾‾|__|‾‾‾‾|__|‾‾‾‾|__|‾‾‾‾|__|‾‾‾‾|__|‾‾‾‾|__|‾‾‾‾|__
```



TODO: Replace with simulation-generated waveform

5.3 Reset Behavior

5.3.1 On Reset Assert

All modules initialize to known states:

Table 1.3.3: Reset States

Component	Reset State
Scheduler FSM	IDLE
Descriptor Engine	IDLE, FIFOs empty
AXI Read Engine	Idle, no outstanding transactions
AXI Write Engine	Idle, no outstanding transactions
SRAM Pointers	All zeros
Alloc/Drain Controllers	Empty, full space available
MonBus	No packets pending

5.3.2 Reset Sequence

```
1. External reset asserted (rst_n = 0)
   - All flip-flops async reset
   - All outputs driven to safe states
   - No AXI transactions

2. System stabilization (minimum 10 cycles)
   - Clock running and stable
   - Configuration registers programmed (optional)

3. Reset deasserted (rst_n = 1, sync to clk rising edge)
```

```
   - FSMs begin operation
   - Ready to accept descriptors/data
```

5.4 Configuration During Reset

Some configuration signals MUST be stable before reset deassertion:

Table 1.3.4: Configuration Timing Requirements

Signal	Requirement
`cfg_channel_enable`	Stable before rst_n=1
`cfg_sched_timeout_cycles`	Stable before rst_n=1
`cfg_axi_rd_xfer_beats`	Stable before rst_n=1
`cfg_axi_wr_xfer_beats`	Stable before rst_n=1

5.5 Reset Macros

RAPIDS uses standardized reset macros from `reset_defs.svh`:

```
`include "reset_defs.svh"

// Standard reset pattern
`ALWAYS_FF_RST(clk, rst_n,
    if (`RST_ASSERTED(rst_n)) begin
        r_state <= IDLE;
        r_counter <= '0;
    end else begin
        r_state <= w_next_state;
        r_counter <= w_next_counter;
    end
)
```

5.6 Power-On Reset Considerations

- External reset should be held for minimum 10 clock cycles after clock is stable
- Configuration registers should be programmed before or during reset
- No AXI transactions until reset is deasserted and channels are enabled

---

Last Updated: 2025-01-10

6 Scheduler Specification

Module: `scheduler.sv` **Location:** `projects/components/rapids/rtl/fub_beats/`
Status: Implemented **Last Updated:** 2025-01-10

6.1 Overview

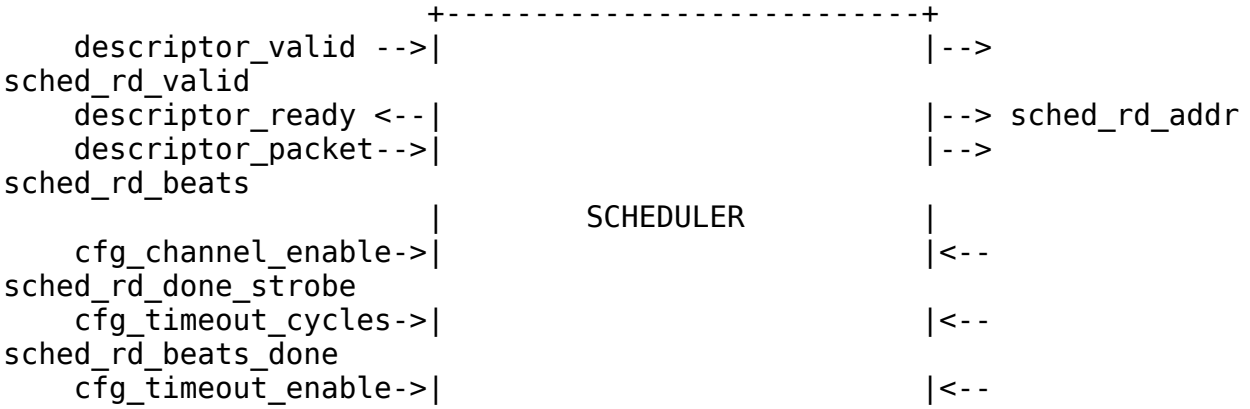
The Scheduler coordinates descriptor-based data transfers for a single RAPIDS channel. It receives descriptors from the descriptor engine, parses transfer parameters, and coordinates read/write operations through the data paths.

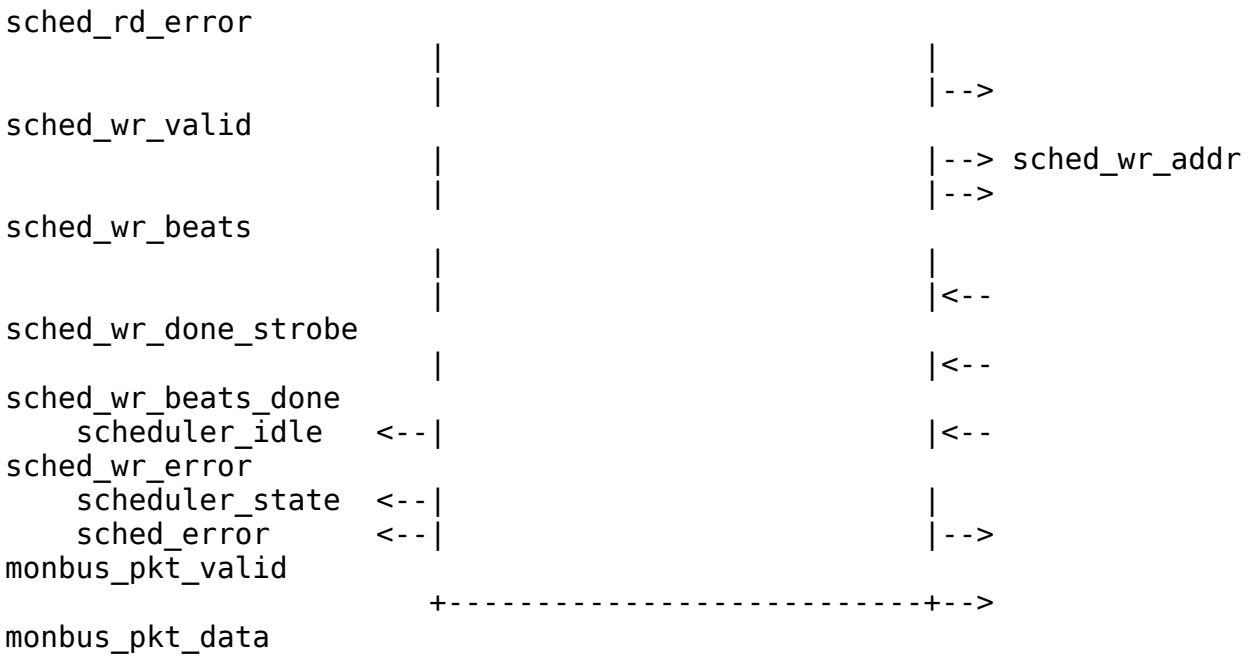
6.1.1 Key Features

- **Descriptor Processing:** Parses 256-bit descriptors for transfer parameters
- **Beat-Based Tracking:** All lengths tracked in beats (data-width units)
- **Concurrent Read/Write:** Simultaneous read and write to prevent deadlock
- **Timeout Detection:** Configurable watchdog for stalled transfers
- **MonBus Integration:** State transition and error event reporting
- **Error Aggregation:** Combines errors from descriptor engine and data paths

6.1.2 Block Diagram

6.1.3 Figure 2.1.1: Scheduler Block Diagram





Source: 02_scheduler_block.mmd

6.2 Parameters

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

// Monitor Bus Parameters
parameter logic [7:0] MON_AGENT_ID = 8'h30;      // RAPIDS Scheduler
Agent ID
parameter logic [3:0] MON_UNIT_ID = 4'h1;        // Unit identifier
parameter logic [5:0] MON_CHANNEL_ID = 6'h0;     // Base channel ID

// Descriptor Width
parameter int DESC_WIDTH = 256;                  // RAPIDS descriptor
width
```

: Table 2.1.1: Scheduler Parameters

6.3 Port List

6.3.1 Clock and Reset

Table 2.1.2: Clock and Reset

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

6.3.2 Configuration Interface

Table 2.1.3: Configuration Interface

Signal	Direction	Width	Description
`cfg_channel_enable`	input	1	Enable this channel
`cfg_channel_reset`	input	1	Channel soft reset
`cfg_sched_timeout_cycles`	input	16	Timeout threshold
`cfg_sched_timeout_enable`	input	1	Enable timeout detection

6.3.3 Descriptor Engine Interface

Table 2.1.4: Descriptor Engine Interface

Signal	Direction	Width	Description
`descriptor_valid`	input	1	Descriptor valid
`descriptor_ready`	output	1	Ready to accept descriptor
`descriptor_packet`	input	256	256-bit descriptor
`descriptor_error`	input	1	Error from descriptor engine

6.3.4 Data Read Interface

Table 2.1.5: Data Read Interface

Signal	Direction	Width	Description
`sched_rd_valid`	output	1	Read request valid
`sched_rd_addr`	output	AW	Source address
`sched_rd_beats`	output	32	Beats to read
`sched_rd_done_strobe`	input	1	Read complete strobe
`sched_rd_beats_done`	input	32	Beats completed
`sched_rd_error`	input	1	Read error

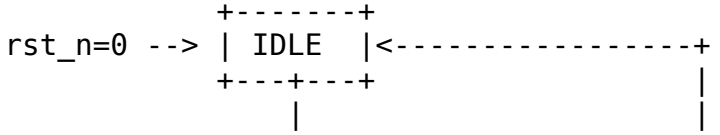
6.3.5 Data Write Interface

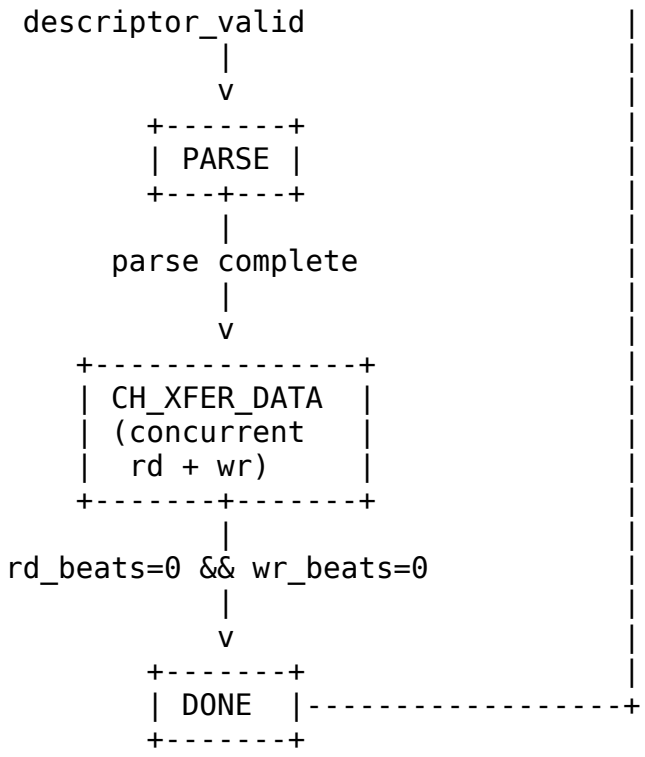
Table 2.1.6: Data Write Interface

Signal	Direction	Width	Description
`sched_wr_valid`	output	1	Write request valid
`sched_wr_addr`	output	AW	Destination address
`sched_wr_beats`	output	32	Beats to write
`sched_wr_done_strobe`	input	1	Write complete strobe
`sched_wr_beats_done`	input	32	Beats completed
`sched_wr_error`	input	1	Write error

6.4 FSM States

6.4.1 Figure 2.1.2: Scheduler FSM State Diagram





Source: 02_scheduler_fsm.mmd

6.4.2 State Encoding

Table 2.1.7: FSM State Encoding

State	Encoding	Description
IDLE	7'b0000001	Waiting for descriptor
PARSE	7'b0000010	Parsing descriptor fields
CH_XFER_DATA	7'b0000100	Concurrent read/write transfer
DONE	7'b0001000	Transfer complete, MonBus report
ERROR	7'b0010000	Error handling
TIMEOUT	7'b0100000	Timeout occurred
SOFT_RESET	7'b1000000	Soft reset handling

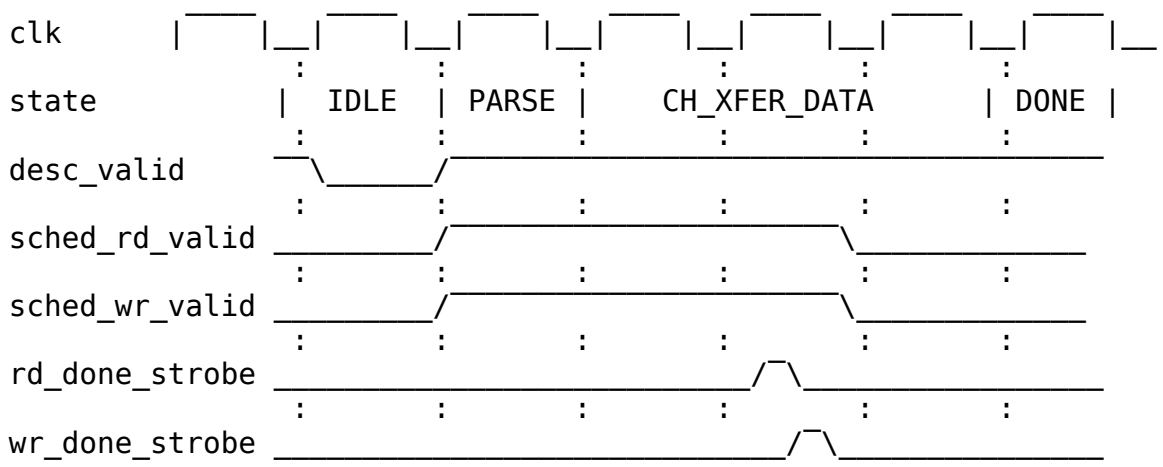
6.5 Operation

6.5.1 Descriptor Format

```
Bits [63:0]    - src_addr: Source memory address
Bits [127:64]  - dst_addr: Destination memory address
Bits [159:128] - length: Transfer length in beats
Bits [191:160] - next_ptr: Next descriptor pointer (0 = last)
Bits [195:192] - channel_id: Channel identifier
Bits [196]     - last: Last descriptor flag
Bits [197]     - gen_irq: Generate interrupt
Bits [255:198] - reserved
```

6.5.2 Transfer Sequence

6.5.3 Figure 2.1.3: Basic Transfer Timing



TODO: Replace with simulation-generated waveform

6.6 Error Handling

Table 2.1.8: Error Handling

Error Source	Detection	Response
Descriptor engine	`descriptor_error`	Transition to ERROR state
Read engine	`sched_rd_error`	Set error flag, continue or abort
Write engine	`sched_wr_error`	Set error flag, continue or abort
Timeout	Counter overflow	Transition to

Error Source	Detection	Response
		TIMEOUT state

6.7 MonBus Events

Table 2.1.9: MonBus Event Codes

Event Code	Description
0x01	Descriptor received
0x02	Transfer started
0x03	Transfer complete
0x04	Error occurred
0x05	Timeout

Last Updated: 2025-01-10

---

7 Descriptor Engine Specification

Module: `descriptor_engine.sv` **Location:** `projects/components/rapids/rtl/fub_beats/` **Status:** Implemented **Last Updated:** 2025-01-10

7.1 Overview

The Descriptor Engine fetches and manages 256-bit descriptors from memory via AXI4. It maintains a prefetch FIFO to hide memory latency and provides parsed descriptors to the scheduler.

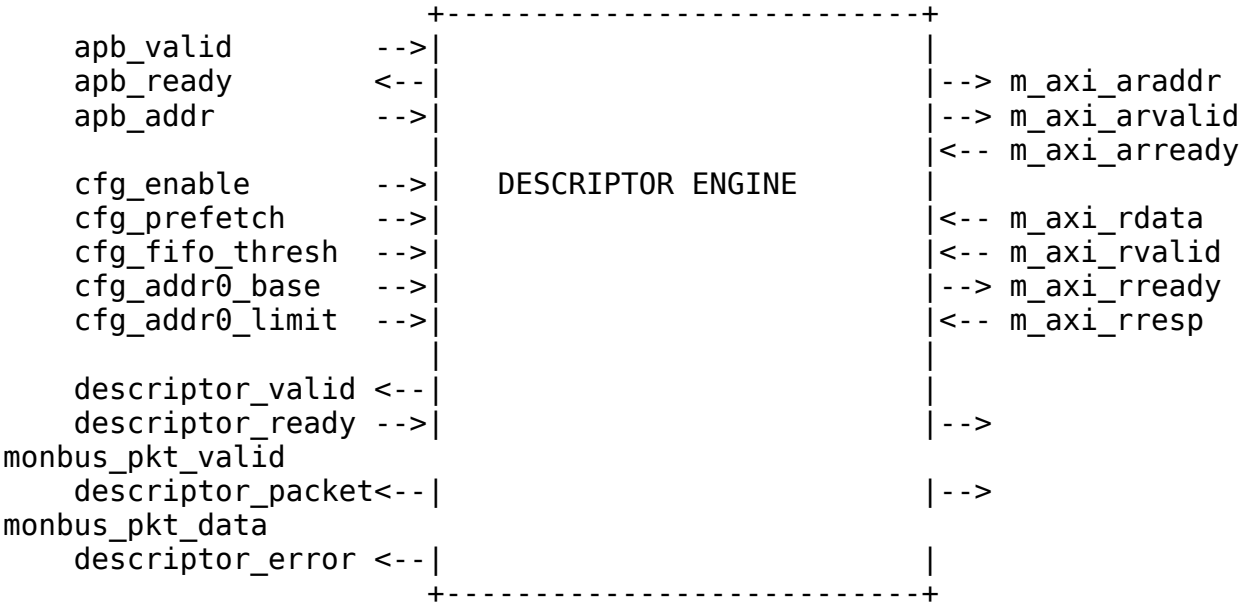
7.1.1 Key Features

- **AXI4 Descriptor Fetch:** Single-beat 256-bit reads
- **Prefetch FIFO:** Configurable depth for latency hiding

- **Address Range Checking:** Validates descriptor addresses
- **Error Detection:** Address violations, AXI errors
- **MonBus Integration:** Fetch events and error reporting

7.1.2 Block Diagram

7.1.3 Figure 2.2.1: Descriptor Engine Block Diagram



Source: 02_descriptor_engine_block.mmd

7.2 Parameters

```
parameter int CHANNEL_ID = 0;                    // Channel identifier
parameter int NUM_CHANNELS = 8;                  // Total channels
parameter int ADDR_WIDTH = 64;                   // Address bus width
parameter int FIFO_DEPTH = 4;                    // Prefetch FIFO
depth

// Monitor Bus Parameters
parameter logic [7:0] MON_AGENT_ID = 8'h10;      // Descriptor Engine
Agent ID
parameter logic [3:0] MON_UNIT_ID = 4'h1;        // Unit identifier
```

: Table 2.2.1: Descriptor Engine Parameters

7.3 Port List

7.3.1 Clock and Reset

Table 2.2.2: Clock and Reset

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

7.3.2 APB Programming Interface

Table 2.2.3: APB Programming Interface

Signal	Direction	Width	Description
`apb_valid`	input	1	Kick-off valid (start descriptor chain)
`apb_ready`	output	1	Ready to accept kick-off
`apb_addr`	input	AW	First descriptor address

7.3.3 Configuration Interface

Table 2.2.4: Configuration Interface

Signal	Direction	Width	Description
`cfg_enable`	input	1	Enable descriptor engine
`cfg_prefetch`	input	1	Enable prefetching
`cfg_fifo_thresh`	input	4	Prefetch threshold
`cfg_addr0_base`	input	AW	Valid address range 0 base
`cfg_addr0_limit`	input	AW	Valid address range 0 limit

Signal	Direction	Width	Description
`cfg_addr1_base`	input	AW	Valid address range 1 base
`cfg_addr1_limit`	input	AW	Valid address range 1 limit

7.3.4 AXI4 Read Master Interface

Table 2.2.5: AXI4 Read Master Interface

Signal	Direction	Width	Description
`m_axi_araddr`	output	AW	Read address
`m_axi_arvalid`	output	1	Address valid
`m_axi_arready`	input	1	Address ready
`m_axi_rdata`	input	256	Read data (descriptor)
`m_axi_rvalid`	input	1	Data valid
`m_axi_rready`	output	1	Data ready
`m_axi_rresp`	input	2	Read response

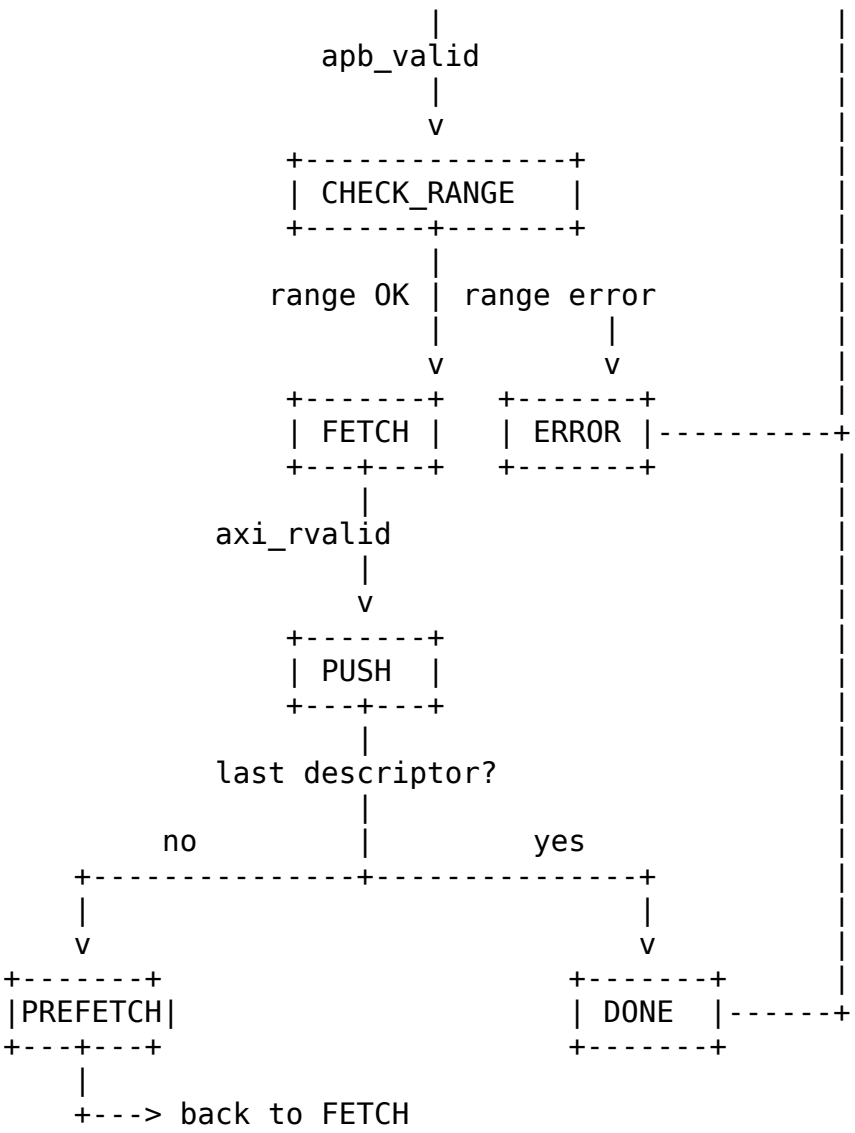
7.3.5 Scheduler Interface

Table 2.2.6: Scheduler Interface

Signal	Direction	Width	Description
`descriptor_valid`	output	1	Descriptor valid
`descriptor_ready`	input	1	Scheduler ready
`descriptor_packet`	output	256	Parsed descriptor
`descriptor_error`	output	1	Error flag

7.4 FSM States

7.4.1 Figure 2.2.2: Descriptor Engine FSM

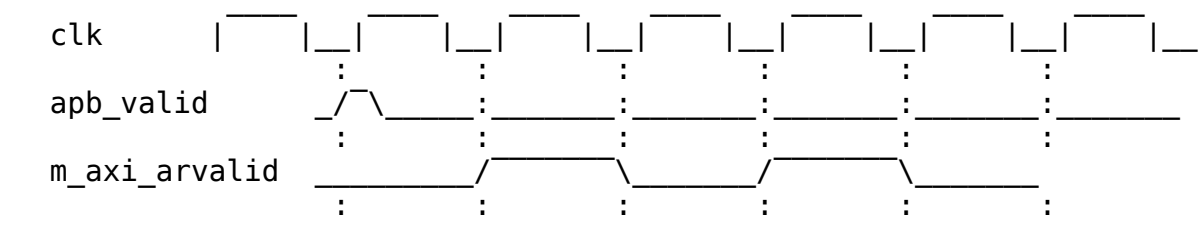


Source: 02_descriptor_engine_fsm.mmd

7.5 Operation

7.5.1 Descriptor Chain Processing

7.5.2 Figure 2.2.3: Descriptor Chain Timing



```
m_axi_araddr   X|=DESC0=|XXXXXXX|=DESC1=|XXXXXXX|XXXXXXX
                :       :       :       :       :       :
m_axi_rvalid   _____________________/‾\_______________/‾\_____
                :       :       :       :       :       :
desc_valid     _________________________/                 ‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾
```

TODO: Replace with simulation-generated waveform

7.6 Address Range Checking

Descriptors must fall within configured valid address ranges:

```
Valid if: (addr >= cfg_addr0_base && addr <= cfg_addr0_limit)
       || (addr >= cfg_addr1_base && addr <= cfg_addr1_limit)
```

Invalid addresses generate an error and halt the descriptor chain.

Last Updated: 2025-01-10

---

8 AXI Read Engine Specification

Module: `axi_read_engine.sv` **Location:** `projects/components/rapids/rtl/fub_beats/` **Status:** Implemented **Last Updated:** 2025-01-10

8.1 Overview

The AXI Read Engine performs burst reads from system memory and writes data to the SRAM buffer. It operates as a streaming pipeline with no FSM, maximizing throughput through continuous data flow.

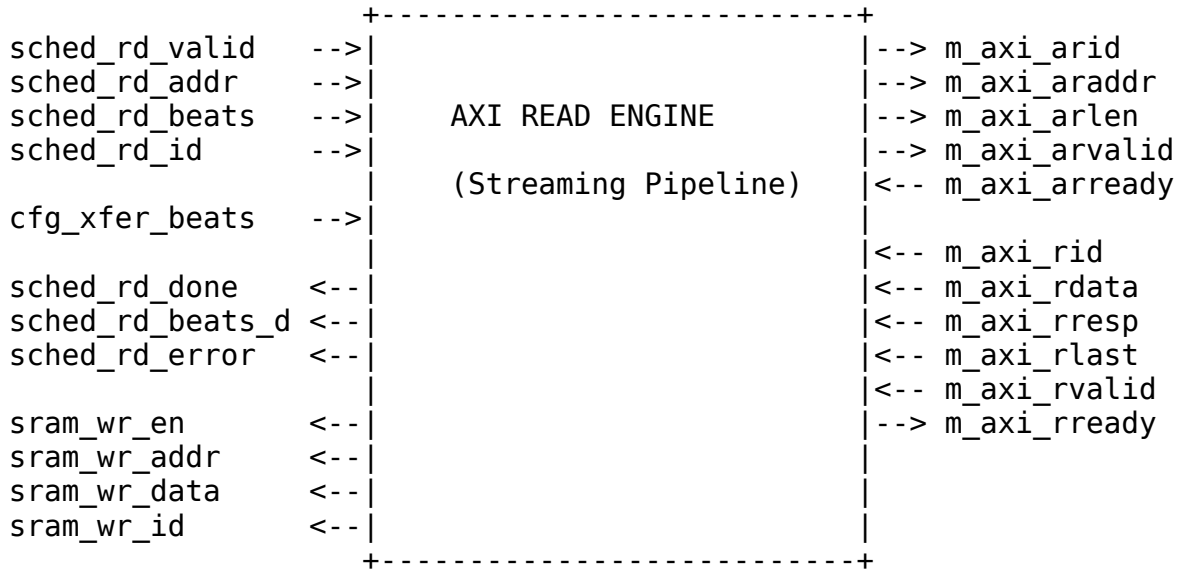
8.1.1 Key Features

- **Streaming Pipeline:** No FSM - pure streaming architecture

- **Multi-Channel Support:** Channel ID passed through AXI ID field
- **Configurable Burst Length:** Up to 256 beats per burst
- **Latency Compensation:** Integrated beats_latency_bridge
- **Error Handling:** AXI RRESP error detection and reporting

8.1.2 Block Diagram

8.1.3 Figure 2.3.1: AXI Read Engine Block Diagram



Source: 02_axi_read_engine_block.mmd

8.2 Parameters

```
parameter int NUM_CHANNELS = 8;                  // Number of channels
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_OUTSTANDING = 8;               // Max outstanding AR
transactions
parameter int PIPELINE = 0;                      // Pipeline stages

// Derived
parameter int CHAN_WIDTH = $clog2(NUM_CHANNELS);
```

: Table 2.3.1: AXI Read Engine Parameters

8.3 Port List

8.3.1 Clock and Reset

Table 2.3.2: Clock and Reset

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

8.3.2 Scheduler Interface

Table 2.3.3: Scheduler Interface

Signal	Direction	Width	Description
`sched_rd_valid`	input	1	Read request valid
`sched_rd_addr`	input	AW	Source address
`sched_rd_beats`	input	32	Total beats to read
`sched_rd_id`	input	CW	Channel ID
`sched_rd_done_strobe`	output	1	Burst complete strobe
`sched_rd_beats_done`	output	32	Beats completed
`sched_rd_error`	output	1	Error flag

8.3.3 AXI4 Read Master Interface

Table 2.3.4: AXI4 Read Master Interface

Signal	Direction	Width	Description
`m_axi_arid`	output	IW	Read address ID
`m_axi_araddr`	output	AW	Read address
`m_axi_arlen`	output	8	Burst length (beats - 1)
`m_axi_arsize`	output	3	Burst size
`m_axi_arburst`	output	2	Burst type (INCR)

Signal	Direction	Width	Description
m_axi_arvalid	output	1	Address valid
m_axi_arready	input	1	Address ready
m_axi_rid	input	IW	Read data ID
m_axi_rdata	input	DW	Read data
m_axi_rresp	input	2	Read response
m_axi_rlast	input	1	Last beat
m_axi_rvalid	input	1	Data valid
m_axi_rready	output	1	Data ready

8.3.4 SRAM Write Interface

Table 2.3.5: SRAM Write Interface

Signal	Direction	Width	Description
sram_wr_en	output	1	SRAM write enable
sram_wr_addr	output	AW	SRAM write address
sram_wr_data	output	DW	SRAM write data
sram_wr_id	output	CW	Channel ID for data

8.4 Operation

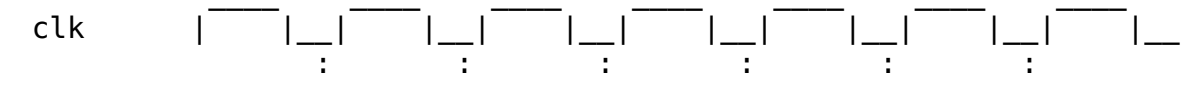
8.4.1 Streaming Pipeline Architecture

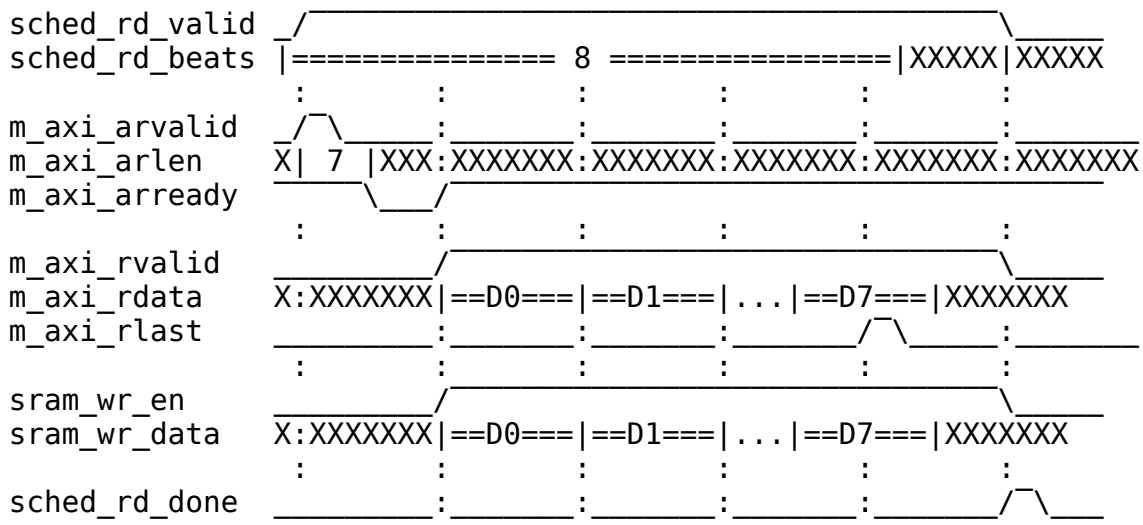
The AXI read engine uses a **streaming pipeline** with no FSM:

```
Scheduler Request --> AR Channel --> R Channel --> SRAM Write
      |                   |             |              |
      +--- beat count     +-- addr      +-- data       +-- wr_en
           tracking           issue         receive        generate
```

8.4.2 Timing Diagram

8.4.3 Figure 2.3.2: AXI Read Burst Timing





TODO: Replace with simulation-generated waveform

8.5 Burst Segmentation

Large transfers are segmented into AXI-compliant bursts:

```
Total beats = 256
cfg_xfer_beats = 64

Bursts issued:
  AR[0]: addr=0x1000, len=63 (64 beats)
  AR[1]: addr=0x2000, len=63 (64 beats)
  AR[2]: addr=0x3000, len=63 (64 beats)
  AR[3]: addr=0x4000, len=63 (64 beats)
```

8.6 Error Handling

Table 2.3.6: Error Handling

Error	Detection	Response
AXI SLVERR	`m_axi_rresp == 2'b10`	Set `sched_rd_error`, continue
AXI DECERR	`m_axi_rresp == 2'b11`	Set `sched_rd_error`, continue

Last Updated: 2025-01-10

9 AXI Write Engine Specification

Module: `axi_write_engine.sv` **Location:** `projects/components/rapids/rtl/fub_beats/` **Status:** Implemented **Last Updated:** 2025-01-10

9.1 Overview

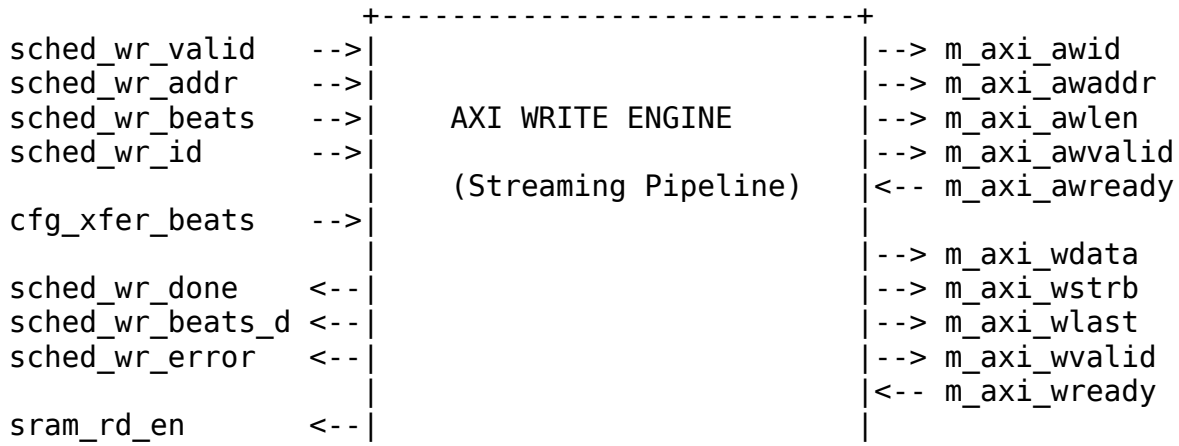
The AXI Write Engine reads data from SRAM and performs burst writes to system memory. It operates as a streaming pipeline and includes write response handling for completion tracking.

9.1.1 Key Features

- **Streaming Pipeline:** No FSM - pure streaming architecture
- **Multi-Channel Support:** Channel ID in AXI ID field
- **Write Response Tracking:** B channel completion handling
- **Configurable Burst Length:** Up to 256 beats per burst
- **Latency Compensation:** Integrated beats_latency_bridge

9.1.2 Block Diagram

9.1.3 Figure 2.4.1: AXI Write Engine Block Diagram



```
    sram_rd_addr     <--|                            |<-- m_axi_bid
    sram_rd_data     -->|                            |<-- m_axi_bresp
    sram_rd_id       <--|                            |<-- m_axi_bvalid
                        |                            |--> m_axi_bready
                        +----------------------------+
```

Source: 02_axi_write_engine_block.mmd

9.2 Parameters

```
parameter int NUM_CHANNELS = 8;                  // Number of channels
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_OUTSTANDING = 8;               // Max outstanding AW
transactions
parameter int W_FIFO_DEPTH = 64;                 // Write data FIFO
depth
parameter int B_FIFO_DEPTH = 16;                 // Write response
FIFO depth
parameter int PIPELINE = 0;                      // Pipeline stages

// Derived
parameter int CHAN_WIDTH = $clog2(NUM_CHANNELS);
```

: Table 2.4.1: AXI Write Engine Parameters

9.3 Port List

9.3.1 Clock and Reset

Table 2.4.2: Clock and Reset

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

9.3.2 Scheduler Interface

Table 2.4.3: Scheduler Interface

Signal	Direction	Width	Description
`sched_wr_valid`	input	1	Write request valid

Signal	Direction	Width	Description
sched_wr_addr	input	AW	Destination address
sched_wr_beats	input	32	Total beats to write
sched_wr_id	input	CW	Channel ID
sched_wr_done_strobe	output	1	Burst complete strobe
sched_wr_beats_done	output	32	Beats completed
sched_wr_error	output	1	Error flag

9.3.3 AXI4 Write Master Interface

Table 2.4.4: AXI4 Write Master Interface

Signal	Direction	Width	Description
m_axi_awid	output	IW	Write address ID
m_axi_awaddr	output	AW	Write address
m_axi_awlen	output	8	Burst length (beats - 1)
m_axi_awsize	output	3	Burst size
m_axi_awburst	output	2	Burst type (INCR)
m_axi_awvalid	output	1	Address valid
m_axi_awready	input	1	Address ready
m_axi_wdata	output	DW	Write data
m_axi_wstrb	output	DW/8	Write strobes
m_axi_wlast	output	1	Last beat
m_axi_wvalid	output	1	Data valid
m_axi_wready	input	1	Data ready
m_axi_bid	input	IW	Response ID
m_axi_bresp	input	2	Write response
m_axi_bvalid	input	1	Response valid

Signal	Direction	Width	Description
`m_axi_bready`	output	1	Response ready

9.3.4 SRAM Read Interface

Table 2.4.5: SRAM Read Interface

Signal	Direction	Width	Description
`sram_rd_en`	output	1	SRAM read enable
`sram_rd_addr`	output	AW	SRAM read address
`sram_rd_data`	input	DW	SRAM read data
`sram_rd_id`	output	CW	Channel ID for read

9.4 Operation

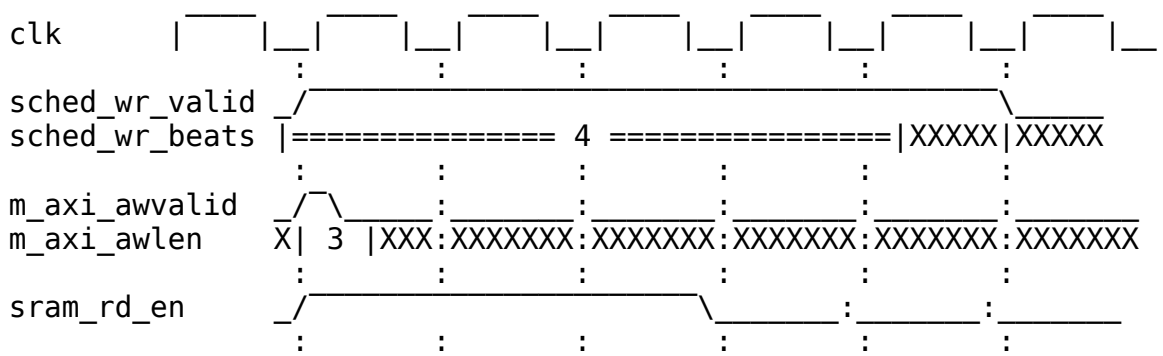
9.4.1 Write Transaction Phases

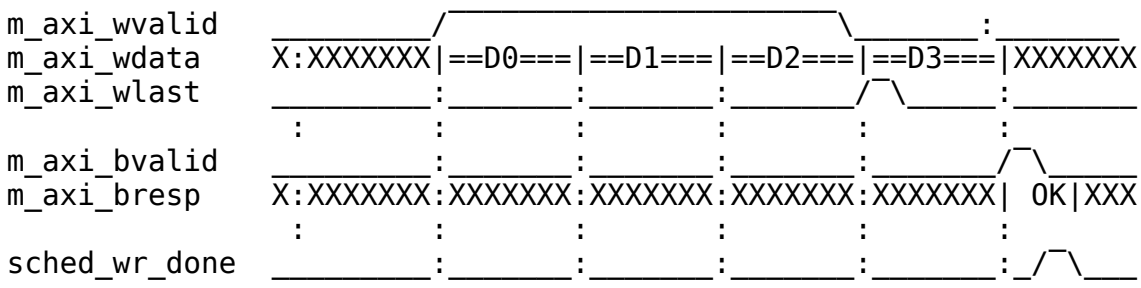
AXI writes have three phases that can overlap:

```
Phase 1: AW (Address)    Phase 2: W (Data)    Phase 3: B (Response)
    |                        |                       |
    +-- Issue address        +-- Stream data         +-- Receive
response
        before data              from SRAM               track
completion
```

9.4.2 Timing Diagram

9.4.3 Figure 2.4.2: AXI Write Burst Timing





TODO: Replace with simulation-generated waveform

9.5 Error Handling

Table 2.4.6: Error Handling

Error	Detection	Response
AXI SLVERR	`m_axi_bresp == 2'b10`	Set `sched_wr_error`
AXI DECERR	`m_axi_bresp == 2'b11`	Set `sched_wr_error`

Last Updated: 2025-01-10

---

10 Beats Alloc Control Specification

Module: `beats_alloc_ctrl.sv` **Location:** `projects/components/rapids/rtl/fub_beats/` **Status:** Implemented **Last Updated:** 2025-01-10

10.1 Overview

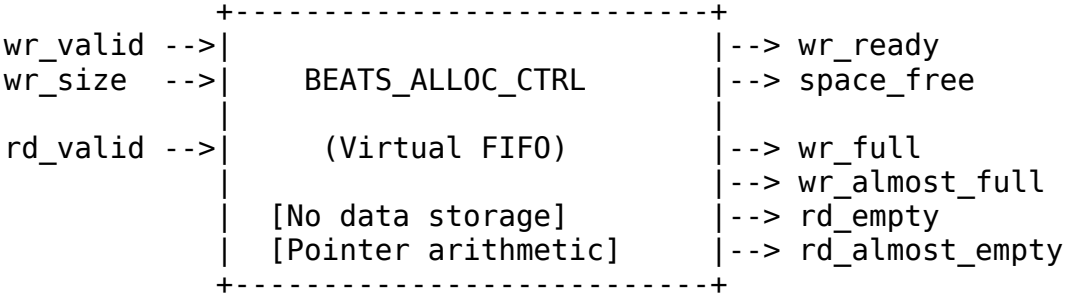
The Beats Alloc Control is a “virtual FIFO” that tracks space allocation without storing actual data. It uses FIFO pointer logic to manage pre-allocation for AXI engines, preventing race conditions between allocation and actual data arrival.

10.1.1 Key Features

- **Virtual FIFO:** Pointer tracking without data storage
- **Variable-Size Allocation:** Supports multi-beat allocation requests
- **Single-Beat Release:** Data writes release one beat at a time
- **Status Flags:** Full, almost full, empty, almost empty
- **Space Tracking:** Reports available space for flow control

10.1.2 Block Diagram

10.1.3 Figure 2.5.1: Beats Alloc Control Block Diagram



Source: 02_beats_alloc_ctrl_block.mmd

10.2 Concept: Virtual FIFO

Unlike a traditional FIFO that stores data, beats_alloc_ctrl only tracks **space reservations**:

```
Traditional FIFO:          Virtual FIFO (alloc_ctrl):

  wr_data --> [storage] -> rd_data    wr_size --> [pointers] -->
space_free

  Data moves through FIFO             Only pointers move, no data
```

10.2.1 Use Case: Pre-Allocation for AXI Reads

```
1. AXI engine requests N beats of space
   - alloc_ctrl.wr_valid = 1
   - alloc_ctrl.wr_size = N
   - wr_ptr advances by N

2. As AXI read data arrives (1 beat at a time)
   - alloc_ctrl.rd_valid = 1 (each beat)
   - rd_ptr advances by 1
```

```
3. Space becomes available again
   - space_free reflects available capacity
```

10.3 Parameters

```
parameter int DEPTH = 512;                    // Virtual FIFO depth
parameter int ALMOST_WR_MARGIN = 1;           // Almost full margin
parameter int ALMOST_RD_MARGIN = 1;           // Almost empty margin
parameter int REGISTERED = 1;                 // Registered outputs

// Derived
parameter int AW = $clog2(DEPTH);             // Address width
```

: Table 2.5.1: Beats Alloc Control Parameters

10.4 Port List

10.4.1 Clock and Reset

Table 2.5.2: Clock and Reset

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

10.4.2 Write Interface (Allocation Requests)

Table 2.5.3: Write Interface

Signal	Direction	Width	Description
`wr_valid`	input	1	Allocate space
`wr_size`	input	8	Number of beats to allocate
`wr_ready`	output	1	Space available for allocation

10.4.3 Read Interface (Data Written)

Table 2.5.4: Read Interface

Signal	Direction	Width	Description
`rd_valid`	input	1	One beat of data written
`rd_ready`	output	1	Not full (can accept)

10.4.4 Status Outputs

Table 2.5.5: Status Outputs

Signal	Direction	Width	Description
`space_free`	output	AW+1	Available space in beats
`wr_full`	output	1	No space available
`wr_almost_full`	output	1	Near full
`rd_empty`	output	1	No allocations pending
`rd_almost_empty`	output	1	Near empty

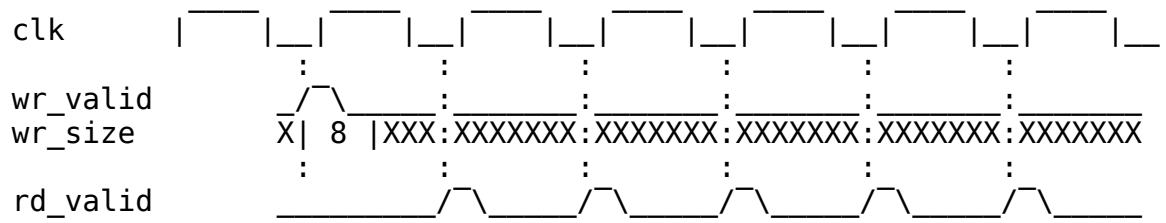
10.5 Operation

10.5.1 Pointer Arithmetic

```
space_free = DEPTH - (wr_ptr - rd_ptr)

Full condition:    wr_ptr - rd_ptr >= DEPTH
Empty condition:   wr_ptr == rd_ptr
```

10.5.2 Timing Diagram

10.5.3 Figure 2.5.2: Allocation and Release Timing



```
                    :       :       :       :       :       :
    space_free     |==512===|==504==|==505==|==506==|==507==|==508==
                    :       :       :       :       :       :
    wr_ptr         |== 0 ===|== 8 ==|== 8 ==|== 8 ==|== 8 ==|== 8 ==
    rd_ptr         |== 0 ===|== 0 ==|== 1 ==|== 2 ==|== 3 ==|== 4 ==
```

TODO: Replace with simulation-generated waveform showing actual flow control

---

10.6 Integration Example

```
// Source path: Pre-allocate SRAM space before AXI read
beats_alloc_ctrl #(
    .DEPTH(SRAM_DEPTH),
    .ALMOST_WR_MARGIN(8)
) u_src_alloc (
    .axi_aclk       (clk),
    .axi_aresetn    (rst_n),

    // Scheduler allocates space before issuing AXI read
    .wr_valid       (sched_rd_valid),
    .wr_size        (sched_rd_beats[7:0]),
    .wr_ready       (alloc_ready),

    // AXI read data arrival releases allocation
    .rd_valid       (m_axi_rvalid && m_axi_rready),
    .rd_ready       (),  // Not used for single-beat

    // Status
    .space_free     (sram_space_available),
    .wr_full        (sram_full),
    .wr_almost_full (sram_almost_full)
);
```

Last Updated: 2025-01-10

---

11 Beats Drain Control Specification

Module: `beats_drain_ctrl.sv` **Location:** `projects/components/rapids/rtl/fub_beats/` **Status:** Implemented **Last Updated:** 2025-01-10

11.1 Overview

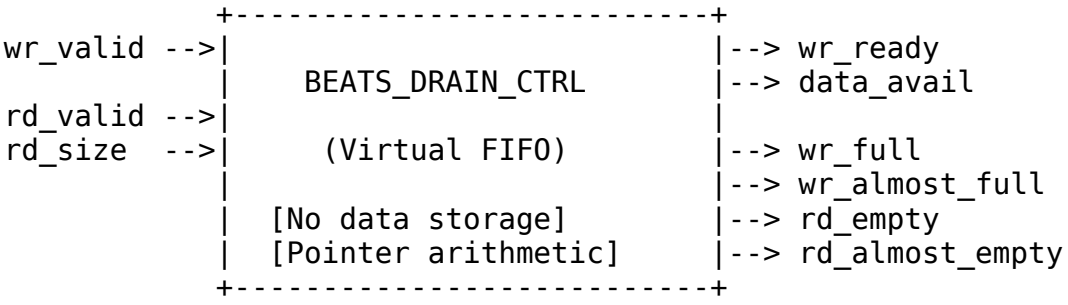
The Beats Drain Control is a "virtual FIFO" that tracks data availability without storing actual data. It complements beats_alloc_ctrl by tracking when data has been written to SRAM and is available for draining.

11.1.1 Key Features

- **Virtual FIFO:** Pointer tracking without data storage
- **Single-Beat Write:** Data arrivals tracked one beat at a time
- **Variable-Size Drain:** Supports multi-beat drain requests
- **Status Flags:** Full, almost full, empty, almost empty
- **Data Availability:** Reports beats available for drain

11.1.2 Block Diagram

11.1.3 Figure 2.6.1: Beats Drain Control Block Diagram



Source: 02_beats_drain_ctrl_block.mmd

11.2 Concept: Data Availability Tracking

The drain_ctrl tracks when data has been written to SRAM:

```
alloc_ctrl:  Tracks SPACE reservations (allocation -> actual write)
drain_ctrl:  Tracks DATA availability (data write -> drain request)
```

11.2.1 Complementary Roles

```
Source Path:
  1. Scheduler allocates space (alloc_ctrl.wr)
  2. AXI reads data into SRAM
  3. Data written to SRAM (drain_ctrl.wr)
  4. Drain consumer reads (drain_ctrl.rd)

Sink Path:
  1. Fill allocates space (alloc_ctrl.wr)
  2. Fill writes to SRAM (alloc_ctrl.rd)
  3. Data available in SRAM (drain_ctrl.wr)
  4. AXI write drains SRAM (drain_ctrl.rd)
```

11.3 Parameters

```
parameter int DEPTH = 512;                    // Virtual FIFO depth
parameter int ALMOST_WR_MARGIN = 1;           // Almost full margin
parameter int ALMOST_RD_MARGIN = 1;           // Almost empty margin
parameter int REGISTERED = 1;                 // Registered outputs

// Derived
parameter int AW = $clog2(DEPTH);             // Address width
```

: Table 2.6.1: Beats Drain Control Parameters

11.4 Port List

11.4.1 Clock and Reset

Table 2.6.2: Clock and Reset

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

11.4.2 Write Interface (Data Written to SRAM)

Table 2.6.3: Write Interface

Signal	Direction	Width	Description
`wr_valid`	input	1	One beat written to SRAM

Signal	Direction	Width	Description
wr_ready	output	1	Can track more data

11.4.3 Read Interface (Drain Requests)

Table 2.6.4: Read Interface

Signal	Direction	Width	Description
rd_valid	input	1	Drain request
rd_size	input	8	Number of beats to drain
rd_ready	output	1	Data available for drain

11.4.4 Status Outputs

Table 2.6.5: Status Outputs

Signal	Direction	Width	Description
data_avail	output	AW+1	Beats available to drain
wr_full	output	1	Tracking full
wr_almost_full	output	1	Near full
rd_empty	output	1	No data to drain
rd_almost_empty	output	1	Near empty

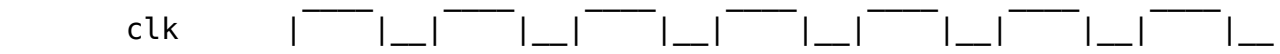
11.5 Operation

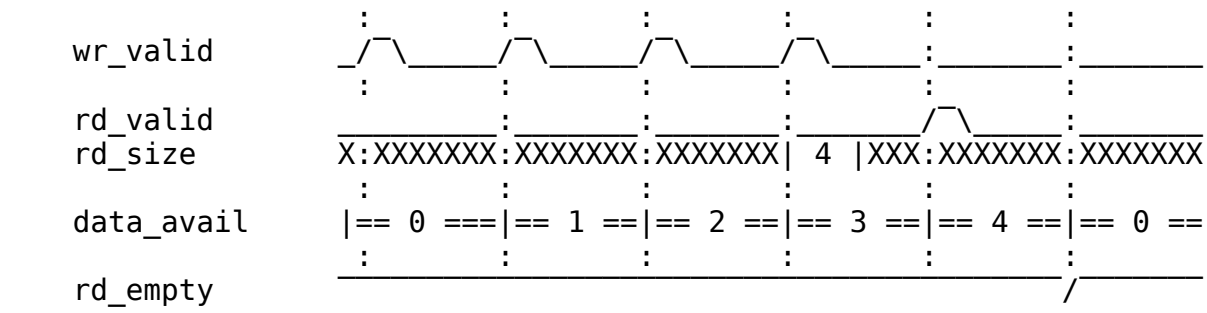
11.5.1 Pointer Arithmetic

```
data_avail = wr_ptr - rd_ptr

Full condition:    wr_ptr - rd_ptr >= DEPTH
Empty condition:   wr_ptr == rd_ptr
```

11.5.2 Timing Diagram

11.5.3 Figure 2.6.2: Data Arrival and Drain Timing





TODO: Replace with simulation-generated waveform showing actual flow control

---

11.6 Integration Example

```
// Sink path: Track data availability for AXI write engine
beats_drain_ctrl #(
    .DEPTH(SRAM_DEPTH),
    .ALMOST_RD_MARGIN(8)
) u_snk_drain (
    .axi_aclk       (clk),
    .axi_aresetn    (rst_n),

    // Fill writes data to SRAM
    .wr_valid       (fill_valid && fill_ready),
    .wr_ready       (),  // Single-beat, always ready

    // AXI write engine drains data
    .rd_valid       (axi_wr_drain_req),
    .rd_size        (axi_wr_burst_len),
    .rd_ready       (data_ready_for_drain),

    // Status
    .data_avail     (beats_to_write),
    .rd_empty       (no_data_to_write)
);
```

11.7 Relationship: alloc_ctrl vs drain_ctrl

Table 2.6.6: alloc_ctrl vs drain_ctrl Comparison

Aspect	beats_alloc_ctrl	beats_drain_ctrl
Tracks	Space reservations	Data availability
Write port	Multi-beat allocation	Single-beat arrival
Read port	Single-beat release	Multi-beat drain

Aspect	beats_alloc_ctrl	beats_drain_ctrl
Output	`space_free`	`data_avail`
Use case	Pre-allocate before fill	Know when to drain

Last Updated: 2025-01-10

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

---

12 Beats Latency Bridge Specification

Module: `beats_latency_bridge.sv` **Location:** `projects/components/rapids/rtl/fub_beats/` **Status:** Implemented **Last Updated:** 2025-01-10

12.1 Overview

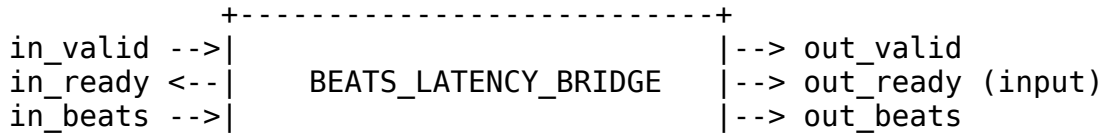
The Beats Latency Bridge provides buffering to compensate for pipeline latency between alloc_ctrl and drain_ctrl in the SRAM controller. It prevents race conditions where drain signals arrive before allocation is complete.

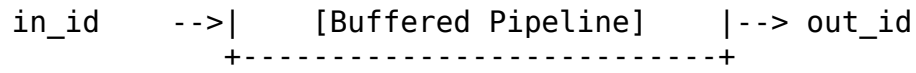
12.1.1 Key Features

- **Latency Compensation:** Buffers requests to match pipeline delays
- **Configurable Depth:** Matches expected pipeline latency
- **Flow Control:** Standard valid/ready handshaking
- **Pass-Through Data:** Preserves beat count and channel ID

12.1.2 Block Diagram

12.1.3 Figure 2.7.1: Beats Latency Bridge Block Diagram

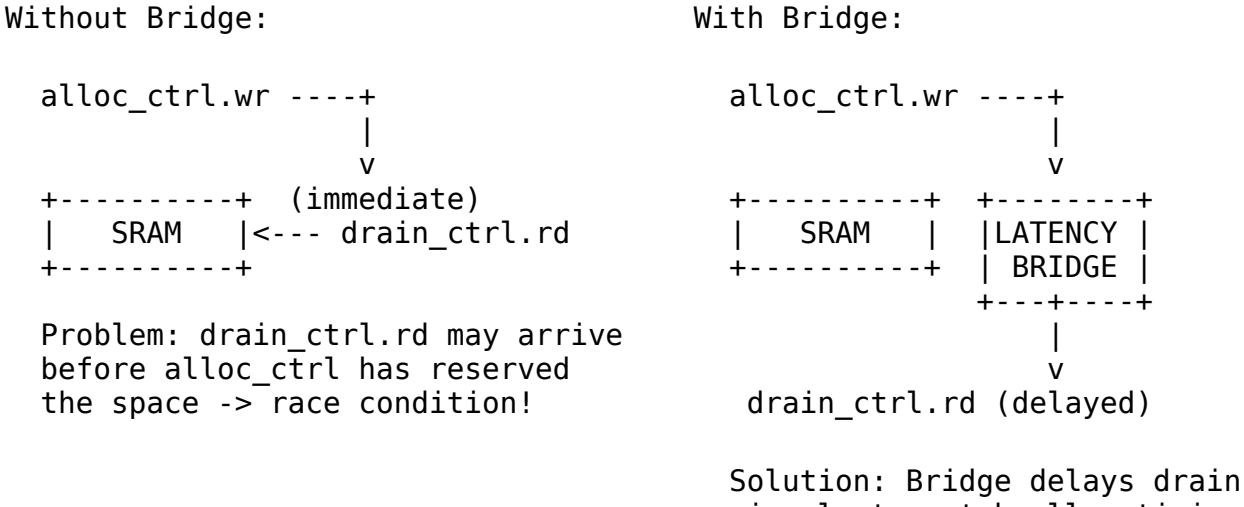




Source: 02_beats_latency_bridge_block.mmd

12.2 Concept: Why Latency Compensation?

The SRAM controller has a timing challenge:



12.3 Parameters

```
parameter int DEPTH = 4;                    // Bridge FIFO depth
parameter int BEATS_WIDTH = 8;              // Beat count width
parameter int ID_WIDTH = 3;                 // Channel ID width
parameter int REGISTERED = 1;               // Registered outputs
```

: Table 2.7.1: Beats Latency Bridge Parameters

12.4 Port List

12.4.1 Clock and Reset

Table 2.7.2: Clock and Reset

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

Signal	Direction	Width	Description
			reset

12.4.2 Input Interface

Table 2.7.3: Input Interface

Signal	Direction	Width	Description
`in_valid`	input	1	Input request valid
`in_ready`	output	1	Bridge ready to accept
`in_beats`	input	BEATS_WIDTH	Beat count
`in_id`	input	ID_WIDTH	Channel ID

12.4.3 Output Interface

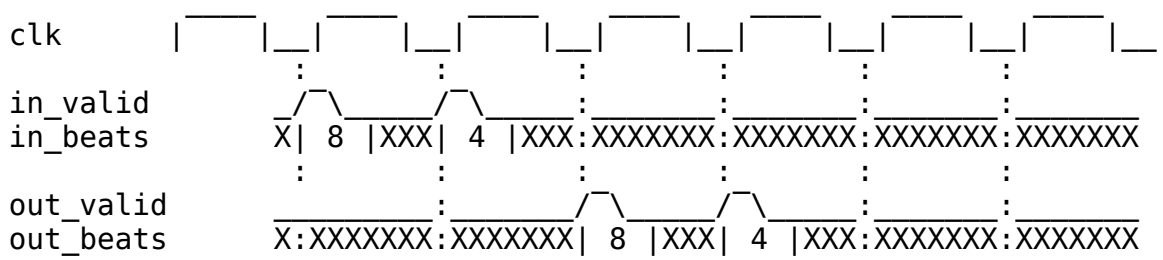
Table 2.7.4: Output Interface

Signal	Direction	Width	Description
`out_valid`	output	1	Output request valid
`out_ready`	input	1	Downstream ready
`out_beats`	output	BEATS_WIDTH	Beat count (delayed)
`out_id`	output	ID_WIDTH	Channel ID (delayed)

12.5 Operation

12.5.1 Timing Diagram

12.5.2 Figure 2.7.2: Latency Bridge Timing (Depth=2)



```
                :       :       :       :       :       :
                :       :       :       :       :       :
                +-------+       +-------+
                | 2-cycle latency through bridge |
```

TODO: Replace with simulation-generated waveform

12.6 Integration Context

The latency bridge is used within the SRAM controller:

```
// In snk_sram_controller_unit
beats_alloc_ctrl u_alloc (
    // Allocation on fill requests
    .wr_valid       (fill_valid),
    .wr_size        (fill_size),
    // Release on actual data write
    .rd_valid       (sram_wr_complete)
);

beats_latency_bridge u_bridge (
    // Input from SRAM write path
    .in_valid       (sram_wr_complete),
    .in_beats       (beats_written),
    // Output to drain controller (delayed)
    .out_valid      (drain_trigger),
    .out_beats      (drain_beats)
);

beats_drain_ctrl u_drain (
    // Delayed notification of data available
    .wr_valid       (drain_trigger),
    // Drain requests from write engine
    .rd_valid       (axi_drain_req),
    .rd_size        (axi_drain_size)
);
```

12.7 Design Considerations

Table 2.7.5: Bridge Depth Selection

Depth	Use Case	Latency
2	Minimal pipeline	2 cycles
4	Standard pipeline	4 cycles

Depth	Use Case	Latency
8	Deep pipeline	8 cycles

The bridge depth should match the maximum pipeline latency from alloc_ctrl write to SRAM data being valid for drain_ctrl.

Last Updated: 2025-01-10

---

13 Beats Scheduler Group Specification

Module: `beats_scheduler_group.sv` **Location:** `projects/components/rapids/rtl/macro_beats/` **Status:** Implemented **Last Updated:** 2025-01-10

13.1 Overview

The Beats Scheduler Group wraps a single channel's scheduler and descriptor engine together with MonBus aggregation. It provides a clean integration point for the scheduler_group_array.

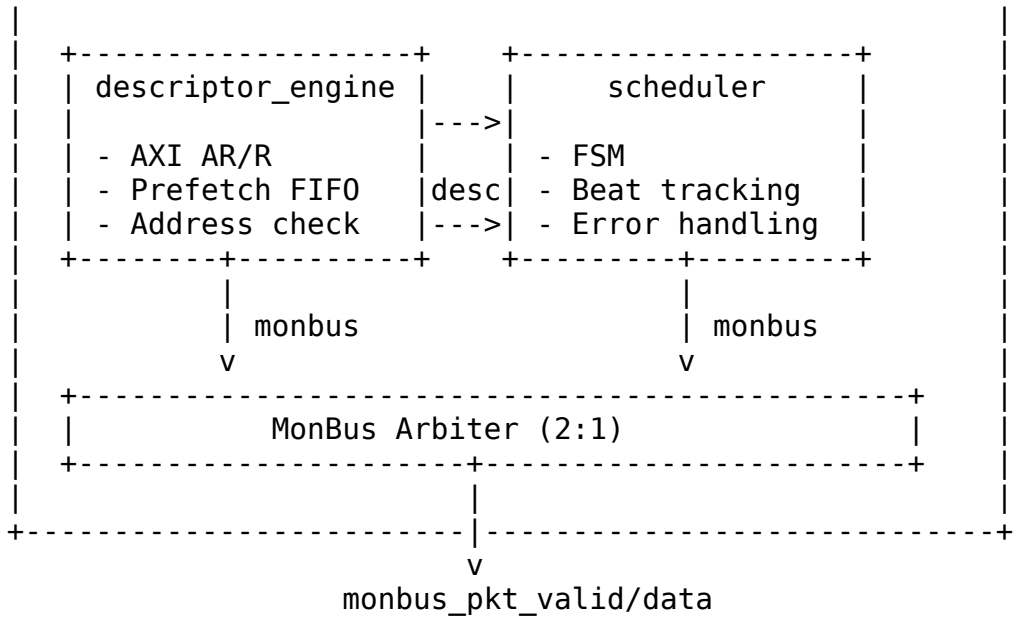
13.1.1 Key Features

- **Single Channel Integration:** One scheduler + one descriptor engine
- **MonBus Aggregation:** Combines scheduler and descriptor engine MonBus outputs
- **Configuration Pass-Through:** Routes configuration to both sub-modules
- **Status Aggregation:** Combined idle and error status

13.1.2 Block Diagram

13.1.3 Figure 3.1.1: Beats Scheduler Group Block Diagram

```
                    beats_scheduler_group
    +-----------------------------------------------------+
```



Source: 03_beats_scheduler_group_block.mmd

13.2 Parameters

```
parameter int CHANNEL_ID = 0;                    // Channel identifier
parameter int NUM_CHANNELS = 8;                  // Total channels
parameter int ADDR_WIDTH = 64;                   // Address bus width
parameter int DATA_WIDTH = 512;                  // Data bus width

// Monitor Bus Parameters
parameter int DESC_MON_AGENT_ID = 16 + CHANNEL_ID;  // Descriptor
engine agent
parameter int SCHED_MON_AGENT_ID = 48 + CHANNEL_ID; // Scheduler agent
parameter int MON_UNIT_ID = 1;
```

: Table 3.1.1: Beats Scheduler Group Parameters

13.3 Port List

13.3.1 Clock and Reset

Table 3.1.2: Clock and Reset

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

Signal	Direction	Width	Description
			reset

13.3.2 APB Programming Interface

Table 3.1.3: APB Programming Interface

Signal	Direction	Width	Description
`apb_valid`	input	1	Channel kick-off
`apb_ready`	output	1	Ready for kick-off
`apb_addr`	input	AW	First descriptor address

13.3.3 Configuration Interface

Table 3.1.4: Configuration Interface

Signal	Direction	Width	Description
`cfg_channel_enable`	input	1	Enable channel
`cfg_channel_reset`	input	1	Soft reset
`cfg_sched_timeout_cycles`	input	16	Timeout threshold
`cfg_sched_timeout_enable`	input	1	Enable timeout
`cfg_desceng_enable`	input	1	Enable descriptor engine
`cfg_desceng_prefetch`	input	1	Enable prefetching
`cfg_desceng_fifo_thresh`	input	4	Prefetch threshold
`cfg_desceng_addr0_base`	input	AW	Address range 0 base
`cfg_desceng_addr0_limit`	input	AW	Address range 0 limit

13.3.4 Descriptor AXI Master Interface

Table 3.1.5: Descriptor AXI Master Interface

Signal	Direction	Width	Description
`m_axi_arvalid`	output	1	AR channel valid
`m_axi_arready`	input	1	AR channel ready
`m_axi_araddr`	output	AW	AR address
`m_axi_rvalid`	input	1	R channel valid
`m_axi_rready`	output	1	R channel ready
`m_axi_rdata`	input	256	R data (descriptor)
`m_axi_rresp`	input	2	R response

13.3.5 Scheduler Data Interfaces

Table 3.1.6: Scheduler Data Interfaces

Signal	Direction	Width	Description
`sched_rd_valid`	output	1	Read request
`sched_rd_addr`	output	AW	Read address
`sched_rd_beats`	output	32	Read beats
`sched_rd_done_strobe`	input	1	Read complete
`sched_wr_valid`	output	1	Write request
`sched_wr_addr`	output	AW	Write address
`sched_wr_beats`	output	32	Write beats
`sched_wr_done_strobe`	input	1	Write complete

13.3.6 Status

Table 3.1.7: Status Interface

Signal	Direction	Width	Description
`scheduler_idle`	output	1	Scheduler idle
`descriptor_engine_idle`	output	1	Descriptor engine idle
`scheduler_state`	output	7	FSM state
`sched_error`	output	1	Error flag

13.3.7 MonBus Interface

Table 3.1.8: MonBus Interface

Signal	Direction	Width	Description
`monbus_pkt_valid`	output	1	Packet valid
`monbus_pkt_ready`	input	1	Consumer ready
`monbus_pkt_data`	output	64	Packet data

13.4 Internal Architecture

```
Internal Signal Flow:

apb_valid/addr -----> descriptor_engine -----> scheduler
                            |                       |
                      desc_valid/packet           |
                            |                       |
                            +--> descriptor_ready <--+

MonBus Aggregation:

desc_engine.monbus_pkt ----+
                           |---> round_robin_arbiter ---> monbus_out
scheduler.monbus_pkt   ----+
```

Last Updated: 2025-01-10

14 Beats Scheduler Group Array Specification

Module: `beats_scheduler_group_array.sv` **Location:** `projects/components/rapids/rtl/macro_beats/` **Status:** Implemented **Last Updated:** 2025-01-10

14.1 Overview

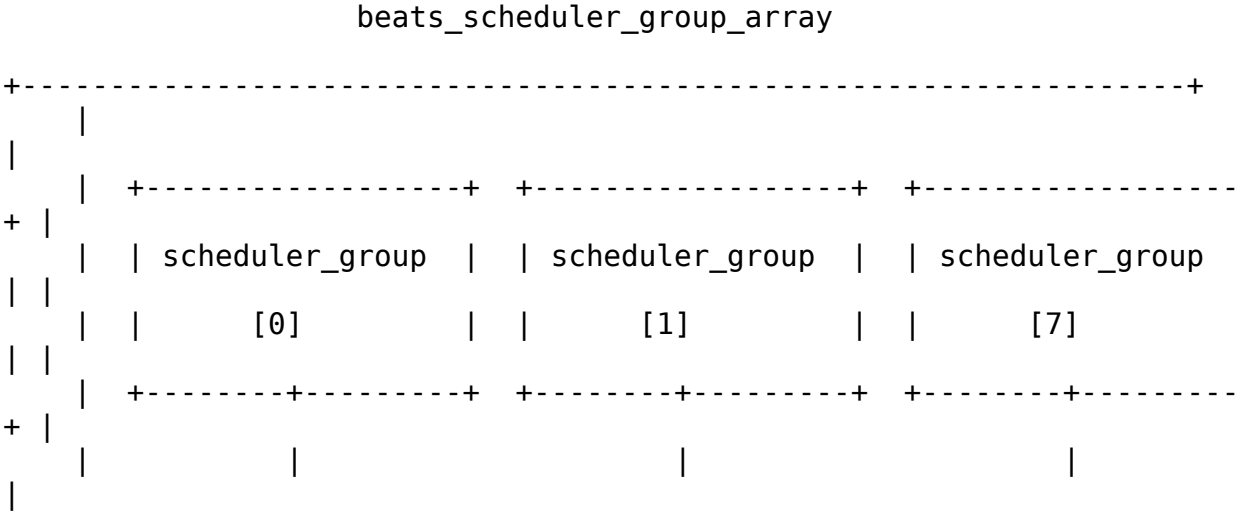
The Beats Scheduler Group Array instantiates 8 scheduler groups with a shared descriptor AXI master and unified MonBus output. It provides the complete scheduler infrastructure for all RAPIDS channels.

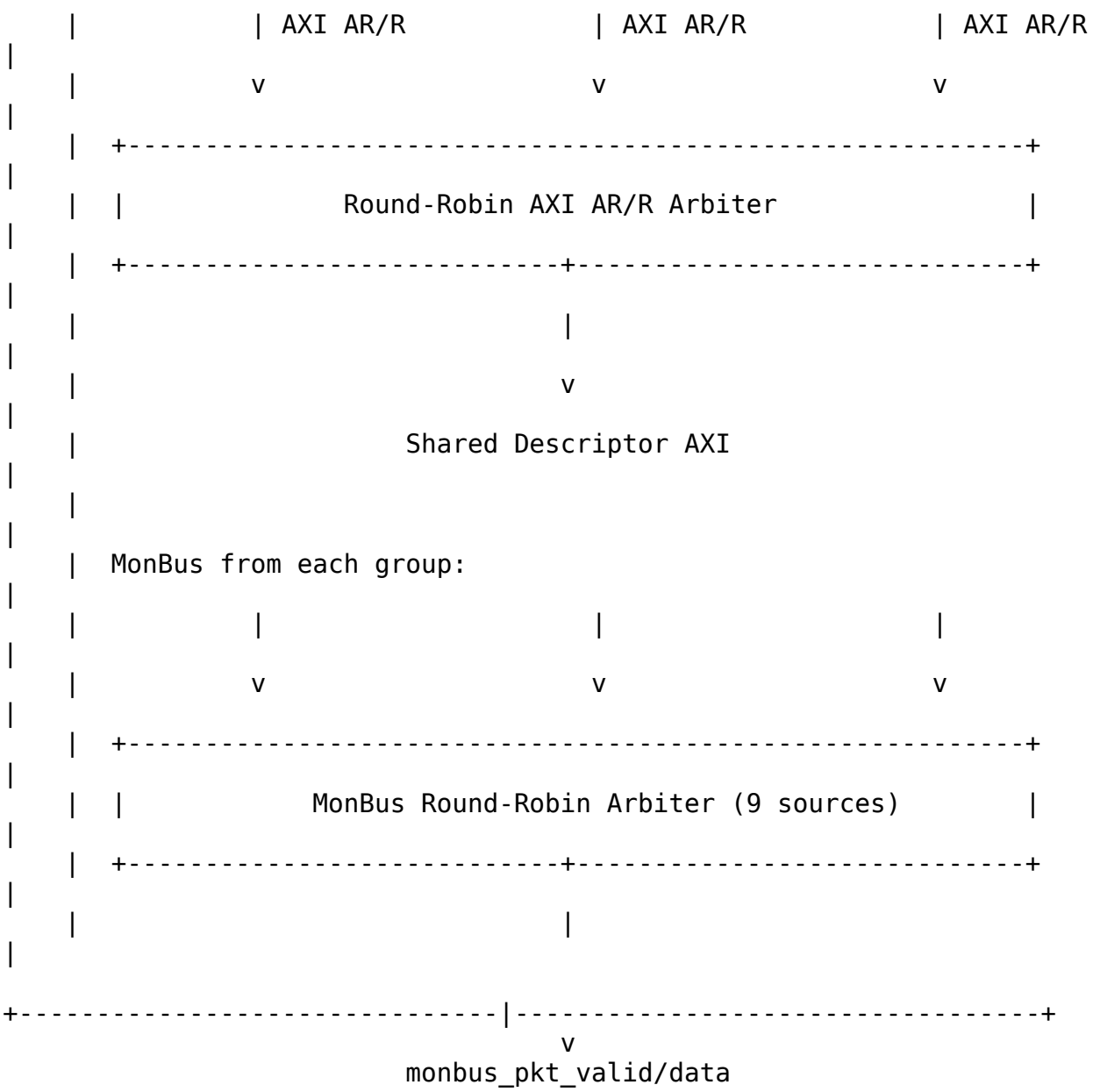
14.1.1 Key Features

- **8-Channel Array:** One scheduler_group per channel
- **Shared Descriptor AXI:** Round-robin arbitration for descriptor fetches
- **Unified MonBus:** Aggregates 9 MonBus sources (8 scheduler + 1 arbiter)
- **Per-Channel Configuration:** Individual enable/reset per channel
- **Aggregate Status:** Combined idle and error flags

14.1.2 Block Diagram

14.1.3 Figure 3.2.1: Beats Scheduler Group Array Block Diagram





Source: 03_beats_scheduler_group_array_block.mmd

14.2 Parameters

```
parameter int NUM_CHANNELS = 8;                 // Number of channels
parameter int ADDR_WIDTH = 64;                  // Address bus width
parameter int DATA_WIDTH = 512;                 // Data bus width
parameter int DESC_DATA_WIDTH = 256;            // Descriptor width

// AXI ID Management
parameter int AXI_ID_WIDTH = 8;                 // ID field width
parameter int MAX_OUTSTANDING = 8;              // Outstanding
```

```
descriptors

// Monitor Bus Parameters
parameter int MON_UNIT_ID = 1;                  // Unit ID for MonBus
```

: Table 3.2.1: Beats Scheduler Group Array Parameters

14.3 Port List

14.3.1 Clock and Reset

Table 3.2.2: Clock and Reset

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

14.3.2 Per-Channel APB Programming

Table 3.2.3: APB Programming Interface

Signal	Direction	Width	Description
`apb_valid`	input	NC	Channel kick-off (per-channel)
`apb_ready`	output	NC	Ready for kick-off
`apb_addr`	input	NC*AW	First descriptor address

14.3.3 Per-Channel Configuration

Table 3.2.4: Per-Channel Configuration

Signal	Direction	Width	Description
`cfg_channel_enable`	input	NC	Enable per channel
`cfg_channel_reset`	input	NC	Soft reset per channel
`cfg_sched_timeout_cycles`	input	NC*16	Timeout threshold

Signal	Direction	Width	Description
`cfg_sched_timeout_enable`	input	NC	Enable timeout
`cfg_desceng_enable`	input	NC	Enable descriptor engine
`cfg_desceng_prefetch`	input	NC	Enable prefetching
`cfg_desceng_fifo_thresh`	input	NC*4	Prefetch threshold
`cfg_desceng_addr0_base`	input	NC*AW	Address range 0 base
`cfg_desceng_addr0_limit`	input	NC*AW	Address range 0 limit

14.3.4 Shared Descriptor AXI Master Interface

Table 3.2.5: Shared Descriptor AXI Master Interface

Signal	Direction	Width	Description
`m_axi_arvalid`	output	1	AR channel valid
`m_axi_arready`	input	1	AR channel ready
`m_axi_araddr`	output	AW	AR address
`m_axi_arid`	output	ID_W	Transaction ID
`m_axi_arlen`	output	8	Burst length
`m_axi_arsize`	output	3	Burst size
`m_axi_arburst`	output	2	Burst type
`m_axi_rvalid`	input	1	R channel valid
`m_axi_rready`	output	1	R channel ready
`m_axi_rdata`	input	256	R data
`m_axi_rid`	input	ID_W	Response ID
`m_axi_rresp`	input	2	R response
`m_axi_rlast`	input	1	Last beat

14.3.5 Per-Channel Scheduler Data Interfaces

Table 3.2.6: Per-Channel Scheduler Data Interfaces

Signal	Direction	Width	Description
`sched_rd_valid`	output	NC	Read request per channel
`sched_rd_addr`	output	NC*AW	Read address
`sched_rd_beats`	output	NC*32	Read beats
`sched_rd_done_strobe`	input	NC	Read complete
`sched_wr_valid`	output	NC	Write request per channel
`sched_wr_addr`	output	NC*AW	Write address
`sched_wr_beats`	output	NC*32	Write beats
`sched_wr_done_strobe`	input	NC	Write complete

14.3.6 Aggregate Status

Table 3.2.7: Aggregate Status

Signal	Direction	Width	Description
`all_channels_idle`	output	1	All channels idle
`scheduler_idle`	output	NC	Per-channel scheduler idle
`descriptor_engine_idle`	output	NC	Per-channel desc eng idle
`scheduler_state`	output	NC*7	FSM states
`sched_error`	output	NC	Error flags

14.3.7 Unified MonBus Interface

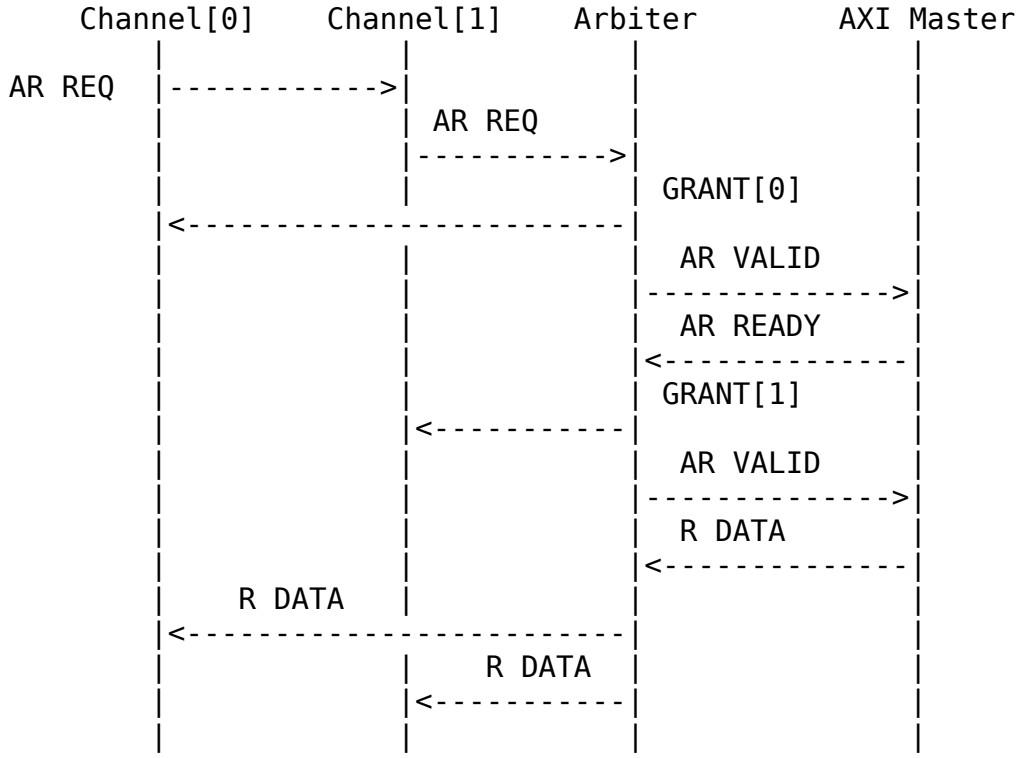
Table 3.2.8: Unified MonBus Interface

Signal	Direction	Width	Description
`monbus_pkt_valid`	output	1	Packet valid
`monbus_pkt_ready`	input	1	Consumer

Signal	Direction	Width	Description
			ready
`monbus_pkt_data`	output	64	Packet data

14.4 AXI Arbitration

14.4.1 Figure 3.2.2: Descriptor AXI Arbitration Sequence



Source: 03_scheduler_group_array_arbitration.mmd

14.5 MonBus Aggregation

The unified MonBus output aggregates 9 sources using round-robin arbitration:

Table 3.2.9: MonBus Source Assignment

Source ID	Origin	Description
0-7	scheduler_group[0:7]	Per-channel aggregated MonBus
8	AXI Arbiter	Arbitration events

Each scheduler_group provides a single MonBus output that combines both scheduler and descriptor engine packets (2:1 internal arbitration).

14.6 Integration Context

```
beats_scheduler_group_array #(
    .NUM_CHANNELS(8),
    .ADDR_WIDTH(64),
    .DATA_WIDTH(512)
) u_scheduler_array (
    .clk                    (clk),
    .rst_n                  (rst_n),

    // APB programming (per-channel)
    .apb_valid              (apb_kick_valid),
    .apb_ready              (apb_kick_ready),
    .apb_addr               (apb_kick_addr),

    // Configuration (per-channel)
    .cfg_channel_enable     (cfg_ch_enable),
    .cfg_sched_timeout_cycles(cfg_timeout_cycles),
    // ... additional config ...

    // Shared descriptor AXI
    .m_axi_arvalid          (desc_axi_arvalid),
    .m_axi_arready          (desc_axi_arready),
    .m_axi_araddr           (desc_axi_araddr),
    .m_axi_rvalid           (desc_axi_rvalid),
    .m_axi_rready           (desc_axi_rready),
    .m_axi_rdata            (desc_axi_rdata),

    // Per-channel scheduler outputs
    .sched_rd_valid         (sched_rd_valid),
    .sched_rd_addr          (sched_rd_addr),
    .sched_rd_beats         (sched_rd_beats),
    .sched_rd_done_strobe   (sched_rd_done_strobe),
    .sched_wr_valid         (sched_wr_valid),
    .sched_wr_addr          (sched_wr_addr),
    .sched_wr_beats         (sched_wr_beats),
    .sched_wr_done_strobe   (sched_wr_done_strobe),

    // Status
    .all_channels_idle      (all_schedulers_idle),

    // Unified MonBus
    .monbus_pkt_valid       (sched_array_monbus_valid),
    .monbus_pkt_ready       (sched_array_monbus_ready),
```

```
    .monbus_pkt_data        (sched_array_monbus_data)
);
```

Last Updated: 2025-01-10

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

---

15 Sink Data Path Specification

Module: `sink_data_path.sv` **Location:** `projects/components/rapids/rtl/macro_beats/` **Status:** Implemented **Last Updated:** 2025-01-10

15.1 Overview

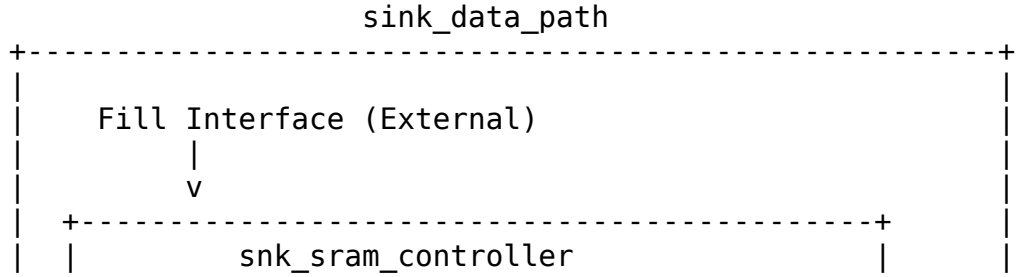
The Sink Data Path integrates the SRAM controller and AXI write engine for network-to-memory transfers. Data enters through the fill interface, is buffered in SRAM, and is written to system memory via AXI4.

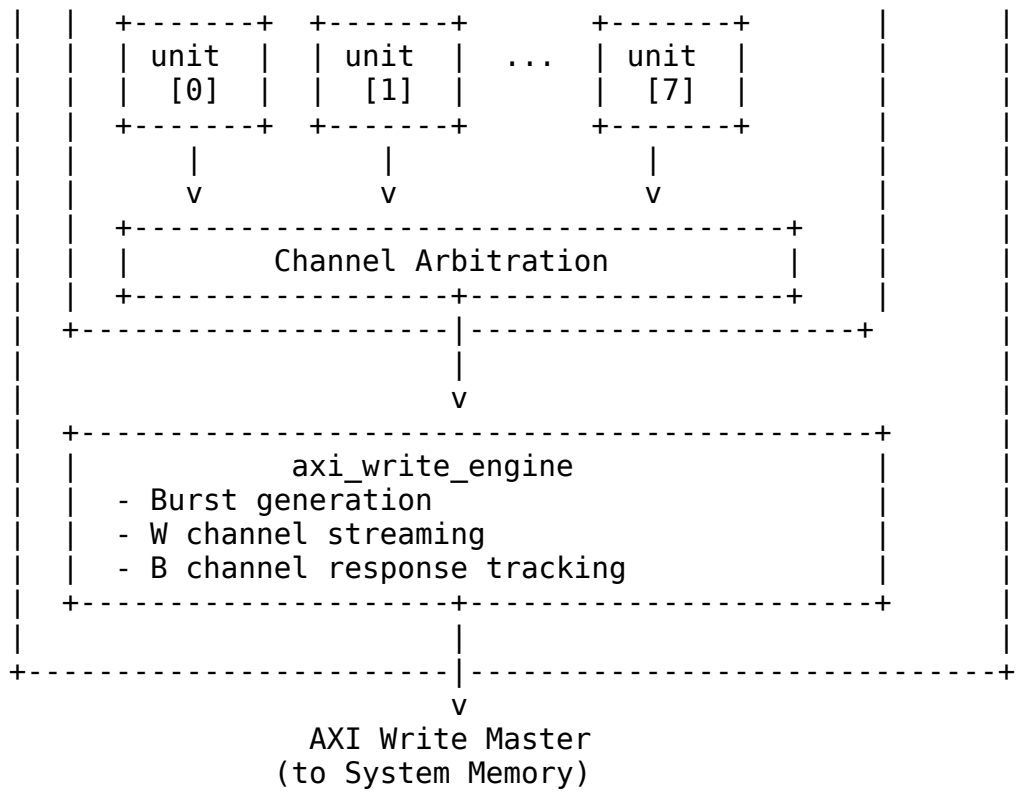
15.1.1 Key Features

- **8-Channel SRAM Controller:** Per-channel buffering with flow control
- **AXI Write Engine:** Streaming burst writes to system memory
- **Beat-Level Tracking:** Precise flow control at beat granularity
- **Scheduler Integration:** Receives transfer requests from scheduler

15.1.2 Block Diagram

15.1.3 Figure 3.3.1: Sink Data Path Block Diagram





Source: 03_sink_data_path_block.mmd

15.2 Parameters

```
parameter int NUM_CHANNELS = 8;
parameter int ADDR_WIDTH = 64;
parameter int DATA_WIDTH = 512;
parameter int AXI_ID_WIDTH = 8;
parameter int SRAM_DEPTH = 512;
parameter int AW_MAX_OUTSTANDING = 8;
parameter int W_PHASE_FIFO_DEPTH = 64;
parameter int B_PHASE_FIFO_DEPTH = 16;
```

: Table 3.3.1: Sink Data Path Parameters

15.3 Data Flow

15.3.1 Figure 3.3.2: Sink Path Data Flow Sequence

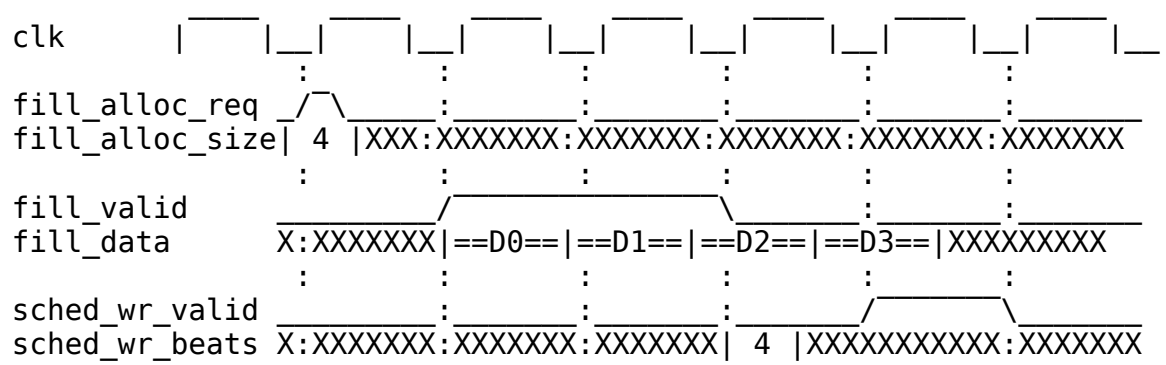
```
1. Fill Request
   - snk_fill_alloc_req = 1
   - snk_fill_alloc_size = N (beats to allocate)
   - snk_fill_alloc_id = channel
```

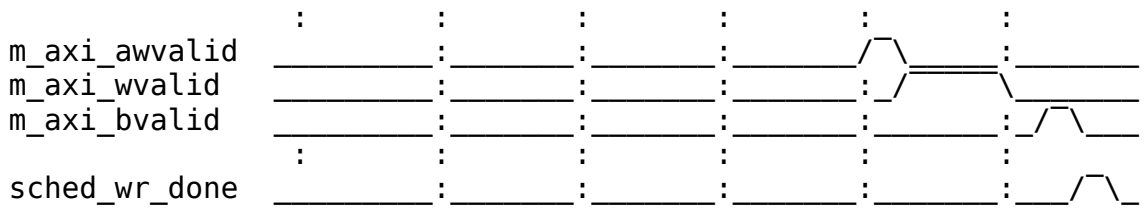
```
        |
        v
2. Space Check (beats_alloc_ctrl)
   - Check snk_fill_space_free[channel] >= N
   - Allocate N beats of space
        |
        v
3. Fill Data Transfer
   - snk_fill_valid = 1
   - snk_fill_data = data beats
   - snk_fill_id = channel
        |
        v
4. SRAM Write
   - Data written to channel's SRAM partition
   - beats_drain_ctrl notified of data availability
        |
        v
5. AXI Write Request
   - Scheduler requests write: sched_wr_valid
   - Address and beat count provided
        |
        v
6. SRAM Read + AXI Write
   - axi_write_engine reads from SRAM
   - Issues AXI bursts to memory
        |
        v
7. Completion
   - B channel response received
   - sched_wr_done_strobe = 1
```

Source: 03_sink_data_flow.mmd

15.4 Timing Diagram

15.4.1 Figure 3.3.3: Sink Path Transfer Timing





TODO: Replace with simulation-generated waveform showing complete sink transfer

---

15.5 Interface Summary

15.5.1 Fill Interface (Input)

Table 3.3.2: Fill Interface

Signal	Direction	Description
`snk_fill_alloc_req`	input	Allocation request
`snk_fill_alloc_size`	input	Beats to allocate
`snk_fill_alloc_id`	input	Channel ID
`snk_fill_space_free`	output	Available space per channel
`snk_fill_valid`	input	Data valid
`snk_fill_ready`	output	Ready for data
`snk_fill_data`	input	Fill data
`snk_fill_last`	input	Last beat
`snk_fill_id`	input	Channel ID

15.5.2 Scheduler Interface

Table 3.3.3: Scheduler Interface

Signal	Direction	Description
`sched_wr_valid`	input	Write request
`sched_wr_addr`	input	Destination address
`sched_wr_beats`	input	Beats to write
`sched_wr_id`	input	Channel ID
`sched_wr_done_strobe`	output	Write complete
`sched_wr_beats_don`	output	Beats completed

Signal	Direction	Description
e sched_wr_error	output	Error flag

Last Updated: 2025-01-10

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

---

16 Sink Data Path AXIS Wrapper Specification

Module: `sink_data_path_axis.sv` **Location:** `projects/components/rapids/rtl/macro_beats/` **Status:** Implemented **Last Updated:** 2025-01-10

16.1 Overview

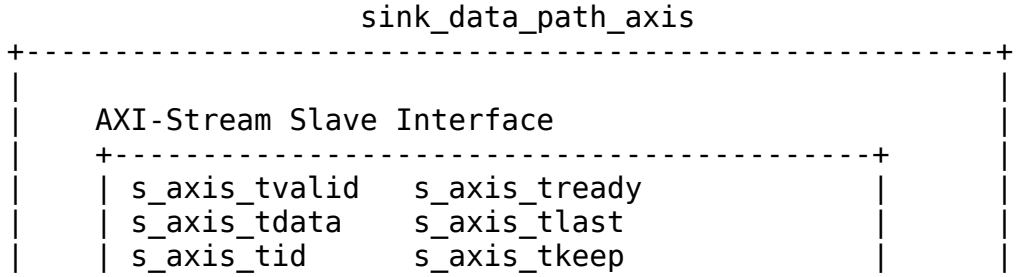
The Sink Data Path AXIS wrapper adds an AXI-Stream slave interface to the sink data path, enabling standard AXIS ingress for network-to-memory transfers.

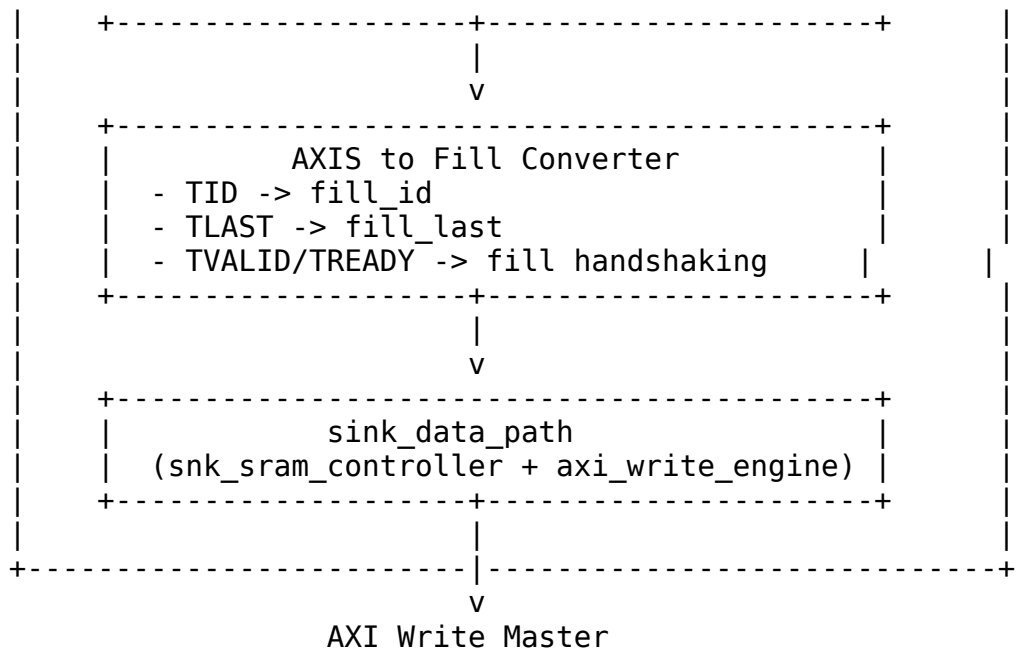
16.1.1 Key Features

- **AXI-Stream Slave Interface:** Standard AXIS for data ingress
- **Per-Channel TID Mapping:** AXIS TID maps to fill channel ID
- **TLAST to Last Mapping:** Packet boundary tracking
- **Core Sink Integration:** Wraps sink_data_path module

16.1.2 Block Diagram

16.1.3 Figure 3.4.1: Sink Data Path AXIS Block Diagram





Source: 03_sink_data_path_axis_block.mmd

16.2 Parameters

```
parameter int NUM_CHANNELS = 8;
parameter int ADDR_WIDTH = 64;
parameter int DATA_WIDTH = 512;
parameter int AXI_ID_WIDTH = 8;
parameter int SRAM_DEPTH = 512;
parameter int TID_WIDTH = 3;                    // log2(NUM_CHANNELS)
parameter int TDEST_WIDTH = 1;
parameter int TUSER_WIDTH = 1;
```

: Table 3.4.1: Sink Data Path AXIS Parameters

16.3 Port List

16.3.1 Clock and Reset

Table 3.4.2: Clock and Reset

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

16.3.2 AXI-Stream Slave Interface

Table 3.4.3: AXI-Stream Slave Interface

Signal	Direction	Width	Description
`s_axis_tvalid`	input	1	Data valid
`s_axis_tready`	output	1	Ready for data
`s_axis_tdata`	input	DATA_WIDTH	Data payload
`s_axis_tkeep`	input	DATA_WIDTH/8	Byte enables
`s_axis_tlast`	input	1	Last beat of packet
`s_axis_tid`	input	TID_WIDTH	Stream ID (channel)
`s_axis_tdest`	input	TDEST_WIDTH	Destination
`s_axis_tuser`	input	TUSER_WIDTH	User sideband

16.3.3 Space Allocation Interface

Table 3.4.4: Space Allocation Interface

Signal	Direction	Width	Description
`snk_alloc_req`	input	1	Allocation request
`snk_alloc_size`	input	16	Beats to allocate
`snk_alloc_id`	input	3	Channel ID
`snk_space_free`	output	NC*16	Available space per channel

16.3.4 Scheduler Interface

Table 3.4.5: Scheduler Interface

Signal	Direction	Width	Description
`sched_wr_valid`	input	1	Write request
`sched_wr_addr`	input	AW	Destination address
`sched_wr_beats`	input	32	Beats to write

Signal	Direction	Width	Description
sched_wr_id	input	3	Channel ID
sched_wr_done_strobe	output	1	Write complete
sched_wr_beats_done	output	32	Beats completed
sched_wr_error	output	1	Error flag

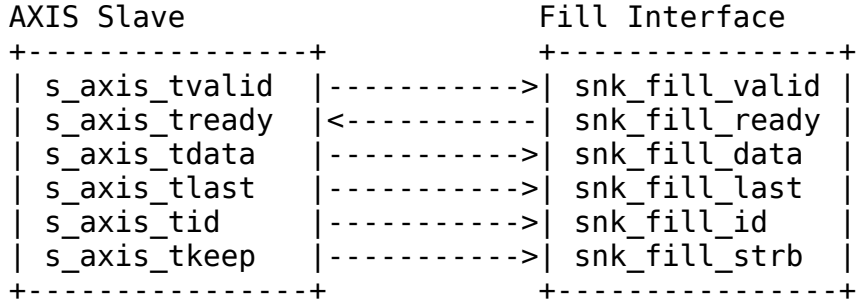
16.3.5 AXI Write Master Interface

Table 3.4.6: AXI Write Master Interface

Signal	Direction	Width	Description
m_axi_awvalid	output	1	AW channel valid
m_axi_awready	input	1	AW channel ready
m_axi_awaddr	output	AW	Write address
m_axi_awlen	output	8	Burst length
m_axi_awsize	output	3	Burst size
m_axi_awburst	output	2	Burst type
m_axi_awid	output	ID_W	Transaction ID
m_axi_wvalid	output	1	W channel valid
m_axi_wready	input	1	W channel ready
m_axi_wdata	output	DW	Write data
m_axi_wstrb	output	DW/8	Write strobes
m_axi_wlast	output	1	Last beat
m_axi_bvalid	input	1	B channel valid
m_axi_bready	output	1	B channel ready
m_axi_bresp	input	2	Write response
m_axi_bid	input	ID_W	Response ID

16.4 Signal Mapping

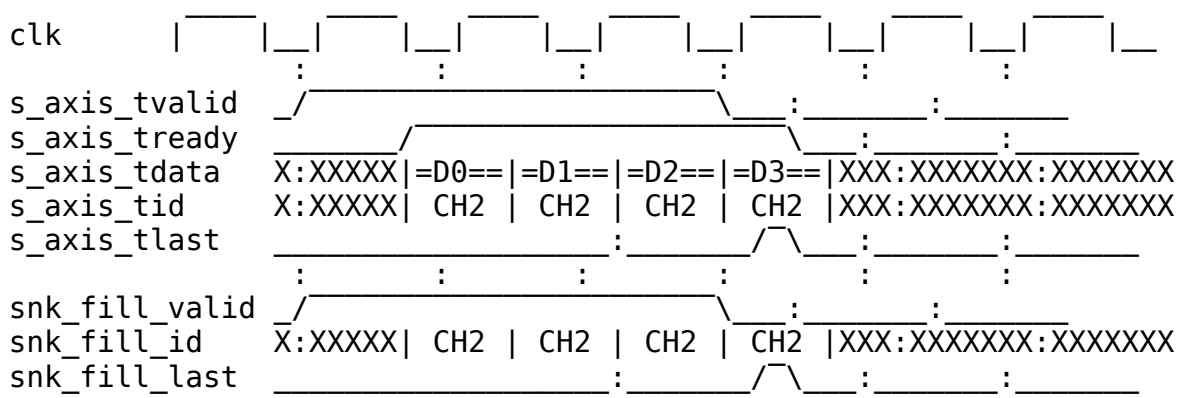
16.4.1 Figure 3.4.2: AXIS to Fill Interface Mapping



Source: 03_sink_axis_mapping.mmd

16.5 Timing Diagram

16.5.1 Figure 3.4.3: AXIS Ingress Timing



TODO: Replace with simulation-generated waveform showing AXIS packet reception

---

16.6 Usage Notes

1. **TID Usage:** AXIS TID directly maps to channel ID for multi-channel routing
2. **Allocation Required:** Space must be allocated via `snk_alloc_*` before AXIS data arrives
3. **Backpressure:** `s_axis_tready` deasserts when SRAM space exhausted
4. **Packet Boundaries:** TLAST marks end of AXIS packets, mapped to fill_last

---

Last Updated: 2025-01-10

---

17 Sink SRAM Controller Specification

Module: `snk_sram_controller.sv` **Location:** `projects/components/rapids/rtl/macro_beats/` **Status:** Implemented **Last Updated:** 2025-01-10

17.1 Overview

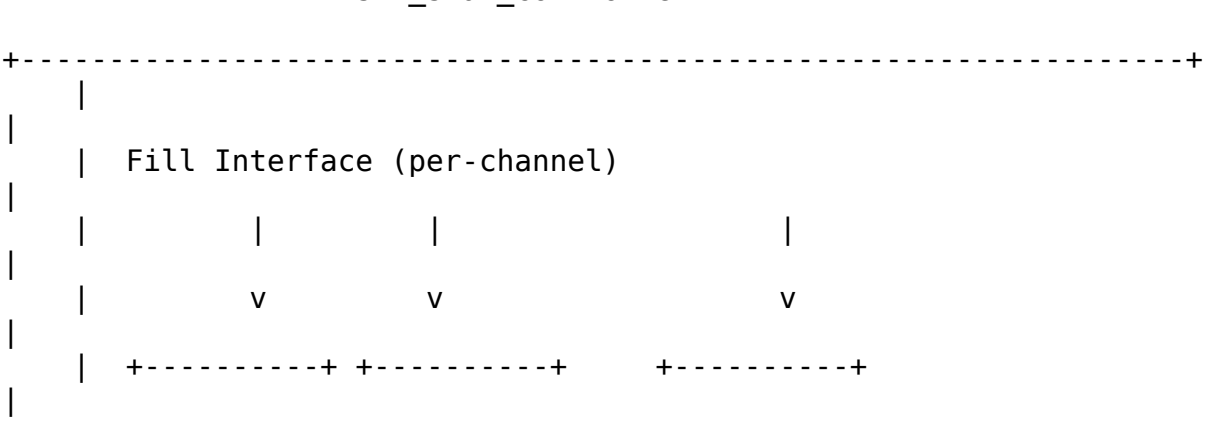
The Sink SRAM Controller manages 8 SRAM controller units, providing per-channel buffering with channel arbitration for the shared AXI write engine.

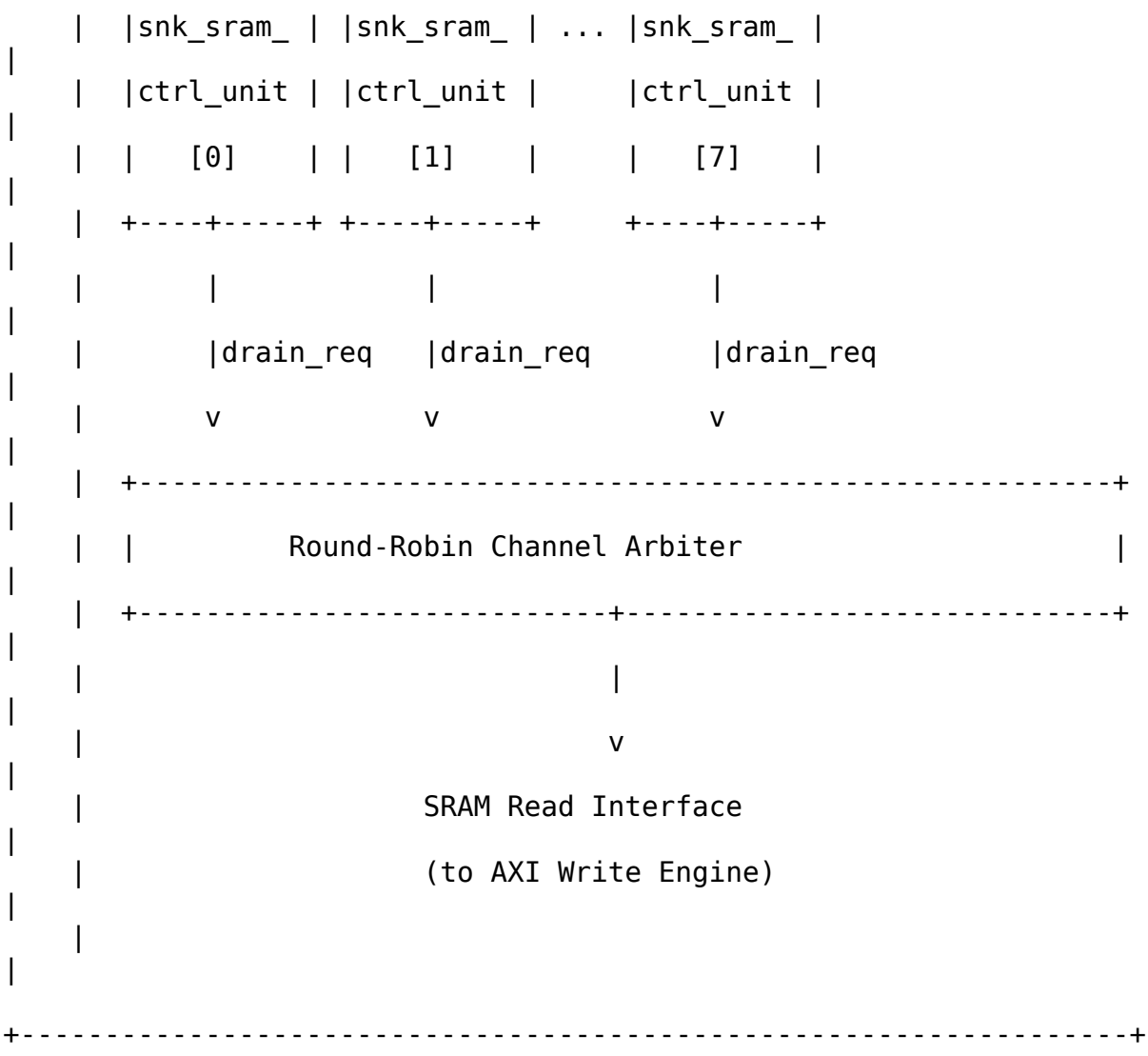
17.1.1 Key Features

- **8-Channel SRAM Array:** Instantiates 8 `snk_sram_controller_unit` modules
- **Channel Arbitration:** Round-robin access to shared AXI write engine
- **Flow Control Integration:** Per-channel alloc_ctrl and drain_ctrl
- **Space Tracking:** Reports available space per channel

17.1.2 Block Diagram

17.1.3 Figure 3.5.1: Sink SRAM Controller Block Diagram





Source: 03_snk_sram_controller_block.mmd

17.2 Parameters

```
parameter int NUM_CHANNELS = 8;
parameter int DATA_WIDTH = 512;
parameter int SRAM_DEPTH = 512;                 // Per-channel depth
parameter int ADDR_WIDTH = $clog2(SRAM_DEPTH);
```

: Table 3.5.1: Sink SRAM Controller Parameters

17.3 Port List

17.3.1 Clock and Reset

Table 3.5.2: Clock and Reset

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

17.3.2 Per-Channel Fill Interface

Table 3.5.3: Per-Channel Fill Interface

Signal	Direction	Width	Description
`snk_fill_alloc_req`	input	NC	Allocation request per channel
`snk_fill_alloc_size`	input	NC*16	Beats to allocate
`snk_fill_space_free`	output	NC*16	Available space per channel
`snk_fill_valid`	input	NC	Fill data valid
`snk_fill_ready`	output	NC	Ready for fill data
`snk_fill_data`	input	NC*DW	Fill data
`snk_fill_last`	input	NC	Last beat marker

17.3.3 Arbitrated Drain Interface (to AXI Write Engine)

Table 3.5.4: Arbitrated Drain Interface

Signal	Direction	Width	Description
`drain_req`	output	1	Drain request (any channel)
`drain_gnt`	input	1	Drain grant
`drain_id`	output	3	Granted channel ID

Signal	Direction	Width	Description
`drain_beats`	output	16	Beats available to drain
`sram_rd_en`	input	1	SRAM read enable
`sram_rd_addr`	input	AW	SRAM read address
`sram_rd_data`	output	DW	SRAM read data

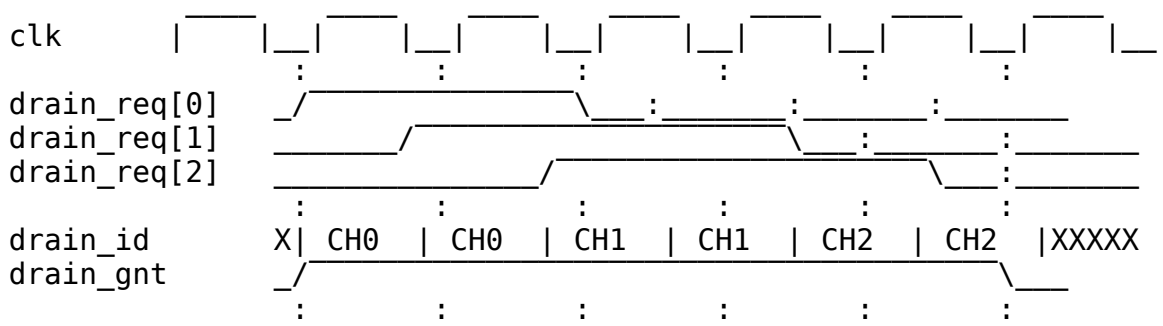
17.3.4 Per-Channel Status

Table 3.5.5: Per-Channel Status

Signal	Direction	Width	Description
`channel_data_avail`	output	NC	Data available per channel
`channel_empty`	output	NC	Channel empty flags
`channel_full`	output	NC	Channel full flags

17.4 Arbitration Logic

17.4.1 Figure 3.5.2: Channel Arbitration Timing



TODO: Replace with simulation-generated waveform showing round-robin arbitration

---

17.5 Integration with AXI Write Engine

```
snk_sram_controller #(
    .NUM_CHANNELS(8),
    .DATA_WIDTH(512),
    .SRAM_DEPTH(512)
) u_snk_sram_ctrl (
    .clk                    (clk),
    .rst_n                  (rst_n),

    // Per-channel fill interface
    .snk_fill_alloc_req     (fill_alloc_req),
    .snk_fill_alloc_size    (fill_alloc_size),
    .snk_fill_space_free    (fill_space_free),
    .snk_fill_valid         (fill_valid),
    .snk_fill_ready         (fill_ready),
    .snk_fill_data          (fill_data),
    .snk_fill_last          (fill_last),

    // Arbitrated drain interface
    .drain_req              (sram_drain_req),
    .drain_gnt              (sram_drain_gnt),
    .drain_id               (sram_drain_id),
    .drain_beats            (sram_drain_beats),
    .sram_rd_en             (sram_rd_en),
    .sram_rd_addr           (sram_rd_addr),
    .sram_rd_data           (sram_rd_data),

    // Status
    .channel_data_avail     (channel_data_avail),
    .channel_empty          (channel_empty),
    .channel_full           (channel_full)
);
```

Last Updated: 2025-01-10

---

18 Sink SRAM Controller Unit Specification

Module: `snk_sram_controller_unit.sv` **Location:** `projects/components/rapids/rtl/macro_beats/` **Status:** Implemented **Last Updated:** 2025-01-10

18.1 Overview

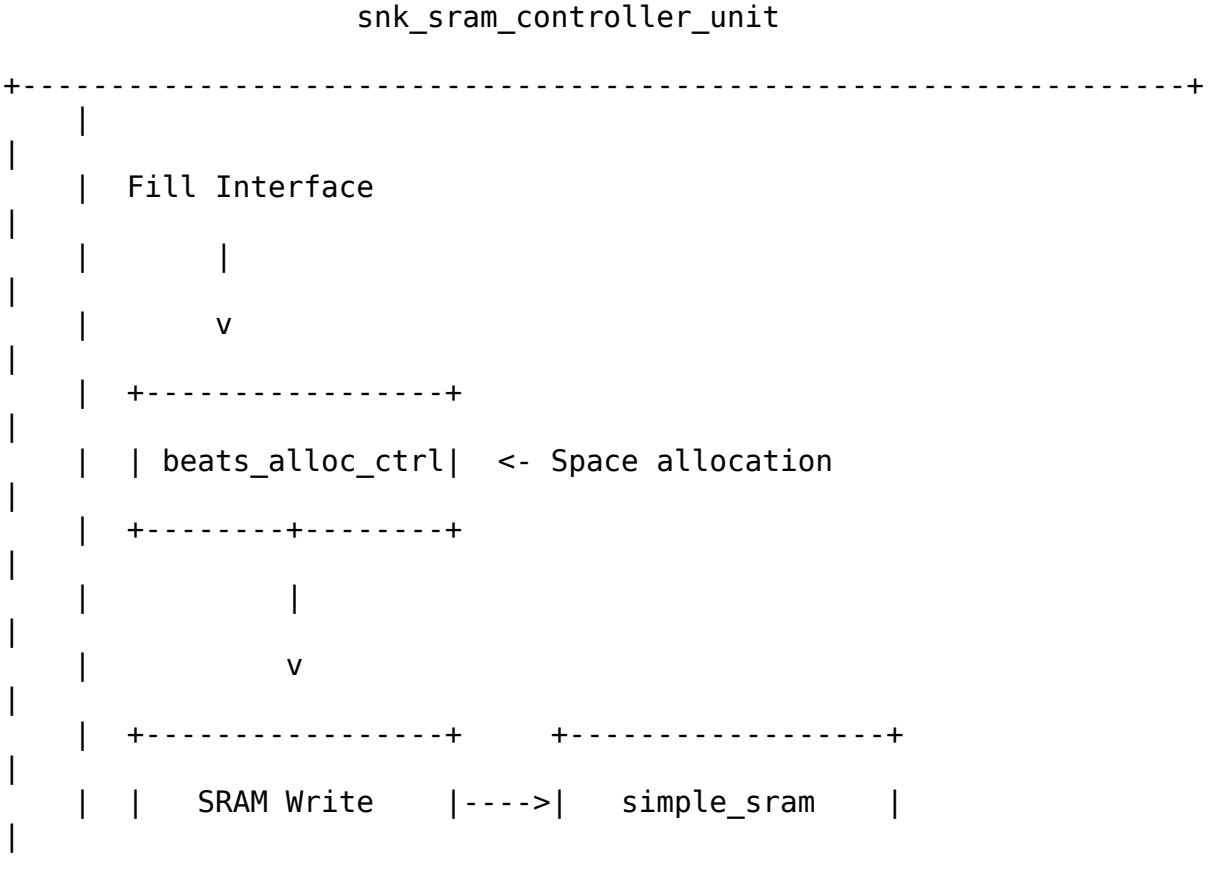
The Sink SRAM Controller Unit manages a single channel's SRAM buffer with beat-level flow control using virtual FIFOs (alloc_ctrl and drain_ctrl).

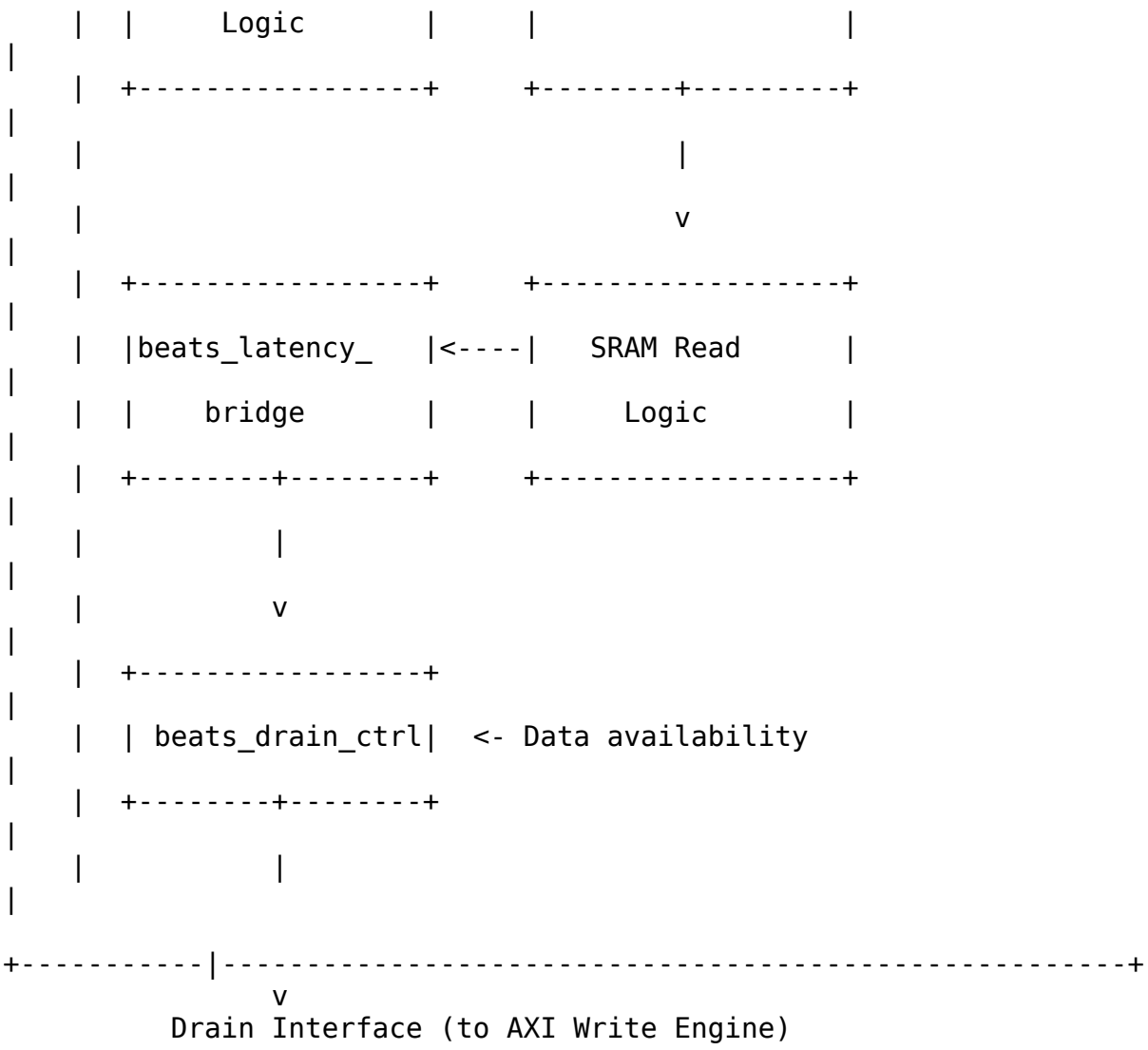
18.1.1 Key Features

- **Single-Channel SRAM:** Dedicated SRAM buffer for one channel
- **Virtual FIFO Flow Control:** beats_alloc_ctrl + beats_drain_ctrl
- **Latency Bridge:** Compensates for pipeline timing
- **Simple SRAM:** No reset on memory, only on control logic

18.1.2 Block Diagram

18.1.3 Figure 3.6.1: Sink SRAM Controller Unit Block Diagram





Source: 03_snk_sram_controller_unit_block.mmd

18.2 Parameters

```
parameter int DATA_WIDTH = 512;
parameter int SRAM_DEPTH = 512;
parameter int ADDR_WIDTH = $clog2(SRAM_DEPTH);
parameter int LATENCY_BRIDGE_DEPTH = 4;
```

: Table 3.6.1: Sink SRAM Controller Unit Parameters

18.3 Port List

18.3.1 Clock and Reset

Table 3.6.2: Clock and Reset

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

18.3.2 Fill Interface (from Network)

Table 3.6.3: Fill Interface

Signal	Direction	Width	Description
`fill_alloc_req`	input	1	Allocate space
`fill_alloc_size`	input	16	Beats to allocate
`fill_space_free`	output	16	Available space
`fill_valid`	input	1	Fill data valid
`fill_ready`	output	1	Ready for fill data
`fill_data`	input	DW	Fill data
`fill_last`	input	1	Last beat marker

18.3.3 Drain Interface (to AXI Write Engine)

Table 3.6.4: Drain Interface

Signal	Direction	Width	Description
`drain_req`	output	1	Data available to drain
`drain_gnt`	input	1	Drain grant
`drain_beats`	output	16	Beats available
`sram_rd_en`	input	1	SRAM read enable
`sram_rd_addr`	input	AW	SRAM read address

Signal	Direction	Width	Description
`sram_rd_data`	output	DW	SRAM read data

18.3.4 Status

Table 3.6.5: Status Interface

Signal	Direction	Width	Description
`data_avail`	output	1	Data available flag
`empty`	output	1	Buffer empty
`full`	output	1	Buffer full

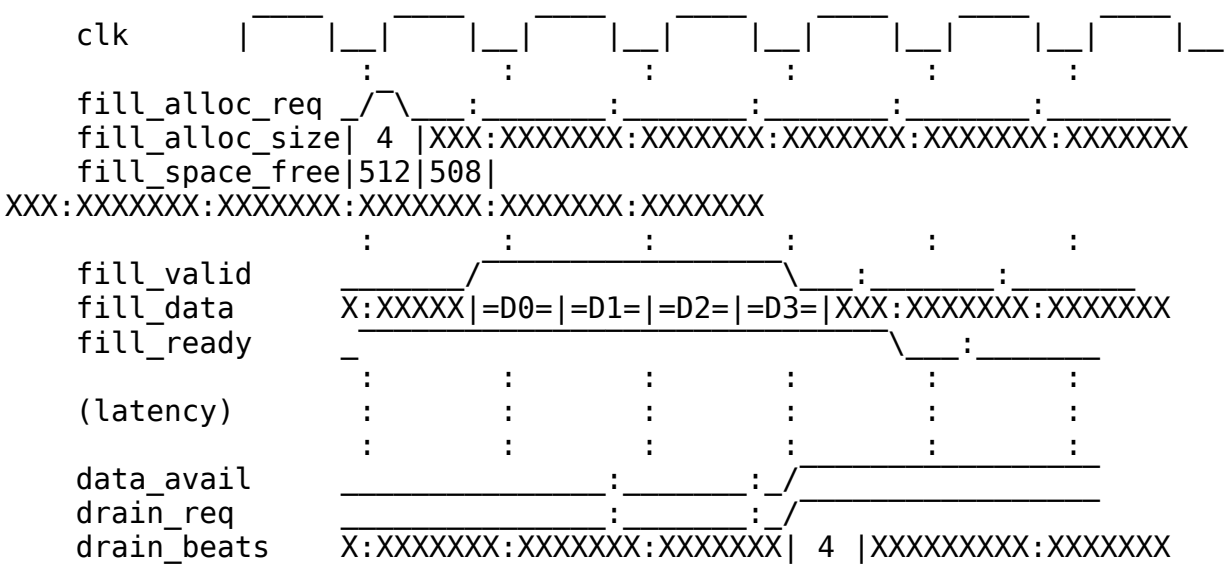
18.4 Internal Architecture

18.4.1 Figure 3.6.2: Internal Data Flow

```
Fill Path:

   fill_alloc_req -----> beats_alloc_ctrl -----> wr_ptr management
                                 |
   fill_valid/data ---------->   | -----> SRAM write
                                 |
   fill_ready <--------------    | <----- space_free > 0


Drain Path:

   SRAM read <---------------    | <----- sram_rd_en
        |                        |
        v                        |
   beats_latency_bridge ------+-----> timing compensation
        |                     |
        v                     |
   beats_drain_ctrl ----------+-----> data_avail tracking
        |
        v
   drain_req -----------------> channel arbiter
```

18.5 Timing Diagram

18.5.1 Figure 3.6.3: Fill and Drain Timing



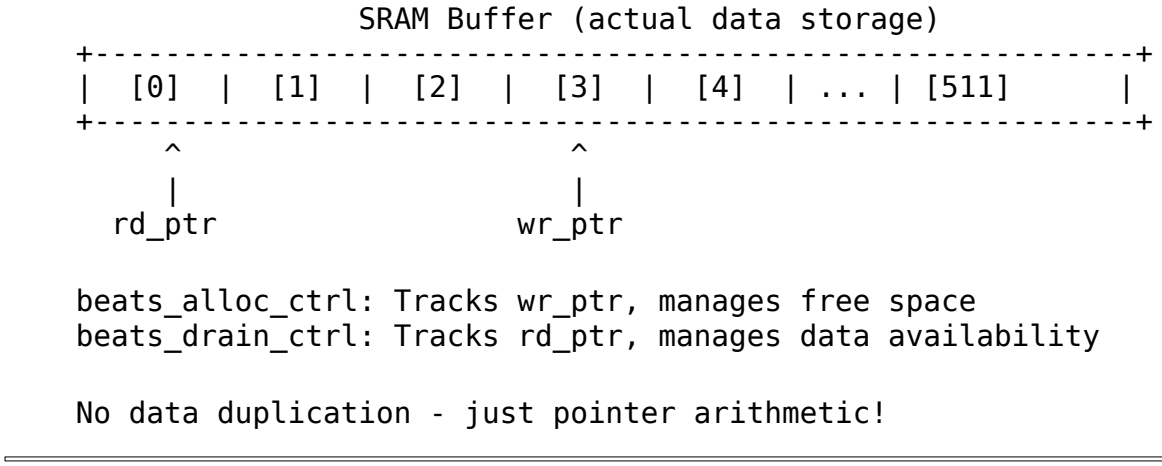
TODO: Replace with simulation-generated waveform showing complete fill-to-drain flow

---

18.6 Virtual FIFO Concept

The controller uses “virtual FIFOs” - pointer-based flow control without duplicating data storage:

18.6.1 Figure 3.6.4: Virtual FIFO Operation



18.7 Integration Example

```
snk_sram_controller_unit #(
    .DATA_WIDTH(512),
    .SRAM_DEPTH(512),
    .LATENCY_BRIDGE_DEPTH(4)
) u_snk_sram_unit (
    .clk                (clk),
    .rst_n              (rst_n),

    // Fill interface
    .fill_alloc_req     (fill_alloc_req[ch]),
    .fill_alloc_size    (fill_alloc_size[ch]),
    .fill_space_free    (fill_space_free[ch]),
    .fill_valid         (fill_valid[ch]),
    .fill_ready         (fill_ready[ch]),
    .fill_data          (fill_data[ch]),
    .fill_last          (fill_last[ch]),

    // Drain interface
    .drain_req          (drain_req[ch]),
    .drain_gnt          (drain_gnt[ch]),
    .drain_beats        (drain_beats[ch]),
    .sram_rd_en         (sram_rd_en[ch]),
    .sram_rd_addr       (sram_rd_addr[ch]),
    .sram_rd_data       (sram_rd_data[ch]),

    // Status
    .data_avail         (data_avail[ch]),
    .empty              (empty[ch]),
    .full               (full[ch])
);
```

Last Updated: 2025-01-10

---

19 Source Data Path Specification

Module: `source_data_path.sv` **Location:** `projects/components/rapids/rtl/macro_beats/` **Status:** Implemented **Last Updated:** 2025-01-10

19.1 Overview

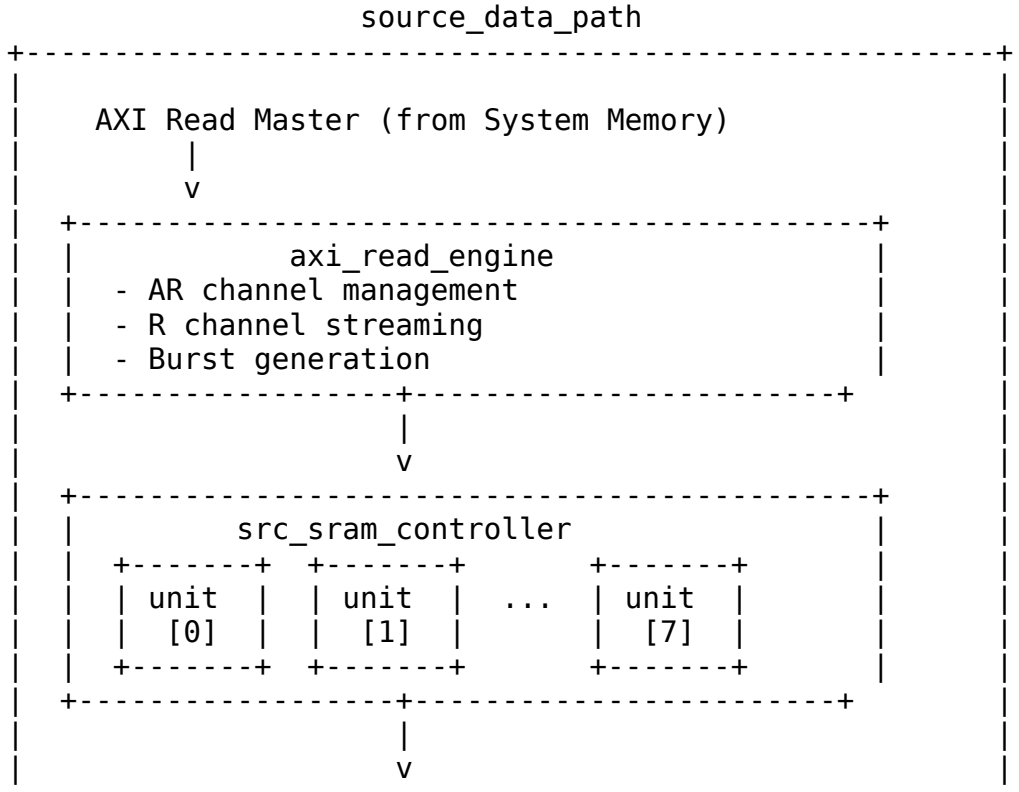
The Source Data Path integrates the AXI read engine and SRAM controller for memory-to-network transfers. Data is read from system memory via AXI4, buffered in SRAM, and delivered through the drain interface to the network.

19.1.1 Key Features

- **AXI Read Engine:** Streaming burst reads from system memory
- **8-Channel SRAM Controller:** Per-channel buffering with flow control
- **Beat-Level Tracking:** Precise flow control at beat granularity
- **Scheduler Integration:** Receives transfer requests from scheduler

19.1.2 Block Diagram

19.1.3 Figure 3.7.1: Source Data Path Block Diagram



```
    +---------------------|---------------------------------+
                          |
                  Drain Interface
                (to Network Master)
```

Source: 03_source_data_path_block.mmd

19.2 Parameters

```
parameter int NUM_CHANNELS = 8;
parameter int ADDR_WIDTH = 64;
parameter int DATA_WIDTH = 512;
parameter int AXI_ID_WIDTH = 8;
parameter int SRAM_DEPTH = 512;
parameter int AR_MAX_OUTSTANDING = 8;
parameter int R_PHASE_FIFO_DEPTH = 64;
```

: Table 3.7.1: Source Data Path Parameters

19.3 Data Flow

19.3.1 Figure 3.7.2: Source Path Data Flow Sequence

```
1. Scheduler Read Request
   - sched_rd_valid = 1
   - sched_rd_addr = source address
   - sched_rd_beats = N
   - sched_rd_id = channel
        |
        v
2. Space Check (beats_alloc_ctrl)
   - Check src_drain_space_free[channel] >= N
   - Allocate N beats of space
        |
        v
3. AXI Read Transaction
   - axi_read_engine issues AR
   - m_axi_araddr = sched_rd_addr
   - m_axi_arlen = calculated from beats
        |
        v
4. R Channel Data
   - Data arrives on m_axi_rdata
   - Written to channel's SRAM partition
        |
        v
5. SRAM Write Complete
```

```
   - beats_drain_ctrl notified via latency bridge
   - Data available for drain
       |
       v
6. Drain Interface
   - src_drain_valid = 1
   - Data delivered to network master
       |
       v
7. Completion
   - sched_rd_done_strobe = 1
   - Scheduler notified of completion
```

Source: 03_source_data_flow.mmd

19.4 Port List

19.4.1 Clock and Reset

Table 3.7.2: Clock and Reset

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

19.4.2 Scheduler Interface

Table 3.7.3: Scheduler Interface

Signal	Direction	Width	Description
`sched_rd_valid`	input	1	Read request
`sched_rd_addr`	input	AW	Source address
`sched_rd_beats`	input	32	Beats to read
`sched_rd_id`	input	3	Channel ID
`sched_rd_done_strobe`	output	1	Read complete
`sched_rd_beats_done`	output	32	Beats completed
`sched_rd_error`	output	1	Error flag

19.4.3 AXI Read Master Interface

Table 3.7.4: AXI Read Master Interface

Signal	Direction	Width	Description
`m_axi_arvalid`	output	1	AR channel valid
`m_axi_arready`	input	1	AR channel ready
`m_axi_araddr`	output	AW	Read address
`m_axi_arlen`	output	8	Burst length
`m_axi_arsize`	output	3	Burst size
`m_axi_arburst`	output	2	Burst type
`m_axi_arid`	output	ID_W	Transaction ID
`m_axi_rvalid`	input	1	R channel valid
`m_axi_rready`	output	1	R channel ready
`m_axi_rdata`	input	DW	Read data
`m_axi_rresp`	input	2	Read response
`m_axi_rid`	input	ID_W	Response ID
`m_axi_rlast`	input	1	Last beat

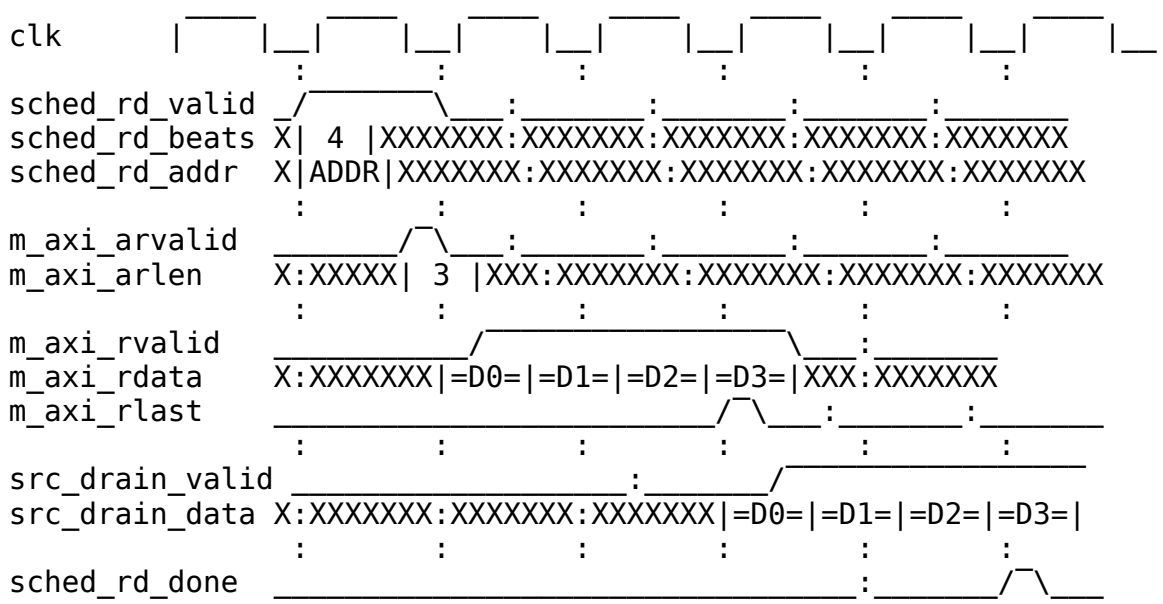
19.4.4 Drain Interface (Output to Network)

Table 3.7.5: Drain Interface

Signal	Direction	Width	Description
`src_drain_valid`	output	1	Drain data valid
`src_drain_ready`	input	1	Ready for drain data
`src_drain_data`	output	DW	Drain data
`src_drain_last`	output	1	Last beat marker
`src_drain_id`	output	3	Channel ID
`src_drain_data_avail`	output	NC*16	Available data per channel

19.5 Timing Diagram

19.5.1 Figure 3.7.3: Source Path Transfer Timing



TODO: Replace with simulation-generated waveform showing complete source transfer

---

19.6 Integration Example

```
source_data_path #(
    .NUM_CHANNELS(8),
    .ADDR_WIDTH(64),
    .DATA_WIDTH(512),
    .SRAM_DEPTH(512)
) u_source_path (
    .clk                    (clk),
    .rst_n                  (rst_n),

    // Scheduler interface
    .sched_rd_valid         (sched_rd_valid),
    .sched_rd_addr          (sched_rd_addr),
    .sched_rd_beats         (sched_rd_beats),
    .sched_rd_id            (sched_rd_id),
    .sched_rd_done_strobe   (sched_rd_done_strobe),
    .sched_rd_beats_done    (sched_rd_beats_done),
    .sched_rd_error         (sched_rd_error),
```

```
    // AXI read master
    .m_axi_arvalid          (src_axi_arvalid),
    .m_axi_arready          (src_axi_arready),
    .m_axi_araddr           (src_axi_araddr),
    .m_axi_arlen            (src_axi_arlen),
    .m_axi_rvalid           (src_axi_rvalid),
    .m_axi_rready           (src_axi_rready),
    .m_axi_rdata            (src_axi_rdata),
    .m_axi_rlast            (src_axi_rlast),

    // Drain interface
    .src_drain_valid        (src_drain_valid),
    .src_drain_ready        (src_drain_ready),
    .src_drain_data         (src_drain_data),
    .src_drain_last         (src_drain_last),
    .src_drain_id           (src_drain_id),
    .src_drain_data_avail   (src_data_avail)
);
```

Last Updated: 2025-01-10

---

20 Source Data Path AXIS Wrapper Specification

Module: `source_data_path_axis.sv` **Location:** `projects/components/rapids/rtl/macro_beats/` **Status:** Implemented **Last Updated:** 2025-01-10

20.1 Overview

The Source Data Path AXIS wrapper adds an AXI-Stream master interface to the source data path, enabling standard AXIS egress for memory-to-network transfers.

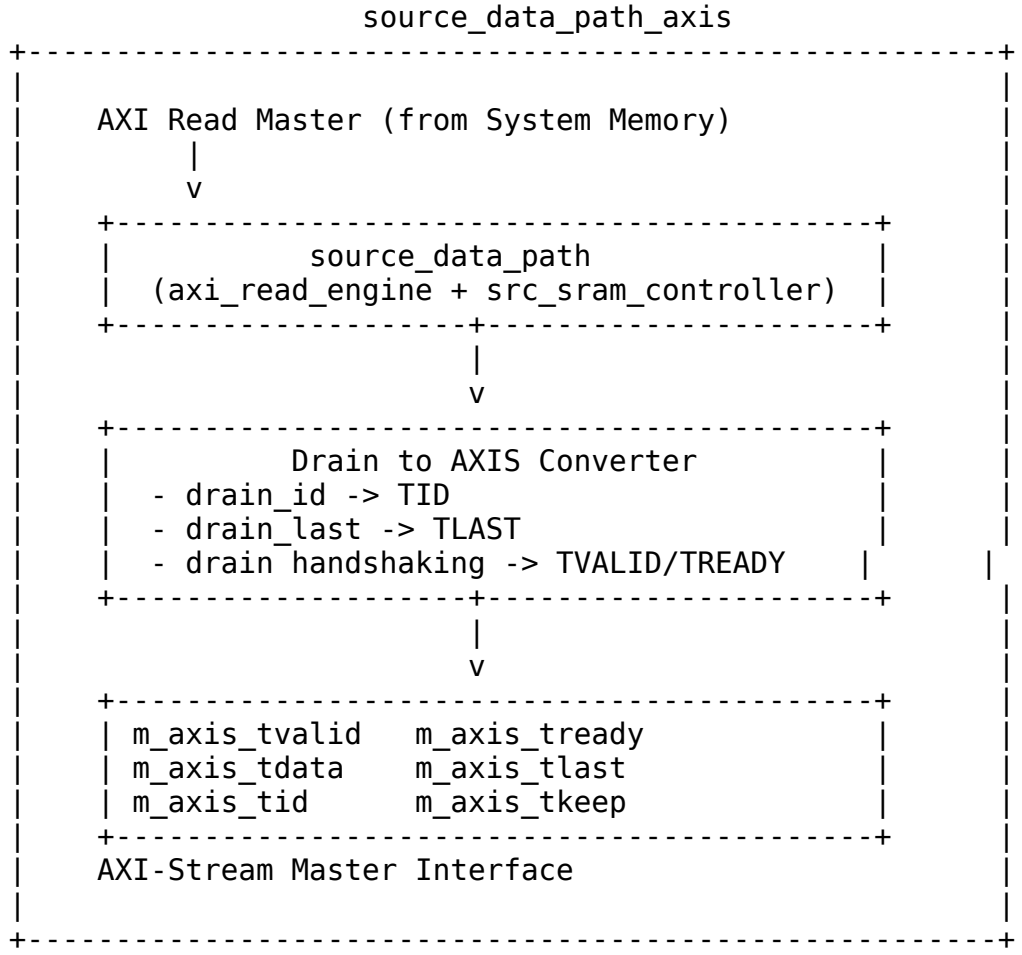
20.1.1 Key Features

- **AXI-Stream Master Interface:** Standard AXIS for data egress

- **Per-Channel TID Mapping:** Drain channel ID maps to AXIS TID
- **TLAST Generation:** Packet boundary marking
- **Core Source Integration:** Wraps source_data_path module

20.1.2 Block Diagram

20.1.3 Figure 3.8.1: Source Data Path AXIS Block Diagram



Source: 03_source_data_path_axis_block.mmd

20.2 Parameters

```
parameter int NUM_CHANNELS = 8;
parameter int ADDR_WIDTH = 64;
parameter int DATA_WIDTH = 512;
parameter int AXI_ID_WIDTH = 8;
parameter int SRAM_DEPTH = 512;
parameter int TID_WIDTH = 3;                    // log2(NUM_CHANNELS)
parameter int TDEST_WIDTH = 1;
parameter int TUSER_WIDTH = 1;
```

: Table 3.8.1: Source Data Path AXIS Parameters

20.3 Port List

20.3.1 Clock and Reset

Table 3.8.2: Clock and Reset

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

20.3.2 Scheduler Interface

Table 3.8.3: Scheduler Interface

Signal	Direction	Width	Description
`sched_rd_valid`	input	1	Read request
`sched_rd_addr`	input	AW	Source address
`sched_rd_beats`	input	32	Beats to read
`sched_rd_id`	input	3	Channel ID
`sched_rd_done_strobe`	output	1	Read complete
`sched_rd_beats_done`	output	32	Beats completed
`sched_rd_error`	output	1	Error flag

20.3.3 AXI Read Master Interface

Table 3.8.4: AXI Read Master Interface

Signal	Direction	Width	Description
`m_axi_arvalid`	output	1	AR channel valid
`m_axi_arready`	input	1	AR channel ready
`m_axi_araddr`	output	AW	Read address
`m_axi_arlen`	output	8	Burst length

Signal	Direction	Width	Description
m_axi_arsize	output	3	Burst size
m_axi_arburst	output	2	Burst type
m_axi_arid	output	ID_W	Transaction ID
m_axi_rvalid	input	1	R channel valid
m_axi_rready	output	1	R channel ready
m_axi_rdata	input	DW	Read data
m_axi_rresp	input	2	Read response
m_axi_rid	input	ID_W	Response ID
m_axi_rlast	input	1	Last beat

20.3.4 AXI-Stream Master Interface

Table 3.8.5: AXI-Stream Master Interface

Signal	Direction	Width	Description
m_axis_tvalid	output	1	Data valid
m_axis_tready	input	1	Ready for data
m_axis_tdata	output	DATA_WIDTH	Data payload
m_axis_tkeep	output	DATA_WIDTH/8	Byte enables
m_axis_tlast	output	1	Last beat of packet
m_axis_tid	output	TID_WIDTH	Stream ID (channel)
m_axis_tdest	output	TDEST_WIDTH	Destination
m_axis_tuser	output	TUSER_WIDTH	User sideband

20.4 Signal Mapping

20.4.1 Figure 3.8.2: Drain to AXIS Interface Mapping

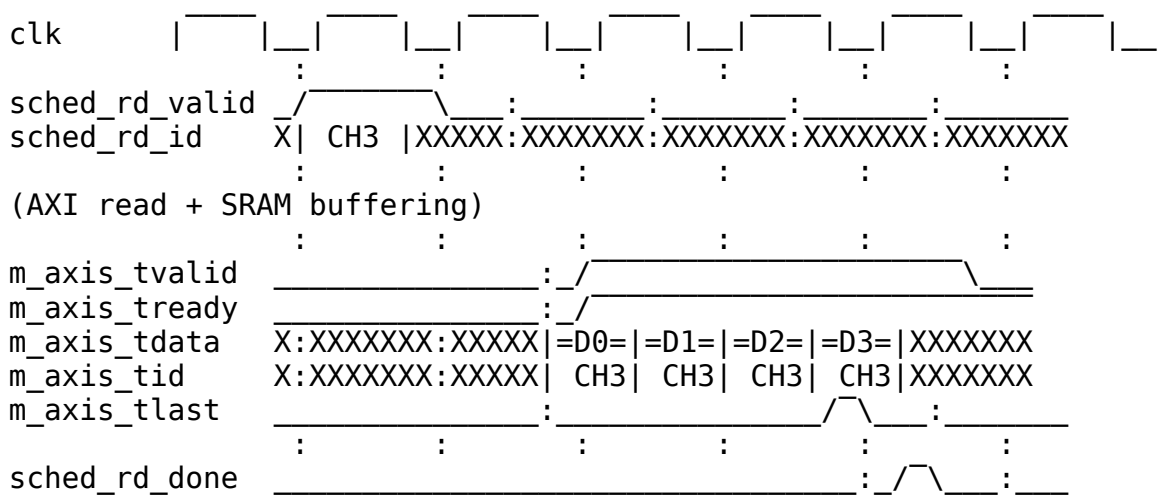
```
    Drain Interface               AXIS Master
    +----------------+            +----------------+
    | src_drain_valid|----------->| m_axis_tvalid  |
    | src_drain_ready|<-----------| m_axis_tready  |
    | src_drain_data |----------->| m_axis_tdata   |
```

```
    | src_drain_last |----------->| m_axis_tlast   |
    | src_drain_id   |----------->| m_axis_tid     |
    | src_drain_strb |----------->| m_axis_tkeep   |
    +----------------+            +----------------+
```

Source: 03_source_axis_mapping.mmd

20.5 Timing Diagram

20.5.1 Figure 3.8.3: AXIS Egress Timing



TODO: Replace with simulation-generated waveform showing AXIS packet transmission

---

20.6 Usage Notes

1. **TID Generation:** Drain channel ID directly maps to AXIS TID
2. **TLAST Generation:** Drain last signal maps to AXIS TLAST
3. **Backpressure:** Network TREADY backpressure propagates to SRAM drain
4. **Packet Boundaries:** TLAST marks end of AXIS packets

20.7 Integration Example

```
source_data_path_axis #(
    .NUM_CHANNELS(8),
    .ADDR_WIDTH(64),
    .DATA_WIDTH(512),
    .TID_WIDTH(3)
) u_source_axis (
```

```
    .clk                    (clk),
    .rst_n                  (rst_n),

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

    // AXI read master
    .m_axi_arvalid          (src_axi_arvalid),
    .m_axi_arready          (src_axi_arready),
    .m_axi_araddr           (src_axi_araddr),
    .m_axi_rvalid           (src_axi_rvalid),
    .m_axi_rready           (src_axi_rready),
    .m_axi_rdata            (src_axi_rdata),

    // AXIS master
    .m_axis_tvalid          (network_tvalid),
    .m_axis_tready          (network_tready),
    .m_axis_tdata           (network_tdata),
    .m_axis_tlast           (network_tlast),
    .m_axis_tid             (network_tid)
);
```

Last Updated: 2025-01-10

---

21 Source SRAM Controller Specification

Module: `src_sram_controller.sv` **Location:** `projects/components/rapids/rtl/macro_beats/` **Status:** Implemented **Last Updated:** 2025-01-10

21.1 Overview

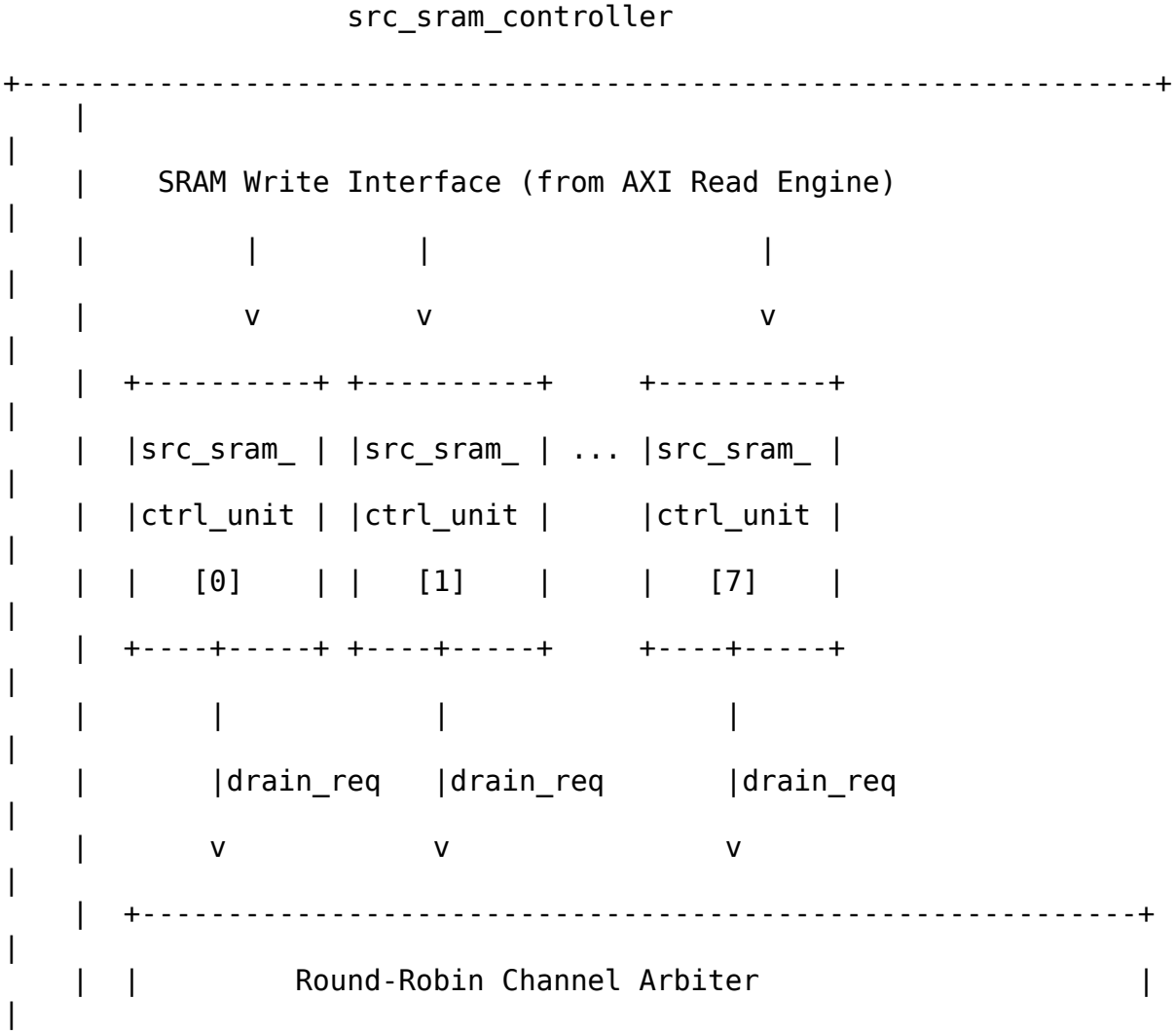
The Source SRAM Controller manages 8 SRAM controller units for the source data path, providing per-channel buffering with channel arbitration for data delivery to the network.

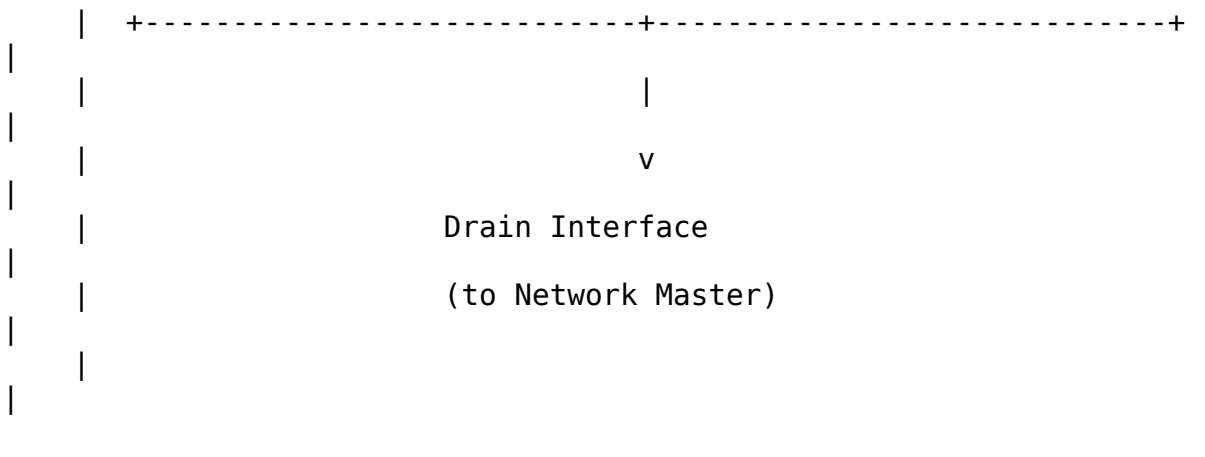
21.1.1 Key Features

- **8-Channel SRAM Array:** Instantiates 8 `src_sram_controller_unit` modules
- **Channel Arbitration:** Round-robin access for drain requests
- **Flow Control Integration:** Per-channel alloc_ctrl and drain_ctrl
- **Data Availability Tracking:** Reports available data per channel

21.1.2 Block Diagram

21.1.3 Figure 3.9.1: Source SRAM Controller Block Diagram





Source: 03_src_sram_controller_block.mmd

21.2 Parameters

```
parameter int NUM_CHANNELS = 8;
parameter int DATA_WIDTH = 512;
parameter int SRAM_DEPTH = 512;                    // Per-channel depth
parameter int ADDR_WIDTH = $clog2(SRAM_DEPTH);
```

: Table 3.9.1: Source SRAM Controller Parameters

21.3 Port List

21.3.1 Clock and Reset

Table 3.9.2: Clock and Reset

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

21.3.2 Per-Channel Fill Interface (from AXI Read Engine)

Table 3.9.3: Per-Channel Fill Interface

Signal	Direction	Width	Description
`src_fill_alloc_req`	input	NC	Allocation request per channel

Signal	Direction	Width	Description
`src_fill_alloc_size`	input	NC*16	Beats to allocate
`src_fill_space_free`	output	NC*16	Available space per channel
`src_fill_valid`	input	NC	Fill data valid
`src_fill_ready`	output	NC	Ready for fill data
`src_fill_data`	input	NC*DW	Fill data
`src_fill_last`	input	NC	Last beat marker

21.3.3 Arbitrated Drain Interface (to Network)

Table 3.9.4: Arbitrated Drain Interface

Signal	Direction	Width	Description
`drain_valid`	output	1	Drain data valid
`drain_ready`	input	1	Ready for drain
`drain_data`	output	DW	Drain data
`drain_id`	output	3	Source channel ID
`drain_last`	output	1	Last beat of transfer

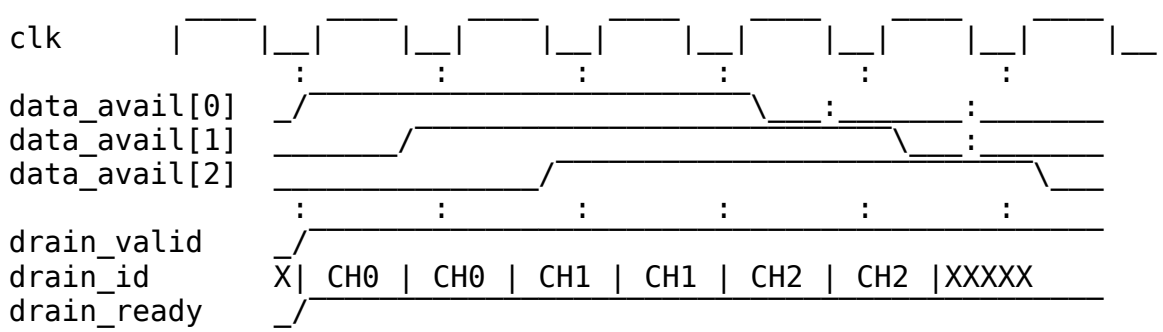
21.3.4 Per-Channel Status

Table 3.9.5: Per-Channel Status

Signal	Direction	Width	Description
`channel_data_avail`	output	NC*16	Data available per channel
`channel_empty`	output	NC	Channel empty flags
`channel_full`	output	NC	Channel full flags

21.4 Arbitration Logic

21.4.1 Figure 3.9.2: Source Channel Arbitration



TODO: Replace with simulation-generated waveform showing round-robin drain

---

21.5 Comparison: Sink vs Source SRAM Controller

Table 3.9.6: Sink vs Source Controller Comparison

Aspect	Sink Controller	Source Controller
Fill Source	Network ingress	AXI read data
Drain Destination	AXI write engine	Network egress
Fill Interface	Per-channel from network	Per-channel from AXI R
Drain Interface	Arbitrated SRAM read	Arbitrated drain to network
Primary Use	Network -> Memory	Memory -> Network

21.6 Integration Example

```
src_sram_controller #(
    .NUM_CHANNELS(8),
    .DATA_WIDTH(512),
    .SRAM_DEPTH(512)
) u_src_sram_ctrl (
    .clk                    (clk),
    .rst_n                  (rst_n),

    // Per-channel fill interface (from AXI read engine)
    .src_fill_alloc_req     (fill_alloc_req),
    .src_fill_alloc_size    (fill_alloc_size),
```

```
    .src_fill_space_free    (fill_space_free),
    .src_fill_valid         (fill_valid),
    .src_fill_ready         (fill_ready),
    .src_fill_data          (fill_data),
    .src_fill_last          (fill_last),

    // Arbitrated drain interface (to network)
    .drain_valid            (src_drain_valid),
    .drain_ready            (src_drain_ready),
    .drain_data             (src_drain_data),
    .drain_id               (src_drain_id),
    .drain_last             (src_drain_last),

    // Status
    .channel_data_avail     (channel_data_avail),
    .channel_empty          (channel_empty),
    .channel_full           (channel_full)
);
```

Last Updated: 2025-01-10

---

22 Source SRAM Controller Unit Specification

Module: `src_sram_controller_unit.sv` **Location:** `projects/components/rapids/rtl/macro_beats/` **Status:** Implemented **Last Updated:** 2025-01-10

22.1 Overview

The Source SRAM Controller Unit manages a single channel's SRAM buffer for the source data path, with beat-level flow control using virtual FIFOs.

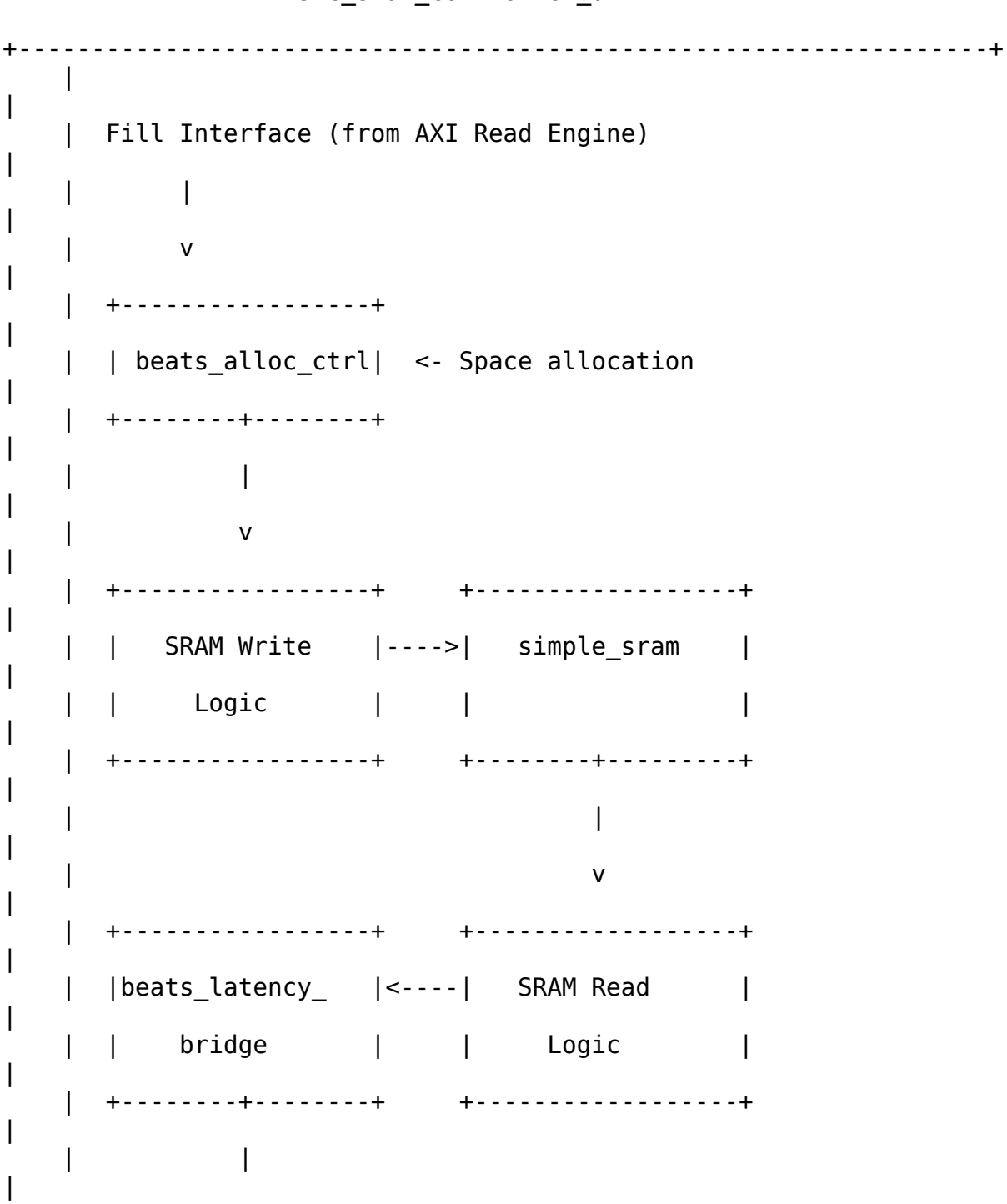
22.1.1 Key Features

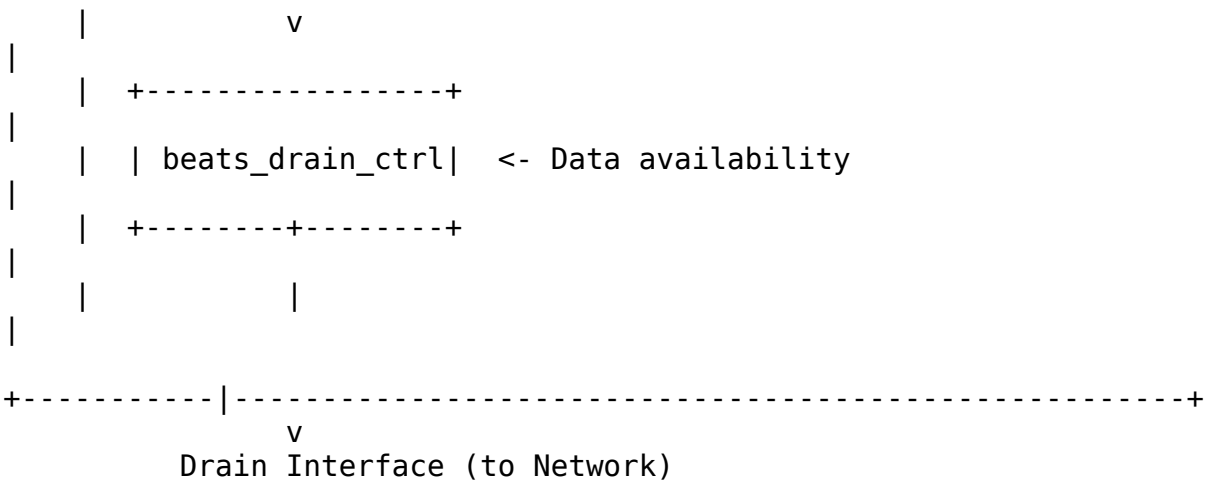
- **Single-Channel SRAM:** Dedicated SRAM buffer for one channel
- **Virtual FIFO Flow Control:** beats_alloc_ctrl + beats_drain_ctrl

- **Latency Bridge:** Compensates for pipeline timing
- **Simple SRAM:** No reset on memory, only on control logic

22.1.2 Block Diagram

22.1.3 Figure 3.10.1: Source SRAM Controller Unit Block Diagram





Source: 03_src_sram_controller_unit_block.mmd

22.2 Parameters

```
parameter int DATA_WIDTH = 512;
parameter int SRAM_DEPTH = 512;
parameter int ADDR_WIDTH = $clog2(SRAM_DEPTH);
parameter int LATENCY_BRIDGE_DEPTH = 4;
```

: Table 3.10.1: Source SRAM Controller Unit Parameters

22.3 Port List

22.3.1 Clock and Reset

Table 3.10.2: Clock and Reset

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

22.3.2 Fill Interface (from AXI Read Engine)

Table 3.10.3: Fill Interface

Signal	Direction	Width	Description
`fill_alloc_req`	input	1	Allocate space
`fill_alloc_size`	input	16	Beats to

Signal	Direction	Width	Description
			allocate
`fill_space_free`	output	16	Available space
`fill_valid`	input	1	Fill data valid
`fill_ready`	output	1	Ready for fill data
`fill_data`	input	DW	Fill data
`fill_last`	input	1	Last beat marker

22.3.3 Drain Interface (to Network)

Table 3.10.4: Drain Interface

Signal	Direction	Width	Description
`drain_req`	output	1	Data available to drain
`drain_gnt`	input	1	Drain grant from arbiter
`drain_beats`	output	16	Beats available
`drain_valid`	output	1	Drain data valid
`drain_ready`	input	1	Network ready
`drain_data`	output	DW	Drain data
`drain_last`	output	1	Last beat marker

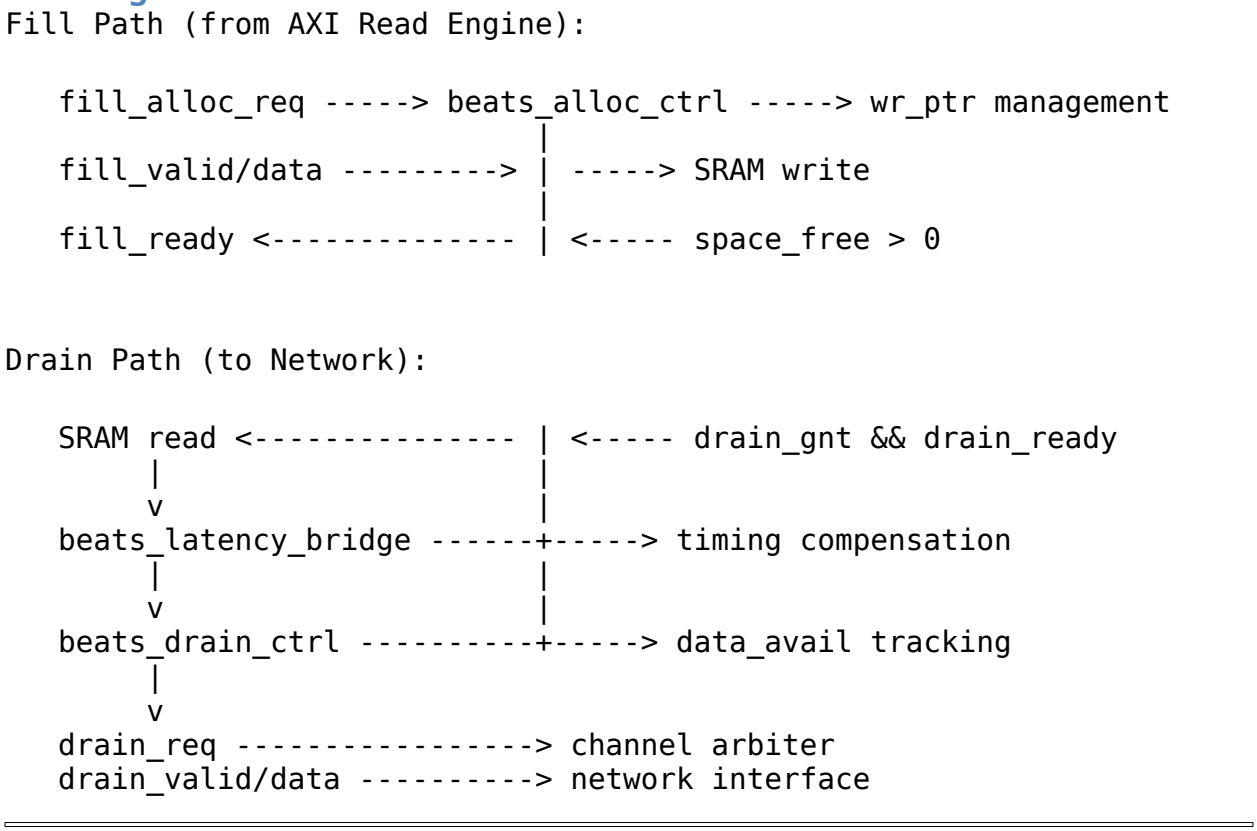
22.3.4 Status

Table 3.10.5: Status Interface

Signal	Direction	Width	Description
`data_avail`	output	16	Data available count
`empty`	output	1	Buffer empty
`full`	output	1	Buffer full

22.4 Internal Architecture

22.4.1 Figure 3.10.2: Internal Data Flow



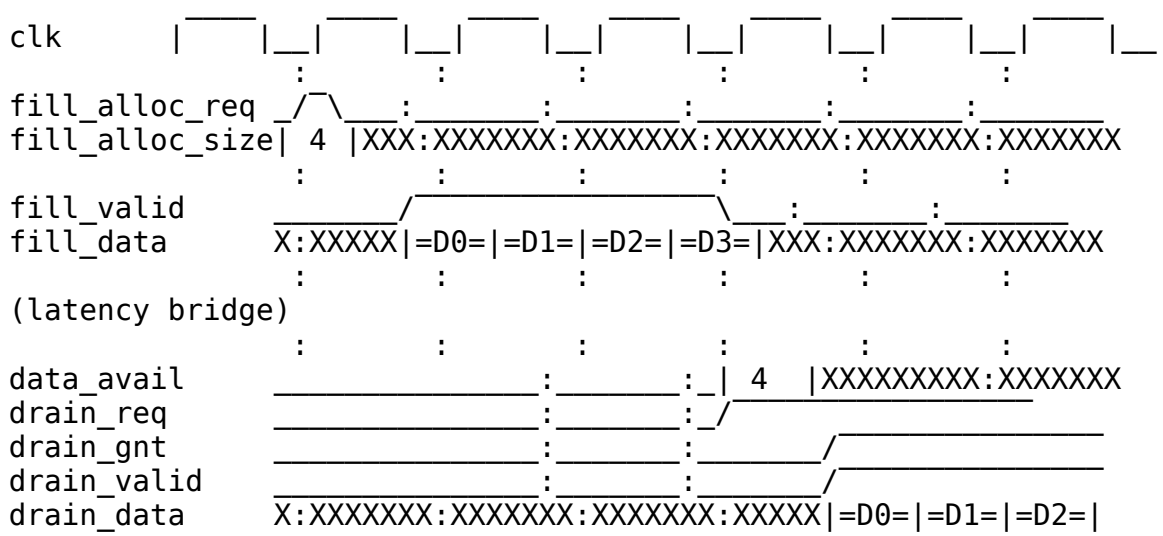
22.5 Comparison: Sink vs Source Unit

Table 3.10.6: Sink vs Source Unit Comparison

Aspect	Sink Unit	Source Unit
Fill Source	Network	AXI Read Engine
Drain Destination	AXI Write Engine	Network
Fill Handshake	fill_valid/ready	fill_valid/ready
Drain Handshake	SRAM read interface	drain_valid/ready
Primary Direction	Network -> SRAM -> AXI	AXI -> SRAM -> Network

22.6 Timing Diagram

22.6.1 Figure 3.10.3: Source Unit Fill and Drain Timing



TODO: Replace with simulation-generated waveform showing complete flow

---

22.7 Integration Example

```
src_sram_controller_unit #(
    .DATA_WIDTH(512),
    .SRAM_DEPTH(512),
    .LATENCY_BRIDGE_DEPTH(4)
) u_src_sram_unit (
    .clk                (clk),
    .rst_n              (rst_n),

    // Fill interface (from AXI read engine)
    .fill_alloc_req     (fill_alloc_req[ch]),
    .fill_alloc_size    (fill_alloc_size[ch]),
    .fill_space_free    (fill_space_free[ch]),
    .fill_valid         (fill_valid[ch]),
    .fill_ready         (fill_ready[ch]),
    .fill_data          (fill_data[ch]),
    .fill_last          (fill_last[ch]),

    // Drain interface (to network)
    .drain_req          (drain_req[ch]),
    .drain_gnt          (drain_gnt[ch]),
    .drain_beats        (drain_beats[ch]),
    .drain_valid        (drain_valid[ch]),
    .drain_ready        (drain_ready[ch]),
    .drain_data         (drain_data[ch]),
```

```
    .drain_last         (drain_last[ch]),

    // Status
    .data_avail         (data_avail[ch]),
    .empty              (empty[ch]),
    .full               (full[ch])
);
```

Last Updated: 2025-01-10

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

---

23 RAPIDS Core Beats Specification

Module: `rapids_core_beats.sv` **Location:** `projects/components/rapids/rtl/macro_beats/` **Status:** Implemented **Last Updated:** 2025-01-10

23.1 Overview

The RAPIDS Core Beats module is the top-level integration of the "beats" architecture, combining the scheduler group array with sink and source data paths.

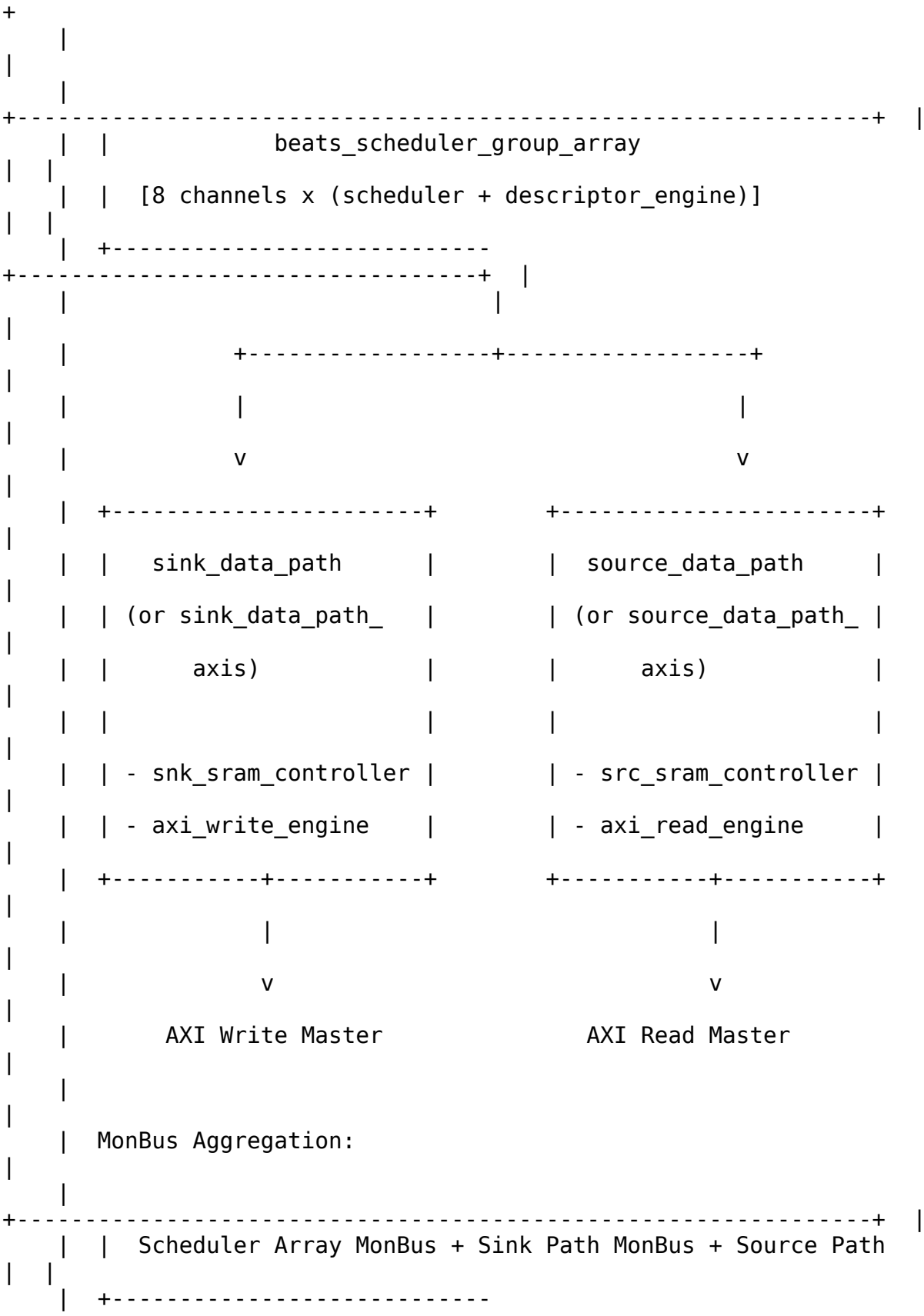
23.1.1 Key Features

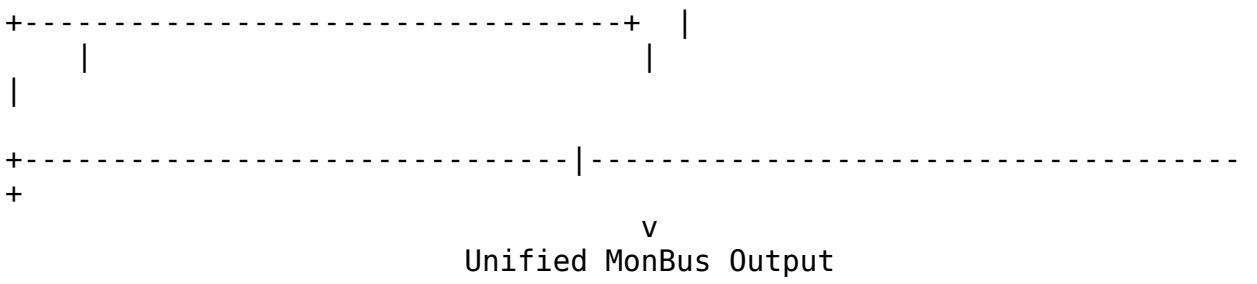
- **Complete RAPIDS Core:** Scheduler array + sink path + source path
- **8-Channel Architecture:** Full multi-channel support
- **Unified MonBus:** Aggregated monitoring from all subsystems
- **Configurable Parameters:** Data width, address width, SRAM depths

23.1.2 Block Diagram

23.1.3 Figure 3.11.1: RAPIDS Core Beats Block Diagram

```
                          rapids_core_beats

+-------------------------------------------------------------------
```


beats_scheduler_group_array
[8 channels x (scheduler + descriptor_engine)]
sink_data_path
(or sink_data_path_
axis)
- snk_sram_controller
- axi_write_engine
source_data_path
(or source_data_path_
axis)
- src_sram_controller
- axi_read_engine
AXI Write Master
AXI Read Master
MonBus Aggregation:
Scheduler Array MonBus + Sink Path MonBus + Source Path



Source: 03_rapids_core_beats_block.mmd

23.2 Parameters

```
parameter int NUM_CHANNELS = 8;
parameter int ADDR_WIDTH = 64;
parameter int DATA_WIDTH = 512;
parameter int DESC_DATA_WIDTH = 256;
parameter int AXI_ID_WIDTH = 8;
parameter int SRAM_DEPTH = 512;

// AXI Parameters
parameter int AR_MAX_OUTSTANDING = 8;
parameter int AW_MAX_OUTSTANDING = 8;
parameter int R_PHASE_FIFO_DEPTH = 64;
parameter int W_PHASE_FIFO_DEPTH = 64;
parameter int B_PHASE_FIFO_DEPTH = 16;

// MonBus Parameters
parameter int MON_UNIT_ID = 1;

// Feature Enables
parameter bit ENABLE_AXIS_WRAPPERS = 0;          // Use AXIS
interfaces
```

: Table 3.11.1: RAPIDS Core Beats Parameters

23.3 Port List

23.3.1 Clock and Reset

Table 3.11.2: Clock and Reset

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

Signal	Direction	Width	Description
			reset

23.3.2 Per-Channel APB Programming

Table 3.11.3: APB Programming Interface

Signal	Direction	Width	Description
`apb_valid`	input	NC	Channel kick-off (per-channel)
`apb_ready`	output	NC	Ready for kick-off
`apb_addr`	input	NC*AW	First descriptor address

23.3.3 Per-Channel Configuration

Table 3.11.4: Per-Channel Configuration

Signal	Direction	Width	Description
`cfg_channel_enable`	input	NC	Enable per channel
`cfg_channel_reset`	input	NC	Soft reset per channel
`cfg_sched_timeout_cycles`	input	NC*16	Timeout threshold
`cfg_sched_timeout_enable`	input	NC	Enable timeout
`cfg_desceng_enable`	input	NC	Enable descriptor engine
`cfg_desceng_prefetch`	input	NC	Enable prefetching

23.3.4 Descriptor AXI Master Interface

Table 3.11.5: Descriptor AXI Master Interface

Signal	Direction	Width	Description
`desc_m_axi_arvalid`	output	1	AR channel valid

Signal	Direction	Width	Description
desc_m_axi_arready	input	1	AR channel ready
desc_m_axi_araddr	output	AW	AR address
desc_m_axi_rvalid	input	1	R channel valid
desc_m_axi_rready	output	1	R channel ready
desc_m_axi_rdata	input	256	R data

23.3.5 Sink AXI Write Master Interface

Table 3.11.6: Sink AXI Write Master Interface

Signal	Direction	Width	Description
snk_m_axi_awvalid	output	1	AW channel valid
snk_m_axi_awready	input	1	AW channel ready
snk_m_axi_awaddr	output	AW	Write address
snk_m_axi_awlen	output	8	Burst length
snk_m_axi_wvalid	output	1	W channel valid
snk_m_axi_wready	input	1	W channel ready
snk_m_axi_wdata	output	DW	Write data
snk_m_axi_wlast	output	1	Last beat
snk_m_axi_bvalid	input	1	B channel valid
snk_m_axi_bready	output	1	B channel ready
snk_m_axi_bresp	input	2	Write response

23.3.6 Source AXI Read Master Interface

Table 3.11.7: Source AXI Read Master Interface

Signal	Direction	Width	Description
`src_m_axi_arvalid`	output	1	AR channel valid
`src_m_axi_arready`	input	1	AR channel ready
`src_m_axi_araddr`	output	AW	Read address
`src_m_axi_arlen`	output	8	Burst length
`src_m_axi_rvalid`	input	1	R channel valid
`src_m_axi_rready`	output	1	R channel ready
`src_m_axi_rdata`	input	DW	Read data
`src_m_axi_rlast`	input	1	Last beat

23.3.7 Sink Fill Interface (or AXIS Slave)

Table 3.11.8: Sink Fill Interface

Signal	Direction	Width	Description
`snk_fill_valid`	input	1	Fill data valid
`snk_fill_ready`	output	1	Ready for fill
`snk_fill_data`	input	DW	Fill data
`snk_fill_id`	input	3	Channel ID
`snk_fill_last`	input	1	Last beat

23.3.8 Source Drain Interface (or AXIS Master)

Table 3.11.9: Source Drain Interface

Signal	Direction	Width	Description
`src_drain_valid`	output	1	Drain data valid
`src_drain_ready`	input	1	Ready for

Signal	Direction	Width	Description
			drain
src_drain_data	output	DW	Drain data
src_drain_id	output	3	Channel ID
src_drain_last	output	1	Last beat

23.3.9 Unified MonBus Interface

Table 3.11.10: Unified MonBus Interface

Signal	Direction	Width	Description
monbus_pkt_valid	output	1	Packet valid
monbus_pkt_ready	input	1	Consumer ready
monbus_pkt_data	output	64	Packet data

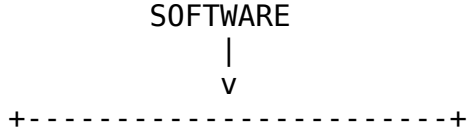
23.3.10 Aggregate Status

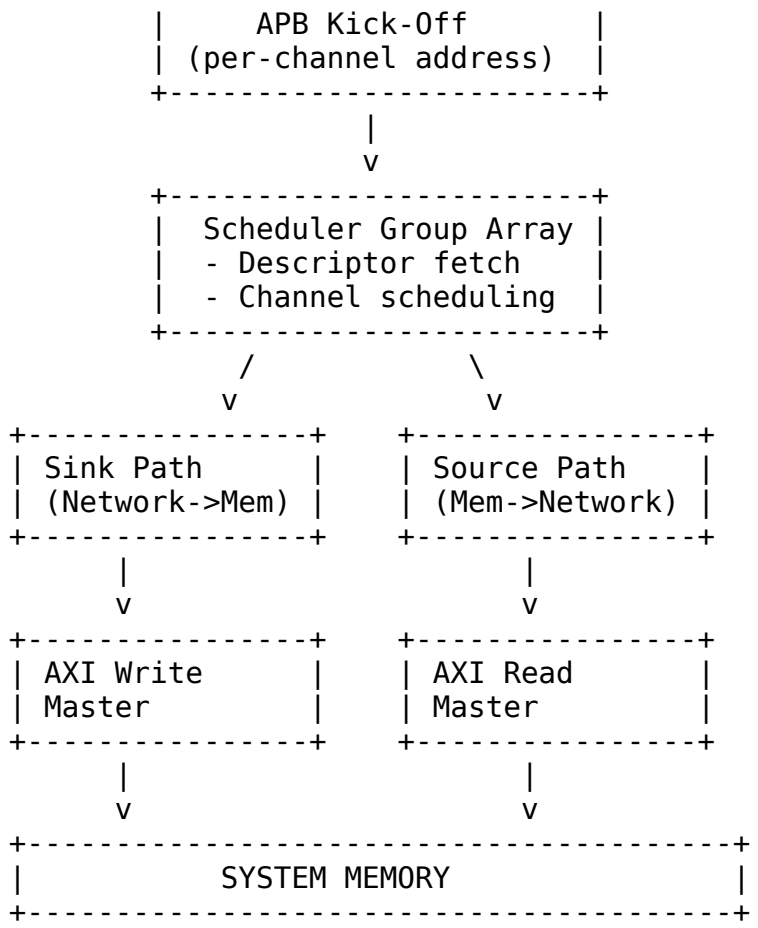
Table 3.11.11: Aggregate Status

Signal	Direction	Width	Description
all_channels_idle	output	1	All channels idle
scheduler_idle	output	NC	Per-channel scheduler idle
sink_idle	output	1	Sink path idle
source_idle	output	1	Source path idle
error_flags	output	32	Combined error flags

23.4 Data Flow

23.4.1 Figure 3.11.2: RAPIDS Core Complete Data Flow





23.5 MonBus Aggregation

The unified MonBus output aggregates sources from all subsystems:

Table 3.11.12: MonBus Source Assignment

Source Range	Origin	Description
0-15	Scheduler Array	Per-channel scheduler + desc engine
16-23	Sink Data Path	Sink SRAM controllers
24-31	Source Data Path	Source SRAM controllers
32	AXI Write Engine	Write completions
33	AXI Read Engine	Read completions

23.6 Integration Example

```
rapids_core_beats #(
    .NUM_CHANNELS(8),
    .ADDR_WIDTH(64),
    .DATA_WIDTH(512),
    .SRAM_DEPTH(512),
    .ENABLE_AXIS_WRAPPERS(0)
) u_rapids_core (
    .clk                    (clk),
    .rst_n                  (rst_n),

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

    // Configuration
    .cfg_channel_enable     (cfg_ch_enable),
    .cfg_sched_timeout_cycles(cfg_timeout),

    // Descriptor AXI
    .desc_m_axi_arvalid     (desc_arvalid),
    .desc_m_axi_arready     (desc_arready),
    .desc_m_axi_araddr      (desc_araddr),
    .desc_m_axi_rvalid      (desc_rvalid),
    .desc_m_axi_rready      (desc_rready),
    .desc_m_axi_rdata       (desc_rdata),

    // Sink AXI write
    .snk_m_axi_awvalid      (snk_awvalid),
    .snk_m_axi_awready      (snk_awready),
    .snk_m_axi_awaddr       (snk_awaddr),
    .snk_m_axi_wvalid       (snk_wvalid),
    .snk_m_axi_wready       (snk_wready),
    .snk_m_axi_wdata        (snk_wdata),
    .snk_m_axi_bvalid       (snk_bvalid),
    .snk_m_axi_bready       (snk_bready),

    // Source AXI read
    .src_m_axi_arvalid      (src_arvalid),
    .src_m_axi_arready      (src_arready),
    .src_m_axi_araddr       (src_araddr),
    .src_m_axi_rvalid       (src_rvalid),
    .src_m_axi_rready       (src_rready),
    .src_m_axi_rdata        (src_rdata),
```

```
    // Sink fill interface
    .snk_fill_valid         (network_rx_valid),
    .snk_fill_ready         (network_rx_ready),
    .snk_fill_data          (network_rx_data),
    .snk_fill_id            (network_rx_id),

    // Source drain interface
    .src_drain_valid        (network_tx_valid),
    .src_drain_ready        (network_tx_ready),
    .src_drain_data         (network_tx_data),
    .src_drain_id           (network_tx_id),

    // MonBus
    .monbus_pkt_valid       (mon_valid),
    .monbus_pkt_ready       (mon_ready),
    .monbus_pkt_data        (mon_data),

    // Status
    .all_channels_idle      (rapids_idle)
);
```

Last Updated: 2025-01-10

---

24 Interface Overview

Last Updated: 2025-01-10

24.1 External Interfaces

The RAPIDS Beats architecture exposes the following external interfaces:

24.1.1 AXI4 Master Interfaces

Table 4.0.1: AXI4 Master Interfaces

Interface	Purpose	Data Width
Descriptor AXI	Fetch descriptors from memory	256-bit
Sink AXI Write	Write sink data to memory	512-bit (param)
Source AXI Read	Read source data from memory	512-bit (param)

24.1.2 Streaming Interfaces

Table 4.0.2: Streaming Interfaces

Interface	Type	Purpose
Fill Interface	Custom	Network data ingress (to SRAM)
Drain Interface	Custom	Network data egress (from SRAM)
AXIS Sink	AXI-Stream Slave	Optional AXIS wrapper for fill
AXIS Source	AXI-Stream Master	Optional AXIS wrapper for drain

24.1.3 Control/Monitor Interfaces

Table 4.0.3: Control and Monitor Interfaces

Interface	Type	Purpose
APB	APB slave	Per-channel kick-off
Configuration	Custom	Global/per-channel config
Status	Custom	Idle, state, error flags
MonBus	Custom 64-bit	Monitor packet output

24.2 Interface Documentation

- AXI4 Interface Specification - Descriptor, sink write, source read
- AXIS Interface Specification - Optional streaming wrappers

- MonBus Interface Specification - Monitor packet format

24.3 Interface Signal Naming Conventions

Table 4.0.4: Signal Naming Conventions

Prefix	Meaning	Example
`m_axi_`	AXI master port	`m_axi_arvalid`
`s_axi_`	AXI slave port	`s_axi_awready`
`m_axis_`	AXIS master port	`m_axis_tvalid`
`s_axis_`	AXIS slave port	`s_axis_tready`
`cfg_`	Configuration input	`cfg_enable`
`snk_`	Sink data path	`snk_fill_valid`
`src_`	Source data path	`src_drain_ready`
`sched_`	Scheduler interface	`sched_rd_valid`
`monbus_`	Monitor bus	`monbus_pkt_data`

Last Updated: 2025-01-10

---

25 AXI4 Interface Specification

Last Updated: 2025-01-10

25.1 Overview

The RAPIDS Beats architecture uses three AXI4 master interfaces: 1. **Descriptor AXI:** Fetches descriptors from memory (256-bit data) 2. **Sink AXI Write:** Writes sink data to memory (512-bit data, configurable) 3. **Source AXI Read:** Reads source data from memory (512-bit data, configurable)

25.2 Descriptor AXI Master Interface

25.2.1 Purpose

Fetches descriptor packets from system memory. Shared across all 8 channels with round-robin arbitration.

25.2.2 Signal Table

Table 4.1.1: Descriptor AXI Master Interface Signals

Signal	Direction	Width	Description
`m_axi_arvalid`	output	1	AR channel valid
`m_axi_arready`	input	1	AR channel ready
`m_axi_araddr`	output	AW	Read address
`m_axi_arid`	output	ID_W	Transaction ID
`m_axi_arlen`	output	8	Burst length (fixed: 0 = 1 beat)
`m_axi_arsize`	output	3	Burst size (fixed: 5 = 32 bytes)
`m_axi_arburst`	output	2	Burst type (fixed: INCR)
`m_axi_rvalid`	input	1	R channel valid
`m_axi_rready`	output	1	R channel ready
`m_axi_rdata`	input	256	Read data (descriptor)
`m_axi_rid`	input	ID_W	Response ID
`m_axi_rresp`	input	2	Read response
`m_axi_rlast`	input	1	Last beat

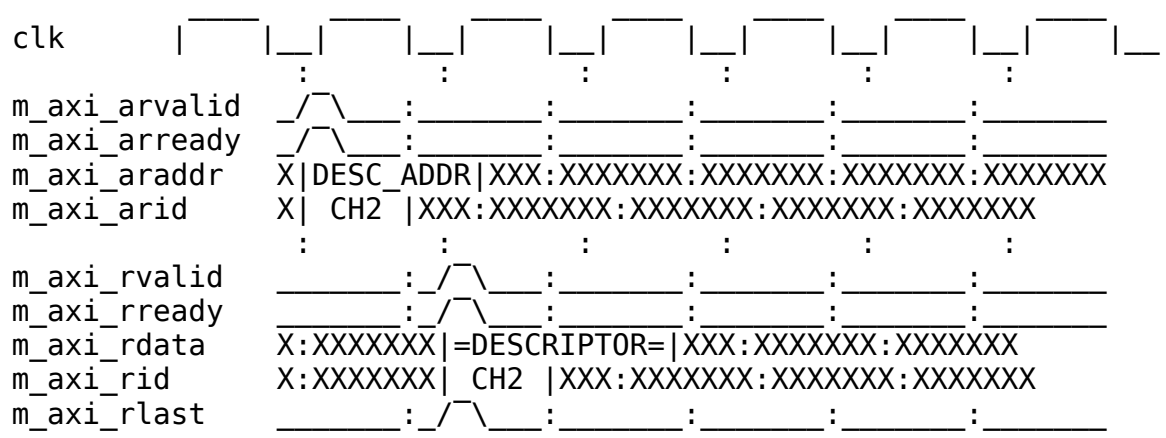
25.2.3 Transaction Characteristics

Table 4.1.2: Descriptor AXI Transaction Characteristics

Parameter	Value	Description
Data Width	256 bits	Fixed descriptor size
Burst Length	1 beat	Single descriptor per transaction
Burst Type	INCR	Incrementing burst
Max Outstanding	8	Configurable via parameter
ID Usage	Per-channel	ID encodes source channel

25.2.4 Timing Diagram

25.2.5 Figure 4.1.1: Descriptor Fetch Timing



TODO: Replace with simulation-generated waveform

25.3 Sink AXI Write Master Interface

25.3.1 Purpose

Writes sink data from SRAM to system memory. Supports burst transactions for efficient memory bandwidth.

25.3.2 Signal Table

Table 4.1.3: Sink AXI Write Master Interface Signals

Signal	Direction	Width	Description
`m_axi_awvalid`	output	1	AW channel valid
`m_axi_awready`	input	1	AW channel ready
`m_axi_awaddr`	output	AW	Write address
`m_axi_awid`	output	ID_W	Transaction ID
`m_axi_awlen`	output	8	Burst length (0-255)
`m_axi_awsize`	output	3	Burst size
`m_axi_awburst`	output	2	Burst type
`m_axi_wvalid`	output	1	W channel valid
`m_axi_wready`	input	1	W channel ready
`m_axi_wdata`	output	DW	Write data
`m_axi_wstrb`	output	DW/8	Write strobes
`m_axi_wlast`	output	1	Last beat
`m_axi_bvalid`	input	1	B channel valid
`m_axi_bready`	output	1	B channel ready
`m_axi_bid`	input	ID_W	Response ID
`m_axi_bresp`	input	2	Write response

25.3.3 Transaction Characteristics

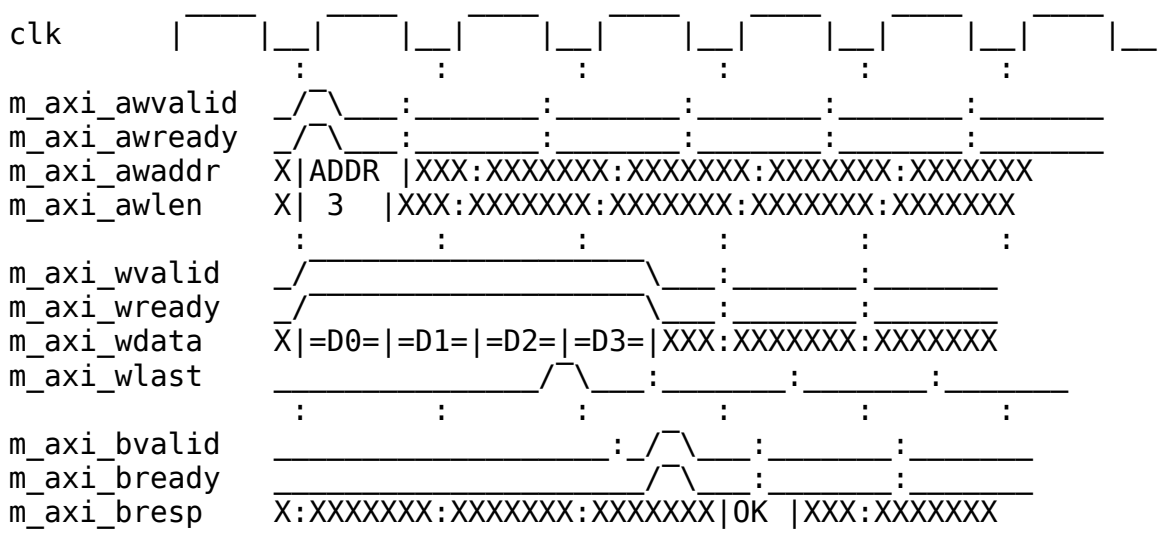
Table 4.1.4: Sink AXI Write Transaction Characteristics

Parameter	Value	Description
Data Width	512 bits	Configurable
Burst Length	1-256 beats	Based on transfer size
Burst Type	INCR	Incrementing burst

Parameter	Value	Description
Max Outstanding AW	8	Configurable
W FIFO Depth	64	Configurable
B FIFO Depth	16	Configurable

25.3.4 Timing Diagram

25.3.5 Figure 4.1.2: Sink AXI Write Burst Timing



TODO: Replace with simulation-generated waveform

25.4 Source AXI Read Master Interface

25.4.1 Purpose

Reads source data from system memory into SRAM. Supports burst transactions for efficient memory bandwidth.

25.4.2 Signal Table

Table 4.1.5: Source AXI Read Master Interface Signals

Signal	Direction	Width	Description
`m_axi_arvalid`	output	1	AR channel valid
`m_axi_arready`	input	1	AR channel ready

Signal	Direction	Width	Description
m_axi_araddr	output	AW	Read address
m_axi_arid	output	ID_W	Transaction ID
m_axi_arlen	output	8	Burst length (0-255)
m_axi_arsize	output	3	Burst size
m_axi_arburst	output	2	Burst type
m_axi_rvalid	input	1	R channel valid
m_axi_rready	output	1	R channel ready
m_axi_rdata	input	DW	Read data
m_axi_rid	input	ID_W	Response ID
m_axi_rresp	input	2	Read response
m_axi_rlast	input	1	Last beat

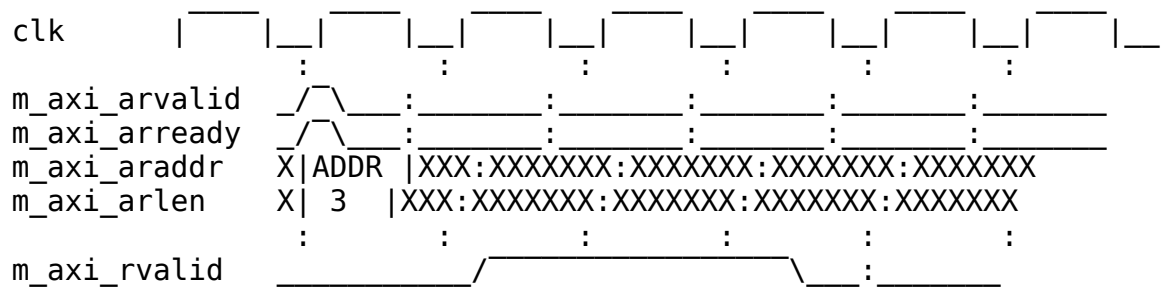
25.4.3 Transaction Characteristics

Table 4.1.6: Source AXI Read Transaction Characteristics

Parameter	Value	Description
Data Width	512 bits	Configurable
Burst Length	1-256 beats	Based on transfer size
Burst Type	INCR	Incrementing burst
Max Outstanding AR	8	Configurable
R FIFO Depth	64	Configurable

25.4.4 Timing Diagram

25.4.5 Figure 4.1.3: Source AXI Read Burst Timing



```
m_axi_rready   ___________/‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾\___:________
m_axi_rdata    X:XXXXXXX|=D0=|=D1=|=D2=|=D3=|XXX:XXXXXXX
m_axi_rlast    ____________________________/‾\___:________:________
```

TODO: Replace with simulation-generated waveform

25.5 AXI Protocol Compliance

25.5.1 Supported Features

Table 4.1.7: AXI Feature Support

Feature	Descriptor	Sink Write	Source Read
Burst Type INCR	Yes	Yes	Yes
Burst Type FIXED	No	No	No
Burst Type WRAP	No	No	No
Exclusive Access	No	No	No
Locked Access	No	No	No
Unaligned Access	No	Yes	Yes
Narrow Transfers	No	Yes	Yes

25.5.2 Error Handling

- **SLVERR:** Transaction aborted, error reported to scheduler
- **DECERR:** Transaction aborted, error reported to scheduler
- **Timeout:** Configurable watchdog, error reported to scheduler

---

Last Updated: 2025-01-10

---

26 AXI-Stream Interface Specification

Last Updated: 2025-01-10

26.1 Overview

The RAPIDS Beats architecture provides optional AXI-Stream (AXIS) wrappers for network interfaces: 1. **AXIS Sink Slave:** Receives network data for memory writes 2. **AXIS Source Master:** Transmits network data from memory reads

These wrappers are available when `ENABLE_AXIS_WRAPPERS = 1`.

26.2 AXIS Sink Slave Interface

26.2.1 Purpose

Receives streaming network data and routes to per-channel SRAM buffers based on TID.

26.2.2 Signal Table

Table 4.2.1: AXIS Sink Slave Interface Signals

Signal	Direction	Width	Description
`s_axis_tvalid`	input	1	Data valid
`s_axis_tready`	output	1	Ready to accept data
`s_axis_tdata`	input	DATA_WIDTH	Data payload
`s_axis_tkeep`	input	DATA_WIDTH/8	Byte enables
`s_axis_tlast`	input	1	Last beat of packet
`s_axis_tid`	input	TID_WIDTH	Stream ID (channel select)
`s_axis_tdest`	input	TDEST_WIDTH	Destination routing
`s_axis_tuser`	input	TUSER_WIDTH	User sideband

26.2.3 Signal Details

26.2.3.1 TID (Transaction ID)

TID is used for per-channel routing:

Table 4.2.2: TID Channel Mapping

TID Value	Destination
0	Channel 0 SRAM
1	Channel 1 SRAM
2	Channel 2 SRAM
3	Channel 3 SRAM
4	Channel 4 SRAM
5	Channel 5 SRAM
6	Channel 6 SRAM
7	Channel 7 SRAM

26.2.3.2 TKEEP (Byte Enables)

TKEEP indicates valid bytes within TDATA:

Table 4.2.3: TKEEP Configuration

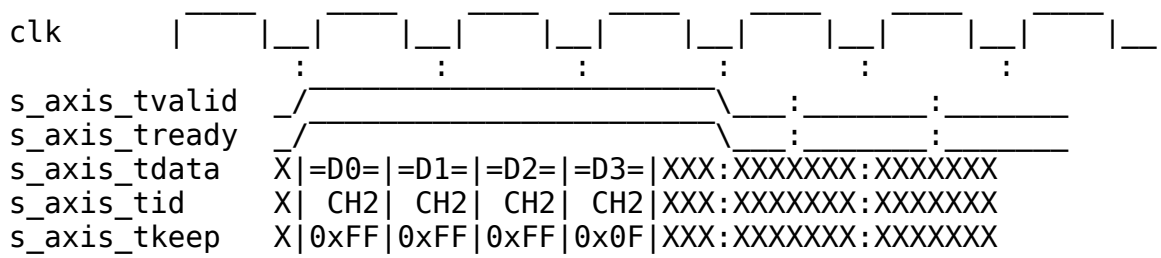
DATA_WIDTH	TKEEP Width	Usage
512 bits	64 bits	Per-byte valid
256 bits	32 bits	Per-byte valid

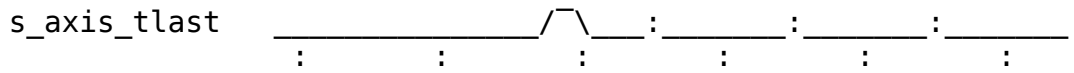
26.2.3.3 TLAST (Packet Boundary)

TLAST marks the end of an AXIS packet. This is mapped to the internal `fill_last` signal for packet boundary tracking.

26.2.4 Timing Diagram

26.2.5 Figure 4.2.1: AXIS Sink Slave Timing





TODO: Replace with simulation-generated waveform

26.2.6 Backpressure

The sink interface asserts backpressure (`s_axis_tready = 0`) when: - Target channel SRAM is full - No space allocated for incoming data - Internal pipeline stalls

---

26.3 AXIS Source Master Interface

26.3.1 Purpose

Transmits streaming network data from per-channel SRAM buffers with TID indicating source channel.

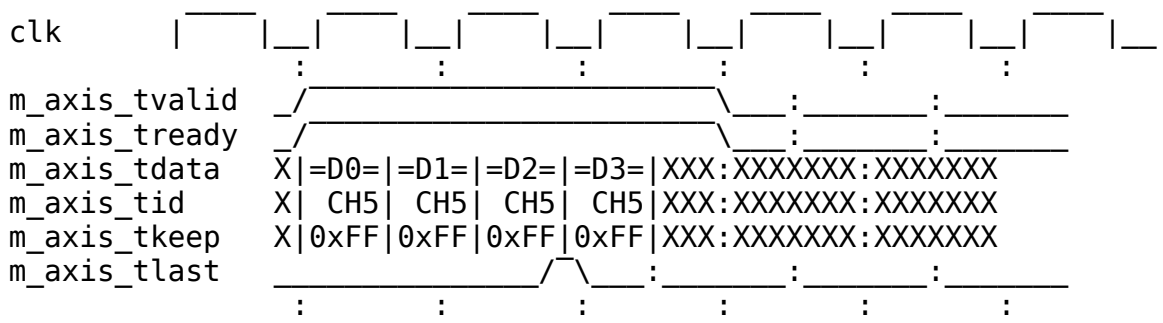
26.3.2 Signal Table

Table 4.2.4: AXIS Source Master Interface Signals

Signal	Direction	Width	Description
`m_axis_tvalid`	output	1	Data valid
`m_axis_tready`	input	1	Ready to accept data
`m_axis_tdata`	output	DATA_WIDTH	Data payload
`m_axis_tkeep`	output	DATA_WIDTH/8	Byte enables
`m_axis_tlast`	output	1	Last beat of packet
`m_axis_tid`	output	TID_WIDTH	Stream ID (source channel)
`m_axis_tdest`	output	TDEST_WIDTH	Destination routing
`m_axis_tuser`	output	TUSER_WIDTH	User sideband

26.3.3 Timing Diagram

26.3.4 Figure 4.2.2: AXIS Source Master Timing



TODO: Replace with simulation-generated waveform

26.3.5 Flow Control

The source interface respects network backpressure: - Data held when `m_axis_tready = 0` - No data loss on backpressure - SRAM drain pauses until ready

---

26.4 AXIS vs Custom Fill/Drain Interfaces

26.4.1 Comparison

Table 4.2.5: Interface Comparison

Aspect	Custom Fill/Drain	AXIS
Standard	Proprietary	AXI-Stream
Channel ID	fill_id / drain_id	TID
Packet Boundary	fill_last / drain_last	TLAST
Byte Enables	fill_strb / drain_strb	TKEEP
Interoperability	RAPIDS only	Industry standard

26.4.2 Selection Guide

Table 4.2.6: Interface Selection Guide

Use Case	Recommended Interface
Internal RAPIDS testing	Custom Fill/Drain
Network IP integration	AXIS
FPGA board-level design	AXIS

Use Case	Recommended Interface
Maximum performance	Custom Fill/Drain

26.5 Configuration Parameters

```
// AXIS Interface Parameters
parameter int DATA_WIDTH = 512;        // Data payload width
parameter int TID_WIDTH = 3;           // Stream ID width (log2
channels)
parameter int TDEST_WIDTH = 1;         // Destination width
parameter int TUSER_WIDTH = 1;         // User sideband width
```

Last Updated: 2025-01-10

---

27 MonBus Interface Specification

Last Updated: 2025-01-10

27.1 Overview

The MonBus (Monitor Bus) is a 64-bit packet-based interface for reporting internal events, status, and performance data. All RAPIDS subsystems generate MonBus packets that are aggregated and output through a unified interface.

27.2 MonBus Signal Interface

27.2.1 Signal Table

Table 4.3.1: MonBus Interface Signals

Signal	Direction	Width	Description
`monbus_pkt_valid`	output	1	Packet valid

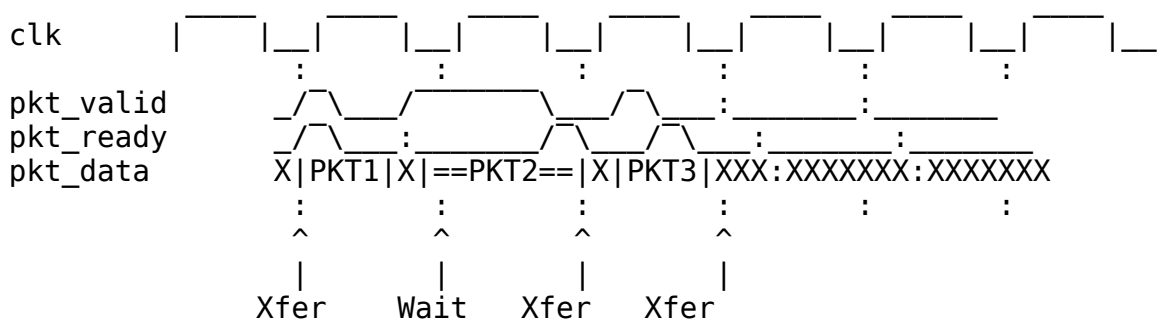
Signal	Direction	Width	Description
`monbus_pkt_ready`	input	1	Consumer ready
`monbus_pkt_data`	output	64	Packet data

27.2.2 Handshaking Protocol

Standard valid/ready handshaking: - Data transfers when `valid & ready` both high - Producer holds `valid` until `ready` seen - Consumer asserts `ready` when able to accept

27.2.3 Timing Diagram

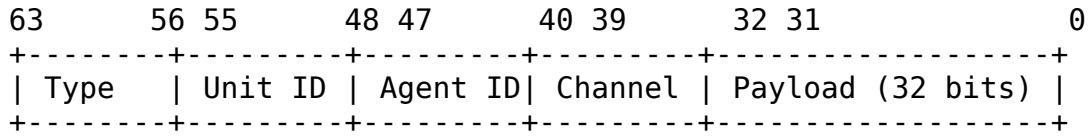
27.2.4 Figure 4.3.1: MonBus Packet Transfer Timing



TODO: Replace with simulation-generated waveform

27.3 MonBus Packet Format

27.3.1 64-bit Packet Structure



27.3.2 Figure 4.3.2: MonBus Packet Format

Table 4.3.2: MonBus Packet Fields

Field	Bits	Description
Type	[63:56]	Packet type code
Unit ID	[55:48]	Reporting unit identifier

Field	Bits	Description
Agent ID	[47:40]	Source agent within unit
Channel	[39:32]	Channel number (0-7)
Payload	[31:0]	Type-specific payload

27.4 Packet Types

27.4.1 Type Code Definitions

Table 4.3.3: MonBus Packet Types

Type Code	Name	Description
0x01	DESC_FETCH	Descriptor fetch started
0x02	DESC_COMPLETE	Descriptor fetch completed
0x03	SCHED_START	Scheduler started transfer
0x04	SCHED_DONE	Scheduler transfer complete
0x10	AXI_RD_START	AXI read burst started
0x11	AXI_RD_DONE	AXI read burst completed
0x12	AXI_WR_START	AXI write burst started
0x13	AXI_WR_DONE	AXI write burst completed
0x20	SRAM_FILL	SRAM fill event
0x21	SRAM_DRAIN	SRAM drain event
0x30	ERROR	Error event
0x31	TIMEOUT	Timeout event
0xFF	HEARTBEAT	Periodic heartbeat

27.4.2 Payload Formats by Type

27.4.2.1 DESC_FETCH / DESC_COMPLETE (0x01, 0x02)

Table 4.3.4: Descriptor Packet Payload

Bits	Description
[31:16]	Descriptor address [47:32]
[15:0]	Descriptor address [31:16]

27.4.2.2 SCHED_START / SCHED_DONE (0x03, 0x04)

Table 4.3.5: Scheduler Packet Payload

Bits	Description
[31:16]	Transfer ID
[15:0]	Beat count

27.4.2.3 AXI_RD/WR_START/DONE (0x10-0x13)

Table 4.3.6: AXI Event Packet Payload

Bits	Description
[31:24]	AXI ID
[23:16]	Burst length
[15:0]	Reserved

27.4.2.4 ERROR (0x30)

Table 4.3.7: Error Packet Payload

Bits	Description
[31:24]	Error code
[23:16]	Error source
[15:0]	Error details

27.5 MonBus Source Aggregation

27.5.1 RAPIDS Core MonBus Sources

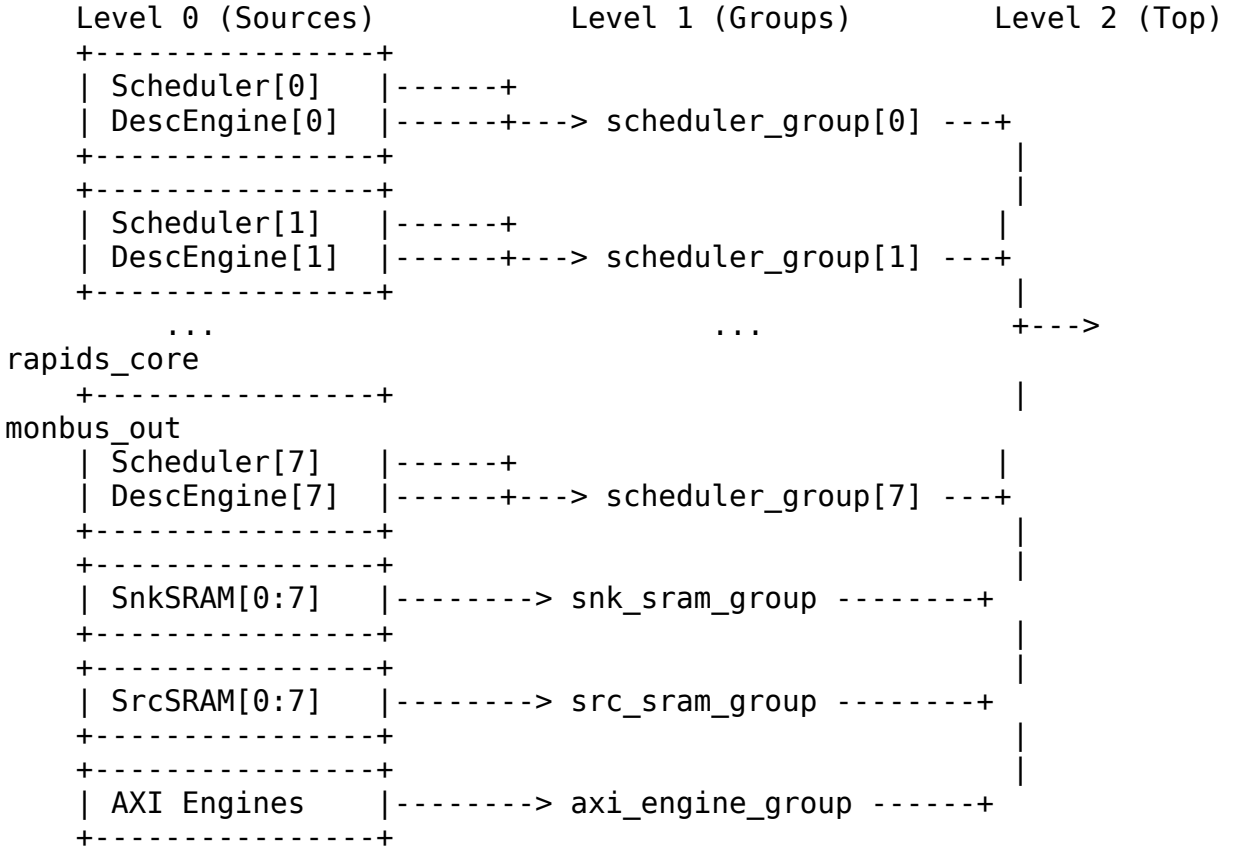
Table 4.3.8: MonBus Source Assignment

Source ID	Agent	Description
0-7	Scheduler[0:7]	Per-channel scheduler events

Source ID	Agent	Description
8-15	DescEngine[0:7]	Per-channel descriptor events
16-23	SnkSRAM[0:7]	Sink SRAM controller events
24-31	SrcSRAM[0:7]	Source SRAM controller events
32	AXI Write Engine	Write transaction events
33	AXI Read Engine	Read transaction events
34	Error Handler	Error events

27.5.2 Aggregation Hierarchy

27.5.3 Figure 4.3.3: MonBus Aggregation Tree



27.6 Integration Guidelines

27.6.1 Consumer Requirements

MonBus consumers must: 1. Provide sufficient ready cycles to prevent packet loss 2. Handle bursty traffic (multiple sources may have packets simultaneously) 3. Process or buffer packets at wire rate

27.6.2 Recommended Consumer Patterns

```
// Pattern 1: Direct FIFO buffering
gaxi_fifo_sync #(
    .DATA_WIDTH(64),
    .DEPTH(256)
) u_monbus_fifo (
    .i_clk      (clk),
    .i_rst_n    (rst_n),
    .i_valid    (monbus_pkt_valid),
    .i_data     (monbus_pkt_data),
    .o_ready    (monbus_pkt_ready),
    .o_valid    (fifo_valid),
    .o_data     (fifo_data),
    .i_ready    (consumer_ready)
);

// Pattern 2: Software register interface
always_ff @(posedge clk) begin
    if (monbus_pkt_valid && monbus_pkt_ready) begin
        sw_monbus_data <= monbus_pkt_data;
        sw_monbus_valid <= 1'b1;
    end
end
assign monbus_pkt_ready = !sw_monbus_valid || sw_read;
```

27.7 Timing Requirements

Table 4.3.9: MonBus Timing Parameters

Parameter	Value	Description
Min Ready	1 cycle	Must assert ready within N cycles
Max Latency	Configurable	Before packet dropped
Packet Rate	Variable	Depends on activity

Last Updated: 2025-01-10

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

---

28 Product Requirements Document (PRD)

28.1 RAPIDS - Rapid AXI Programmable In-band Descriptor System

Version: 1.0 **Date:** 2025-09-30 **Status:** Active Development - Validation in Progress **Owner:** RTL Design Sherpa Project **Parent Document:** `/PRD.md`

28.2 1. Executive Summary

The Rapid AXI Programmable In-band Descriptor System (RAPIDS) is a custom hardware accelerator designed for efficient memory-to-memory data movement with network interface integration. It demonstrates complex FSM coordination, descriptor-based DMA operations, and comprehensive monitoring capabilities.

28.2.1 1.1 Quick Stats

- **Modules:** 17 SystemVerilog files
- **Architecture:** 3 major blocks (Scheduler, Sink, Source)
- **Interfaces:** AXI4 (memory), Network (network), AXIL4 (control), MonBus (monitoring)
- **Test Coverage:** ~80% functional (basic scenarios validated)
- **Status:** Active validation, known issues documented

28.2.2 1.2 Subsystem Goals

- **Primary:** Demonstrate complex accelerator design patterns
- **Secondary:** Provide DMA-style memory transfer capability
- **Tertiary:** Educational reference for descriptor-based engines

---

28.3 2. Documentation Structure

This PRD provides a high-level overview. **Detailed specifications are maintained separately:**

28.3.1 📚 Complete RAPIDS Specification

Location: `projects/components/rapids/docs/rapids_spec/`

- **Index** - Complete specification structure

28.3.1.1 Chapter 1: Overview

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

28.3.1.2 Chapter 2: Block Specifications

- Blocks Overview

Scheduler Group: - Scheduler Group - Scheduler - Task management FSM - Descriptor Engine - Descriptor parsing - Program Engine - Program sequencing

Sink Data Path (Network → Memory): - Sink Data Path - Network Slave - Network ingress - Sink SRAM Control - Buffer management - Sink AXI Write Engine - Memory writes

Source Data Path (Memory → Network): - Source Data Path - Network Master - Network egress - Source SRAM Control - Buffer management - Source AXI Read Engine - Memory reads

Monitoring: - MonBus AXIL Group - Control/status - FSM Summary - State machine overview

28.3.1.3 Chapter 3: Interfaces

- Top-Level Ports
- AXIL4 Interface
- AXI4 Interface
- Network Interface
- MonBus Interface

28.3.1.4 Chapter 4 & 5: Programming

- Programming Model
- Register Definitions

28.3.2 🐛 Known Issues

Location: `projects/components/rapids/known_issues/`

- **Scheduler** - Credit counter initialization bug
- **Sink Data Path** - Minor issues
- **Sink SRAM Control** - Edge cases

28.3.3 📖 Other Documentation

- **README** - Quick start and integration guide
- **CLAUDE** - AI assistance guide for this subsystem
- **TASKS** - Current work items (to be created)
- **Validation Report** - Test results

28.4 2.4 Organizational Standards - RAPIDS Code Location

⚠️ MANDATORY: All RAPIDS-specific code must be in the project area ⚠️

28.4.1 Code Organization Principle

"All RAPIDS-specific verification code MUST reside in `projects/components/rapids/dv/` for easy discovery."

This subsystem follows the repository-wide organizational standard (see `/PRD.md` Section 2.3) requiring all project-specific code to be located in the project area, NOT the framework area.

28.4.2 RAPIDS Directory Structure

```
projects/components/rapids/
├── rtl/                              # RTL source code
│   ├── includes/                     # RAPIDS packages (rapids_pkg.sv)
│   ├── rapids_fub/                   # Functional unit blocks
│   └── rapids_macro/                 # Top-level integration
│
└── dv/                               # Design verification (all RAPIDS-
specific)
    ├── tbclasses/                    # ★ RAPIDS TB classes HERE (not
framework!)
    │   ├── scheduler_tb.py           # Scheduler testbench class
    │   ├── descriptor_engine_tb.py
```

```
│   ├── program_engine_tb.py
│   └── rapids_integration_tb.py
│
├── components/               # ★ RAPIDS-specific BFMs
│   └── (project-specific components if needed)
│
├── scoreboards/              # ★ RAPIDS-specific scoreboards
│   └── (verification scoreboards)
│
└── tests/                    # Test runners (import TB classes)
    ├── fub_tests/            # Functional unit block tests
    │   ├── scheduler/
    │   │   └── test_scheduler.py
    │   ├── descriptor_engine/
    │   └── program_engine/
    ├── integration_tests/    # Multi-block scenarios
    └── system_tests/         # Full RAPIDS operation
```

28.4.3 What Goes Where?

Code Type	✓ CORRECT Location	✗ WRONG Location
RAPIDS TB Classes	`projects/components/rapids/dv/tbclasses/`	`bin/CocoTBFramework/tbclasses/rapids/`
RAPIDS-Specific BFMs	`projects/components/rapids/dv/components/`	`bin/CocoTBFramework/components/rapids/`
RAPIDS Scoreboards	`projects/components/rapids/dv/scoreboards/`	`bin/CocoTBFramework/scoreboards/rapids/`
Test Runners	`projects/components/rapids/dv/tests/`	Anywhere else
Shared AXI4/APB BFMs	`bin/CocoTBFramework/components/{protocol}/`	Project area

28.4.4 Import Pattern for RAPIDS Tests

✓ CORRECT - Import from Project Area:

```python
# Import framework utilities (PYTHONPATH includes bin/)
import os, sys
from CocoTBFramework.tbclasses.shared.utilities import get_repo_root
from CocoTBFramework.tbclasses.shared.tbbase import TBBase

# Add repo root to Python path using robust git-based method
repo_root = get_repo_root()
sys.path.insert(0, repo_root)
```

```
# Import RAPIDS TB classes from PROJECT AREA
from projects.components.rapids.dv.tbclasses.scheduler_tb import
SchedulerTB
from projects.components.rapids.dv.tbclasses.descriptor_engine_tb
import DescriptorEngineTB

# Shared framework components
from CocoTBFramework.components.axi4.axi4_master import AXI4Master
```

✗ WRONG - Don't Import from Framework:

```
# DON'T DO THIS!
from CocoTBFramework.tbclasses.rapids.scheduler_tb import SchedulerTB
# ✗ WRONG!
```

28.4.5 Benefits of This Organization

1. **Easy Discovery** - All RAPIDS code in ONE place: `projects/components/rapids/`
2. **Clear Ownership** - RAPIDS team owns their `dv/` area completely
3. **No Confusion** - Never wonder "where does this TB class live?"
4. **Maintainability** - Changes isolated to RAPIDS area don't affect other projects
5. **Framework Stays Clean** - Only truly shared cross-project code in framework

28.4.6 Compliance Status

√ **RAPIDS is now compliant** - All TB classes moved to project area as of 2025-10-18

Migration History: - **Before:** TB classes incorrectly in `bin/CocoTBFramework/tbclasses/rapids/` - **After:** TB classes correctly in `projects/components/rapids/dv/tbclasses/` - **Test Imports:** Updated to import from project area

📖 **Complete Documentation:** See `/PRD.md` Section 2.3 for repository-wide organizational standards.

28.5 3. Architecture Overview

28.5.1 3.1 Top-Level Block Diagram

```
RAPIDS (Rapid AXI Programmable In-band Descriptor System)
├── Scheduler Group
│   ├── Scheduler          (Task FSM, credit management)
```

```
│   ├── Descriptor Engine  (Descriptor FIFO, parsing)
│   └── Program Engine     (Program sequencing, alignment)
│
├── Sink Data Path (Network → SRAM → System Memory)
│   ├── Network Slave        (Network ingress interface)
│   ├── Sink SRAM Control (Buffering, flow control)
│   └── Sink AXI Writer   (AXI4 write to system memory)
│
├── Source Data Path (System Memory → SRAM → Network)
│   ├── Source AXI Reader (AXI4 read from system memory)
│   ├── Source SRAM Control (Buffering, flow control)
│   └── Network Master       (Network egress interface)
│
└── MonBus AXIL Group
    ├── AXIL4 Slave       (Control/status registers)
    └── MonBus Reporter   (Monitor packet generation)
```

📖 **See:** `rapids_spec/ch02_blocks/00_overview.md` for detailed architecture

28.5.2 3.2 Data Flow

Sink Path (Receive): 1. Network packets arrive via Network Slave 2. Buffered in Sink SRAM 3. DMA'd to system memory via AXI4 Write Engine 4. Completion reported via MonBus

Source Path (Transmit): 1. Descriptor specifies data location in system memory 2. Source AXI Reader fetches data to Source SRAM 3. Network Master transmits to network 4. Completion reported via MonBus

Scheduler Coordination: - Parses descriptors from Descriptor Engine - Manages credit-based flow control - Sequences program engine operations - Coordinates sink/source data paths

---

28.6 4. Key Features

28.6.1 4.1 Descriptor-Based Operation

Feature	Status	Description
Descriptor FIFO	✓	Queued descriptor processing
Multi-field parsing	✓	Address, length, control fields
Chained descriptors	⏳	Future enhancement
Completion	✓	Via MonBus packets

Feature	Status	Description
reporting		

28.6.2 4.2 Data Path Features

Feature	Status	Description
SRAM buffering	✓	Decouple network from memory
AXI4 burst support	✓	Efficient memory transfers
Backpressure handling	✓	Flow control on all interfaces
Data alignment	✓	Handle unaligned transfers

28.6.3 4.3 Scheduler Features

Feature	Status	Description
Task FSM	✓	Multi-state coordination
Credit management	✓	Exponential encoding (0→1, 1→2, 2→4, ..., 15→∞)
Program sequencing	✓	Coordinated operations
Error detection	✓	Timeout, overflow detection

Credit Management Details: - Uses **exponential credit encoding** for compact configuration - 4-bit `cfg_initial_credit` decodes to actual credit counts: - `0` → 1 credit (2^0), 4 → 16 credits (2^4), 8 → 256 credits (2^8) - `15` → ∞ (unlimited credits, 0xFFFFFFFF) - Encoding applied at initialization; runtime operations are linear (increment/decrement by 1) - Provides wide range (1 to 16384) with minimal configuration overhead

📖 **See:** `rapids_spec/ch02_blocks/01_01_scheduler.md` for complete encoding table

28.6.4 4.4 Monitoring Integration

Feature	Status	Description
MonBus packets	✓	StandardAMBA 64-bit format
Descriptor events	✓	Start/complete reporting

Feature	Status	Description
Error events	√	Timeout, overflow, underflow
Performance metrics	⏳	Future enhancement

28.7 5. Interfaces

28.7.1 5.1 External Interfaces

Interface	Type	Width	Purpose
AXIL4	Slave	32-bit	Control/status registers
AXI4 (Sink)	Master	Configurable	Write to system memory
AXI4 (Source)	Master	Configurable	Read from system memory
Network (Sink)	Slave	Configurable	Network ingress
Network (Source)	Master	Configurable	Network egress
MonBus	Master	64-bit	Monitor packet output

📖 **See:** `rapids_spec/ch03_interfaces/` for complete interface specs

28.7.2 5.2 Configuration Parameters

```
// Example RAPIDS instantiation parameters
miop_top #(
    .AXI_ADDR_WIDTH(32),
    .AXI_DATA_WIDTH(64),
    .Network_DATA_WIDTH(64),
    .SRAM_DEPTH(1024),
    .MAX_DESCRIPTORS(16)
) u_miop (
    .aclk               (clk),
    .aresetn            (rst_n),
    // AXIL4 control interface
    .s_axil_*           (...),
    // AXI4 memory interfaces
```

```
    .m_axi_sink_*       (...),
    .m_axi_source_*     (...),
    // Network network interfaces
    .s_network_*           (...),
    .m_network_*           (...),
    // MonBus output
    .monbus_pkt_*       (...)
);
```

28.8 6. Use Cases

28.8.1 6.1 DMA-Style Transfers

Scenario: Move data from network to system memory

Flow: 1. Software writes descriptor to Descriptor Engine 2. Scheduler parses descriptor, activates Sink path 3. Network packets arrive via Network Slave 4. Data buffered in Sink SRAM 5. AXI4 Write Engine DMAs to system memory 6. Completion packet on MonBus

28.8.2 6.2 Network Packet Processing

Scenario: Read data from memory, transmit to network

Flow: 1. Descriptor specifies source address, length 2. Source AXI Reader fetches data to SRAM 3. Network Master transmits to network 4. Completion/error reporting via MonBus

28.8.3 6.3 Custom Data Path Acceleration

Educational value: Shows how to build custom accelerators - Descriptor-based control - Multi-block FSM coordination - Buffering strategies - Error handling - Performance monitoring

---

28.9 7. Test Coverage

28.9.1 7.1 Current Status

Overall: ~85% functional coverage (basic scenarios validated, descriptor engine complete)

Component	Test Coverage	Status
Scheduler	~95%	Credit encoding fixed and verified (43/43 tests passing)
Descriptor Engine	√ 100%	**All tests passing** (14/14 tests, 100% success rate)
Program Engine	~85%	Alignment tested
Sink Data Path	~75%	Basic flows working
Source Data Path	~70%	Basic flows working
SRAM Controllers	~80%	Buffer management tested
Integration	~60%	More stress testing needed

Test Location: `projects/components/rapids/dv/tests/fub_tests/` and `projects/components/rapids/dv/tests/integration_tests/`

Recent Achievements: - √ **Descriptor Engine (2025-10-13):** Achieved 100% test pass rate using continuous background monitoring pattern - 14/14 tests passing across all test levels (basic, medium, full) - All test classes passing (APB_ONLY, RDA_ONLY, MIXED) - All delay profiles passing (fast_producer, fast_consumer, fixed_delay, minimal_delay) - Applied continuous monitoring methodology for asynchronous output capture

📖 **See:** `docs/RAPIDS_Validation_Status_Report.md` for detailed results

28.9.2 7.2 Test Strategy

FUB (Functional Unit Block) Tests: - Individual block testing - Located in `projects/components/rapids/dv/tests/fub_tests/` - Focus on module-level functionality

Integration Tests: - Multi-block scenarios - Located in `projects/components/rapids/dv/tests/integration_tests/` - End-to-end data flow validation

System Tests: - Full RAPIDS operation - Located in `projects/components/rapids/dv/tests/system_tests/` - Realistic traffic patterns

---

28.10 8. Known Issues Summary

28.10.1 8.1 Critical Issues

✓ **Scheduler Credit Counter Initialization - FIXED (2025-10-11)** - **File:** `projects/components/rapids/rtl/rapids_fub/scheduler.sv:567-570` - **Issue:** Credit counter was initializing to 0 instead of using exponential encoding - **Fix Applied:** Implemented exponential credit encoding - **Status:** Fixed, awaiting test verification - **Documentation:** `known_issues/scheduler.md`

Fix Details:

```
// Previous (wrong):
r_descriptor_credit_counter <= 32'h0;

// Fixed - Exponential encoding:
// 0→1, 1→2, 2→4, 3→8, ..., 14→16384, 15→∞
r_descriptor_credit_counter <= (cfg_initial_credit == 4'hF) ?
32'hFFFFFFFF :
                               (cfg_initial_credit == 4'h0) ?
32'h00000001 :
                               (32'h1 << cfg_initial_credit);
```

Encoding Rationale: - Compact 4-bit configuration covers 1 to 16384 credits - Fine-grained control for low traffic (1, 2, 4, 8) - High-throughput support (256, 1024, 16384) - Special unlimited mode (15 → ∞) - Exponential encoding applied at initialization only; runtime operations are linear

28.10.2 8.2 Medium Priority Issues

Descriptor Engine Edge Cases: - Some stress test failures under high load - Edge case handling needs improvement - **Priority:** P2

SRAM Control Timing: - Rare timing issues in back-to-back operations - **Priority:** P2

📖 **See:** `known_issues/` directory for complete issue tracking

28.11 9. Integration Guidelines

28.11.1 9.1 Quick Start

```
miop_top #(
    .AXI_ADDR_WIDTH(32),
    .AXI_DATA_WIDTH(64),
    .Network_DATA_WIDTH(64)
) u_miop (
    // Clocking & Reset
    .aclk               (system_clk),
    .aresetn            (system_rst_n),

    // AXIL4 Control (connect to control fabric)
    .s_axil_awaddr      (ctrl_awaddr),
    .s_axil_awvalid     (ctrl_awvalid),
    .s_axil_awready     (ctrl_awready),
    // ... other AXIL signals

    // AXI4 Memory (connect to memory controller)
    .m_axi_sink_awaddr  (mem_awaddr),
    .m_axi_sink_awvalid (mem_awvalid),
    // ... AXI write channel for sink
    // ... AXI read channel for source

    // Network Network (connect to network fabric)
    .s_network_tdata       (net_rx_data),
    .s_network_tvalid      (net_rx_valid),
    // ... Network slave (receive)
    // ... Network master (transmit)

    // MonBus Output (connect to monitor aggregator)
    .monbus_pkt_valid   (miop_mon_valid),
    .monbus_pkt_ready   (miop_mon_ready),
    .monbus_pkt_data    (miop_mon_data)
);
```

28.11.2 9.2 Configuration Steps

1. **Initialize via AXIL4:**
   - Configure SRAM depths
   - Set timeout thresholds
   - Enable credit management (or disable if using workaround)
2. **Load Descriptors:**
   - Write descriptors to Descriptor Engine FIFO
   - Each descriptor specifies: address, length, control bits
3. **Enable Operation:**

- Set enable bits via AXIL4 registers
- Monitor MonBus for completion/error packets

📖 **See:** `rapids_spec/ch04_programming_models/01_programming.md` for register details

28.12 10. Development Status

28.12.1 10.1 Current Phase

Phase: Validation and Bug Fixing (In Progress)

- ✓ Core architecture implemented
- ✓ Basic functionality verified
- ✓ Scheduler credit counter bug fixed (exponential encoding implemented)
- ⏳ Credit management tests need verification (remove workarounds)
- ⏳ Stress testing ongoing
- ⏳ Edge case refinement

📖 **See:** `TASKS.md` for detailed work items

28.12.2 10.2 Roadmap

Near-Term (Q4 2025): - ✓ Fix scheduler credit counter bug (completed 2025-10-11) - ⏳ Verify credit management tests (remove workarounds) - ⏳ Complete descriptor engine stress testing - ⏳ Integration test suite expansion - ⏳ Performance benchmarking

Long-Term (2026+): - Chained descriptor support - Advanced error recovery - Performance optimizations - Multi-channel support

---

28.13 11. Performance Characteristics

28.13.1 11.1 Throughput

Target: Match network/memory interface bandwidth

Bottlenecks: - SRAM buffer size - AXI4 burst efficiency - Scheduler overhead

Optimization: - Increase SRAM depth for larger packets - Tune AXI4 burst parameters - Pipeline scheduler operations

28.13.2 11.2 Latency

Components: - Descriptor parsing: ~10 cycles - SRAM buffering: Configurable depth - AXI4 memory access: System dependent - End-to-end: Typically <100 cycles for small packets

28.13.3 11.3 Resource Utilization

Area: - Scheduler: ~2K LUTs - Each data path: ~3K LUTs - SRAM buffers: Configurable (dominant area) - Total: ~10K LUTs + SRAM

Power: - Clock gating opportunities in idle blocks - SRAM power depends on depth/width

---

28.14 12. Verification Infrastructure

28.14.1 12.1 Test Organization

Location: `projects/components/rapids/dv/tests/`

Structure:

```
projects/components/rapids/dv/tests/
├── fub_tests/              # Functional Unit Block tests
│   ├── scheduler/
│   ├── descriptor_engine/
│   ├── program_engine/
│   ├── network_interfaces/
│   └── simple_sram/
├── integration_tests/      # Multi-block scenarios
└── system_tests/           # Full RAPIDS operation
```

28.14.2 12.2 CocoTB Framework

Location: `bin/CocoTBFramework/tbclasses/rapids/`

Components: - RAPIDS-specific drivers - Descriptor generators - Traffic patterns - Monitor checkers

📖 **See:** `docs/markdown/CocoTBFramework/` (once created)

28.14.3 12.2.1 MANDATORY: BFM Usage for FUB Tests

⚠️ **CRITICAL DESIGN REQUIREMENT** ⚠️

All RAPIDS FUB (Functional Unit Block) level tests MUST use CocoTB Framework BFMs. Manual handshake driving is NOT allowed.

Required BFM Components:

Interface Type	Framework Location	BFM Component
Custom valid/ready	`bin/CocoTBFramework/components/gaxi/`	GAXI Master/Slave
AXI4	`bin/CocoTBFramework/components/axi4/`	AXI4 Master/Slave
AXI4-Lite (AXIL)	`bin/CocoTBFramework/components/axil4/`	AXIL Master/Slave
APB	`bin/CocoTBFramework/components/apb/`	APB Master/Slave
AXI-Stream (AXIS)	`bin/CocoTBFramework/components/axis4/`	AXIS Master/Slave
Network	`bin/CocoTBFramework/components/network/`	Network Master/Slave
MonBus	`bin/CocoTBFramework/components/monbus/`	MonBus drivers

Rationale: 1. **Consistency**: All tests use standardized handshake protocols 2. **Correctness**: BFMs handle complex timing scenarios (backpressure, randomization) 3. **Reusability**: Same BFM across all RAPIDS tests 4. **Maintainability**: Fix once in BFM, all tests benefit 5. **Coverage**: BFMs include comprehensive timing profiles

Example - Program Engine:

```
# ✗ WRONG: Manual handshake (violates design requirement)
async def send_request(self, addr, data):
    self.dut.program_valid.value = 1
    self.dut.program_pkt_addr.value = addr
    # ... manual handshaking logic ...

# ✓ CORRECT: Use GAXI Master BFM
from CocoTBFramework.components.gaxi.gaxi_master import GAXIMaster

class ProgramEngineTB(TBBase):
    def __init__(self, dut):
        super().__init__(dut)
        self.program_master = GAXIMaster(
            dut=dut,
            clock=dut.clk,
            valid_signal='program_valid',
            ready_signal='program_ready',
```

```
            data_signals=['program_pkt_addr', 'program_pkt_data'],
            data_widths=[64, 32]
        )

    async def send_request(self, addr, data):
        await self.program_master.write({'program_pkt_addr': addr,
'program_pkt_data': data})
```

📖 **See:** - `projects/components/rapids/CLAUDE.md` - Rule #1 for complete BFM usage guidelines - `bin/CocoTBFramework/components/gaxi/README.md` - GAXI BFM documentation - `bin/CocoTBFramework/components/axi4/README.md` - AXI4 BFM documentation

28.14.4 12.3 Test File Structure (Standard Pattern)

⚠️ MANDATORY: All RAPIDS tests must follow this structure ⚠️

RAPIDS tests follow the same pattern as AMBA tests for consistency across the repository:

```
# Example:
projects/components/rapids/dv/tests/fub_tests/scheduler/test_scheduler
.py

import os
import pytest
import cocotb
from cocotb_test.simulator import run

# Import REUSABLE testbench class (NOT defined in this file!)
from CocoTBFramework.tbclasses.rapids.scheduler_tb import SchedulerTB
from CocoTBFramework.tbclasses.shared.utilities import get_paths,
create_view_cmd
from CocoTBFramework.tbclasses.shared.tbbase import TBBase

#
==========================================================================
=====
# COCOTB TEST FUNCTIONS - prefix with "cocotb_" to prevent pytest
collection
#
==========================================================================
=====

@cocotb.test(timeout_time=100, timeout_unit="ms")
async def cocotb_test_basic_flow(dut):
    """Test basic descriptor flow."""
    tb = SchedulerTB(dut)
```

```
    await tb.setup_clocks_and_reset()  # Mandatory init method
    await tb.initialize_test()
    result = await tb.test_basic_descriptor_flow()
    assert result, "Basic descriptor flow test failed"

@cocotb.test(timeout_time=100, timeout_unit="ms")
async def cocotb_test_credit_encoding(dut):
    """Test exponential credit encoding."""
    tb = SchedulerTB(dut)
    await tb.setup_clocks_and_reset()  # Mandatory init method
    await tb.initialize_test()
    result = await tb.test_exponential_encoding_all_values()
    assert result, "Credit encoding test failed"

#
================================================================================
=====
# PARAMETER GENERATION - at bottom of file
#
================================================================================
=====

def generate_scheduler_test_params():
    """Generate test parameters for scheduler tests."""
    return [
        # (channel_id, num_channels, data_width, credit_width)
        (0, 8, 512, 8),  # Standard configuration
        # Add more parameter sets as needed
    ]

scheduler_params = generate_scheduler_test_params()

#
================================================================================
=====
# PYTEST WRAPPER FUNCTIONS - at bottom of file
#
================================================================================
=====

@pytest.mark.parametrize("channel_id, num_channels, data_width,
credit_width", scheduler_params)
def test_basic_flow(request, channel_id, num_channels, data_width,
credit_width):
    """
    Scheduler basic flow test.

    Run with: pytest
```

```
projects/components/rapids/dv/tests/fub_tests/scheduler/test_scheduler
.py::test_basic_flow -v
    """
    module, repo_root, tests_dir, log_dir, rtl_dict = get_paths({
        'rtl_rapids_fub': '../../rtl/rapids_fub'
    })

    dut_name = "scheduler"
    toplevel = dut_name

    verilog_sources = [
        os.path.join(repo_root, 'rtl', 'amba', 'includes',
'monitor_pkg.sv'),
        os.path.join(repo_root, 'rtl', 'rapids', 'includes',
'rapids_pkg.sv'),
        os.path.join(rtl_dict['rtl_rapids_fub'], f'{dut_name}.sv'),
    ]

    # Format parameters for unique test name
    cid_str = TBBase.format_dec(channel_id, 2)
    nc_str = TBBase.format_dec(num_channels, 2)
    dw_str = TBBase.format_dec(data_width, 4)
    cw_str = TBBase.format_dec(credit_width, 2)
    test_name_plus_params =
f"test_{dut_name}_cid{cid_str}_nc{nc_str}_dw{dw_str}_cw{cw_str}"

    # Add worker ID for pytest-xdist parallel execution
    worker_id = os.environ.get('PYTEST_XDIST_WORKER', '')
    if worker_id:
        test_name_plus_params = f"{test_name_plus_params}_{worker_id}"

    log_path = os.path.join(log_dir, f'{test_name_plus_params}.log')
    sim_build = os.path.join(tests_dir, 'local_sim_build',
test_name_plus_params)
    os.makedirs(sim_build, exist_ok=True)
    os.makedirs(log_dir, exist_ok=True)

    rtl_parameters = {
        'CHANNEL_ID': channel_id,
        'NUM_CHANNELS': num_channels,
        'DATA_WIDTH': data_width,
        'CREDIT_WIDTH': credit_width,
        # Add other RTL parameters as needed
    }

    extra_env = {
        'LOG_PATH': log_path,
        'TEST_CHANNEL_ID': str(channel_id),
```

```
        'TEST_NUM_CHANNELS': str(num_channels),
        'TEST_DATA_WIDTH': str(data_width),
    }

    compile_args = ["-Wno-TIMESCALEMOD"]
    sim_args = []
    plusargs = []

    cmd_filename = create_view_cmd(log_dir, log_path, sim_build,
module, test_name_plus_params)

    try:
        run(
            python_search=[tests_dir],
            verilog_sources=verilog_sources,
            includes=[
                os.path.join(repo_root, 'rtl', 'rapids', 'includes'),
                os.path.join(repo_root, 'rtl', 'amba', 'includes'),
            ],
            toplevel=toplevel,
            module=module,
            testcase="cocotb_test_basic_flow",  # ← cocotb test
function name
            parameters=rtl_parameters,
            sim_build=sim_build,
            extra_env=extra_env,
            waves=False,
            keep_files=True,
            compile_args=compile_args,
            sim_args=sim_args,
            plusargs=plusargs,
        )

        print(f"✓ Scheduler basic flow test completed!")
        print(f"Logs: {log_path}")

    except Exception as e:
        print(f"✗ Scheduler basic flow test failed: {str(e)}")
        print(f"Logs preserved at: {log_path}")
        raise
```

Key Structure Requirements:

1. **Testbench Class Location:**
   - ALWAYS in `bin/CocoTBFramework/tbclasses/rapids/`
   - NEVER inline in test file
   - Reusable across multiple test files

2. **CocoTB Test Functions:**
   - Prefix with `cocotb_` to prevent pytest collection
   - Located at top of test file
   - Use `@cocotb.test()` decorator
   - Call testbench methods
3. **Parameter Generation:**
   - Function returns list of parameter tuples
   - Located near bottom of file (before pytest wrappers)
   - Stored in variable (e.g., `scheduler_params`)
4. **Pytest Wrapper Functions:**
   - Located at bottom of file
   - Use `@pytest.mark.parametrize()` with parameter variable
   - Build unique test names with `TBBase.format_dec()`
   - Call `run()` with `testcase="cocotb_test_function_name"`
   - Handle parallel execution (`PYTEST_XDIST_WORKER`)
5. **Mandatory TB Methods:**
   - `async def setup_clocks_and_reset(self)` - Complete initialization
   - `async def assert_reset(self)` - Assert reset signal(s)
   - `async def deassert_reset(self)` - Deassert reset signal(s)

📖 **See:** - `val/amba/test_apb_slave.py` - Reference example - `projects/components/rapids/CLAUDE.md` - Detailed TB requirements

---

28.15 13. Quick Reference

28.15.1 13.1 Key Files

File	Purpose
`projects/components/rapids/PRD.md`	This document (overview)
`projects/components/rapids/README.md`	Quick start guide
`projects/components/rapids/CLAUDE.md`	AI assistance guide
`projects/components/rapids/TASKS.md`	Work items (to be created)
`projects/components/rapids/docs/rapids_spec/`	**Complete specification**

File	Purpose
projects/components/rapids/known_issues/	Bug tracking
docs/RAPIDS_Validation_Status_Report.md	Test results

28.15.2 13.2 Commands

```
# Run RAPIDS tests
pytest projects/components/rapids/dv/tests/fub_tests/ -v          # Individual blocks
pytest projects/components/rapids/dv/tests/integration_tests/ -v   # Multi-block
pytest projects/components/rapids/dv/tests/system_tests/ -v        # Full system

# Run specific FUB test
pytest projects/components/rapids/dv/tests/fub_tests/scheduler/ -v

# Lint RAPIDS RTL
verilator --lint-only projects/components/rapids/rtl/rapids_fub/scheduler.sv

# View specifications
cat projects/components/rapids/docs/rapids_spec/miop_index.md
cat projects/components/rapids/docs/rapids_spec/ch02_blocks/01_01_scheduler.md
```

28.16 14. Success Criteria

28.16.1 14.1 Functional

- ✓ All major blocks implemented
- ✓ Basic data flows working
- ✓ Scheduler credit bug fixed (exponential encoding implemented)
- ⏳ Credit management tests verified (remove workarounds, run full suite)
- ⏳ 100% descriptor test pass rate (currently ~80%)
- ⏳ Stress tests passing

28.16.2 14.2 Quality

- ⏳ >90% functional coverage (currently ~80%)
- ⏳ >85% code coverage

- ✓ All FSMs documented
- ⏳ Integration guide complete

28.16.3 14.3 Documentation

- ✓ Complete specification in rapids_spec/
- ✓ Known issues documented
- ⏳ Integration examples
- ⏳ Performance characterization

28.17 15. Educational Value

RAPIDS demonstrates: - ✓ Complex FSM coordination (scheduler ↔ data paths) - ✓ Descriptor-based DMA design patterns - ✓ Buffer management strategies - ✓ Credit-based flow control with exponential encoding - ✓ Multi-interface integration - ✓ Comprehensive monitoring - ✓ Error detection and reporting - ✓ Compact configuration encoding strategies

Target Audience: - Advanced RTL designers - Accelerator architects - DMA engine developers - System integration engineers

---

28.18 15. Attribution and Contribution Guidelines

28.18.1 15.1 Git Commit Attribution

When creating git commits for RAPIDS documentation or implementation:

Use:

```
Documentation and implementation support by Claude.
```

Do NOT use:

```
Co-Authored-By: Claude <noreply@anthropic.com>
```

Rationale: RAPIDS documentation and organization receives AI assistance for structure and clarity, while design concepts and architectural decisions remain human-authored.

28.19 16. Documentation Generation

28.19.1 16.1 Generating PDF/DOCX from Specification

Tool: `/mnt/data/github/rtldesignsherpa/bin/md_to_docx.py`

Use this tool to convert the linked specification index into a single all-inclusive PDF or DOCX file.

Basic Usage:

```
# Generate DOCX from rapids_spec index
python bin/md_to_docx.py \
    projects/components/rapids/docs/rapids_spec/rapids_index.md \
    -o projects/components/rapids/docs/RAPIDS_Specification_v0.25.docx
\
    --toc \
    --title-page

# Generate both DOCX and PDF
python bin/md_to_docx.py \
    projects/components/rapids/docs/rapids_spec/rapids_index.md \
    -o projects/components/rapids/docs/RAPIDS_Specification_v0.25.docx
\
    --toc \
    --title-page \
    --pdf

# With custom template (optional)
python bin/md_to_docx.py \
    projects/components/rapids/docs/rapids_spec/rapids_index.md \
    -o projects/components/rapids/docs/RAPIDS_Specification_v0.25.docx
\
    -t path/to/template.dotx \
    --toc \
    --title-page \
    --pdf
```

Key Features: - **Recursive Collection:** Follows all markdown links in the index file - **Heading Demotion:** Automatically adjusts heading levels for included files - **Table of Contents:** `--toc` flag generates automatic ToC - **Title Page:** `--title-page` flag creates title page from first heading - **PDF Export:** `--pdf` flag generates both DOCX and PDF - **Image Support:** Resolves images relative to source directory - **Template Support:** Optional custom DOCX/DOTX template via `-t` flag

Common Workflow:

```
# 1. Update version number in index file (rapids_index.md)
# 2. Generate documentation
cd /mnt/data/github/rtldesignsherpa
python bin/md_to_docx.py \
    projects/components/rapids/docs/rapids_spec/rapids_index.md \
    -o projects/components/rapids/docs/RAPIDS_Specification_v0.25.docx
\
    --toc --title-page --pdf

# 3. Output files created:
#    - RAPIDS_Specification_v0.25.docx
#    - RAPIDS_Specification_v0.25.pdf (if --pdf used)
```

Debug Mode:

```
# Generate debug markdown to see combined output
python bin/md_to_docx.py \
    projects/components/rapids/docs/rapids_spec/rapids_index.md \
    -o output.docx \
    --debug-md

# This creates debug.md showing the complete merged content
```

Tool Requirements: - Python 3.6+ - Pandoc installed and in PATH - For PDF generation: LaTeX (e.g., texlive) or use Pandoc's built-in PDF writer

📖 **See:** `/mnt/data/github/rtldesignsherpa/bin/md_to_docx.py` for complete implementation details

28.20 16.2 PDF Generation Location

IMPORTANT: PDF files should be generated in the docs directory:

`/mnt/data/github/rtldesignsherpa/projects/components/rapids/docs/`

Quick Command: Use the provided shell script:

```
cd /mnt/data/github/rtldesignsherpa/projects/components/rapids/docs
./generate_pdf.sh
```

The shell script will automatically: 1. Use the md_to_docx.py tool from bin/ 2. Process the rapids_spec index file 3. Generate both DOCX and PDF files in the docs/ directory 4. Create table of contents and title page

📖 **See:** `bin/md_to_docx.py` for complete implementation details

28.21 17. References

28.21.1 16.1 Internal Documentation

- **Complete Spec:** `rapids_spec/` ← **Primary technical reference**
- **Validation:** `docs/RAPIDS_Validation_Status_Report.md`
- **Master PRD:** `/PRD.md`
- **Repository Guide:** `/CLAUDE.md`

28.21.2 16.2 Related Subsystems

- **AMBA:** `rtl/amba/` - Monitor infrastructure used in RAPIDS
- **Common:** `rtl/common/` - Building blocks (counters, FIFOs, etc.)
- **CocoTB Framework:** `bin/CocoTBFramework/tbclasses/rapids/`

28.21.3 16.3 External References

- AXI4 Specification: ARM IHI0022E
- AXIL4 Specification: ARM IHI0022E (subset)
- Network interface specs (custom Network protocol)

Document Version: 1.0 **Last Updated:** 2025-09-30 **Review Cycle:** Monthly during active development **Next Review:** 2025-10-30 **Owner:** RTL Design Sherpa Project

---

28.22 Navigation

- **← Back to Root:** `/PRD.md`
- **Complete Specification:** `rapids_spec/miop_index.md`
- **Quick Start:** `README.md`
- **AI Guidance:** `CLAUDE.md`
- **Tasks:** `TASKS.md` (to be created)
- **Issues:** `known_issues/`

---

29 Claude Code Guide: RAPIDS Subsystem

Version: 1.0 **Last Updated:** 2025-10-11 **Purpose:** AI-specific guidance for working with RAPIDS subsystem

---

29.1 Quick Context

What: Rapid AXI Programmable In-band Descriptor System - Custom DMA-style accelerator with network interfaces **Status:** 🟡 Active development - ~80% test coverage, validation in progress **Your Role:** Help users understand architecture, fix bugs, extend functionality

📖 **Complete Specification:** `projects/components/rapids/docs/rapids_spec/` ← **Always reference this for technical details**

29.2 📖 Global Requirements Reference

IMPORTANT: Check `/GLOBAL_REQUIREMENTS.md` before starting RAPIDS work

All mandatory requirements are consolidated in the global requirements document: - **See:** `/GLOBAL_REQUIREMENTS.md` - Repository-wide mandatory requirements - **RAPIDS-Specific:** Attribution format, BFM usage requirements - **Universal:** TB location, three methods, TBBase inheritance, 100% success

This CLAUDE.md provides RAPIDS-specific guidance. Also review: - Root `/CLAUDE.md` - Repository-wide patterns - `projects/components/CLAUDE.md` - Project area standards (reset macros, FPGA attributes) - `bin/CocoTBFramework/CLAUDE.md` - Framework usage patterns

---

29.3 Critical Rules for This Subsystem

29.3.1 Rule #0: Attribution Format for Git Commits

IMPORTANT: When creating git commit messages for RAPIDS documentation or code:

Use:

```
Documentation and implementation support by Claude.
```

Do NOT use:

```
Co-Authored-By: Claude <noreply@anthropic.com>
```

Rationale: RAPIDS receives AI assistance for structure and clarity, while design concepts and architectural decisions remain human-authored.

29.3.2 Rule #0.1: Testbench Location and Test Structure (MANDATORY)

📖 **See:** `/GLOBAL_REQUIREMENTS.md` Section 2.1 for complete requirement

RAPIDS-Specific Directory Structure:

```
projects/components/rapids/dv/
├── tbclasses/                    # ★ RAPIDS TB classes (project
area!)
│   ├── scheduler_tb.py           # Scheduler testbench
│   ├── descriptor_engine_tb.py   # Descriptor engine testbench
│   └── rapids_integration_tb.py  # Integration testbench
└── tests/                        # Test runners
    ├── fub_tests/scheduler/test_scheduler.py
    ├── fub_tests/descriptor_engine/test_desc_engine.py
    └── integration_tests/test_miop_integration.py
```

RAPIDS Import Pattern:

```python
# Import framework utilities (PYTHONPATH includes bin/)
import os, sys
from CocoTBFramework.tbclasses.shared.utilities import get_repo_root
from CocoTBFramework.tbclasses.shared.tbbase import TBBase

# Add repo root to Python path using robust git-based method
repo_root = get_repo_root()
sys.path.insert(0, repo_root)

# Import RAPIDS TB from project area
from projects.components.rapids.dv.tbclasses.scheduler_tb import
SchedulerTB
```

RAPIDS Test File Organization:

```python
# 1. CocoTB functions at top (prefix "cocotb_test_*")
@cocotb.test(timeout_time=100, timeout_unit="ms")
async def cocotb_test_basic_flow(dut):
    tb = SchedulerTB(dut)
    await tb.setup_clocks_and_reset()
    # ... test logic

# 2. Parameter generation near bottom
def generate_scheduler_test_params():
```

```
    return [(0, 8, 512, 8)]  # (channel_id, num_channels, data_width,
credit_width)

scheduler_params = generate_scheduler_test_params()

# 3. Pytest wrappers at bottom
@pytest.mark.parametrize("channel_id, ...", scheduler_params)
def test_basic_flow(request, channel_id, ...):
    run(..., testcase="cocotb_test_basic_flow", ...)
```

Why RAPIDS Tests Use Project Area: 1. **Reusability:** Same TB in FUB tests, integration tests, system tests 2. **Composition:** Scheduler TB + Descriptor TB → Integration TB 3. **Discovery:** All RAPIDS code under `projects/components/rapids/`

📖 **Complete Pattern:** `val/amba/test_apb_slave.py` lines 1251-1346 (reference example)

📖 **Queue-Based Verification Pattern:** See `/GLOBAL_REQUIREMENTS.md` Section 2.4

RAPIDS-Specific Usage:

RAPIDS uses queue-based verification for in-order verification of program engine, descriptor engine, and AXI transactions.

Example - Program Engine Verification:

```
# Direct queue access for in-order RAPIDS operations
class ProgramEngineTB(TBBase):
    async def verify_write_operation(self, expected_addr,
expected_data):
        """RAPIDS program engine: In-order writes, simple queue
verification"""
        # Wait for AXI transaction
        await self.wait_for_transaction(self.aw_monitor)

        # Get from queue - program engine writes are always in-order
        aw_pkt = self.aw_monitor._recvQ.popleft()
        w_pkt = self.w_monitor._recvQ.popleft()

        # Verify
        assert aw_pkt.addr == expected_addr
        assert w_pkt.data == expected_data
```

When RAPIDS Uses Memory Models: - ✗ Descriptor engine (in-order) - Use queue access - ✗ Program engine (in-order) - Use queue access - ✓ Integration tests with multiple masters - Memory model tracks state

📖 **Complete Patterns:** `docs/VERIFICATION_ARCHITECTURE_GUIDE.md`

29.3.3 Rule #0.5: Three Mandatory TB Methods (MANDATORY)

📖 **See:** `/GLOBAL_REQUIREMENTS.md` Section 2.2 for complete requirement

RAPIDS-Specific Context:

Many RAPIDS modules (especially scheduler and credit management) require configuration signals set BEFORE reset is released. This is because some counters/state are initialized during reset based on these config values.

Critical Example - Exponential Credit Encoding:

```
async def setup_clocks_and_reset(self):
    """RAPIDS-specific: Config signals MUST be set before reset"""
    # Start clock
    await self.start_clock('clk', freq=10, units='ns')

    # ⚠️ CRITICAL: Set exponential credit config BEFORE reset
    # Scheduler's credit counter initializes during reset based on
this value
    self.dut.cfg_initial_credit.value = 4  # 4 = 16 credits (2^4)
    self.dut.cfg_use_credit.value = 1

    # Now perform reset sequence
    await self.assert_reset()
    await self.wait_clocks('clk', 10)
    await self.deassert_reset()
    await self.wait_clocks('clk', 5)

async def assert_reset(self):
    """Assert active-low reset"""
    self.dut.rst_n.value = 0

async def deassert_reset(self):
    """Release active-low reset"""
    self.dut.rst_n.value = 1
```

Why Config-Before-Reset Matters for RAPIDS: - Scheduler credit counter: Initialized to `(1 << cfg_initial_credit)` during reset - Descriptor engine depth: Configuration read during reset sequence - SRAM parameters: Must be stable before memory controllers come out of reset

Common RAPIDS Configs to Set Before Reset: - `cfg_initial_credit` - Exponential credit encoding (0→1, 1→2, 2→4, etc.) - `cfg_use_credit` - Enable/disable credit-based flow control - `cfg_timeout_threshold` - Watchdog timer configuration - `cfg_sram_depth` - SRAM buffer size parameters

---

29.3.4 Rule #0.75: Audit Signal Naming BEFORE Writing Testbenches

⚠️ CRITICAL: Check for Signal Naming Conflicts BEFORE Factory Usage ⚠️

Before writing testbench code that uses AXI factory functions, audit your RTL for signal naming conflicts.

The Problem: AXI factory pattern matching searches for signals using prefix + channel patterns (e.g., `{prefix}ar_valid`, `{prefix}r_ready`). If you have: - Internal signals: `desc_valid`, `desc_ready` (simple handshake) - External AXI ports: `desc_ar_valid`, `desc_ar_ready` (AXI AR channel)

Both match `desc_*valid` → Factory finds BOTH signals → Initialization FAILS!

Solution - Signal Naming Audit Tool:

```
# Audit single file before writing testbench
../../bin/audit_signal_naming_conflicts.py
projects/components/rapids/rtl/rapids_macro/scheduler_group.sv

# Audit entire RAPIDS subsystem
../../bin/audit_signal_naming_conflicts.py
projects/components/rapids/rtl/

# Generate markdown report
../../bin/audit_signal_naming_conflicts.py
projects/components/rapids/rtl/ --markdown signal_conflicts.md
```

Known RAPIDS Conflicts: See `projects/components/rapids/known_issues/scheduler_group_signal_naming_conflicts.md` for documented conflicts in scheduler_group.sv: - `desc` prefix: 4 internal + 4 external (AR/R channels) - `prog` prefix: 4 internal + 6 external (AW/W/B channels)

Recommended Workflow: 1. ✓ Write RTL module with signals 2. ✓ **Run audit script to detect conflicts** 3. ✓ Fix naming conflicts (rename internal signals with `_to_sched` suffix) 4. ✓ Write testbench using factory pattern matching

Three Solutions When Conflicts Found: 1. **Rename internal signals** (recommended): `desc_valid` → `desc_to_sched_valid` 2. **Use explicit signal_map**: Bypass pattern matching with manual signal mapping 3. **Test at higher level**: Where internal signals aren't visible (e.g., miop_top)

📖 **Complete Guide:** `bin/SIGNAL_NAMING_AUDIT.md`

29.3.5 Rule #1: MANDATORY BFM Usage for FUB-Level Tests

⚠️ CRITICAL DESIGN INTENTION - USE FRAMEWORK BFMs ⚠️

For all RAPIDS FUB (Functional Unit Block) level tests, you MUST use appropriate BFMs from the CocoTB Framework:

Interface Type → Required BFM Component:

Interface Type	Framework Component Location	Usage
Custom valid/ready	`bin/CocoTBFramework/components/gaxi/`	**GAXI Master/Slave BFMs**
AXI4	`bin/CocoTBFramework/components/axi4/`	AXI4 Master/Slave drivers
AXI4-Lite (AXIL)	`bin/CocoTBFramework/components/axil4/`	AXIL Master/Slave drivers
APB	`bin/CocoTBFramework/components/apb/`	APB Master/Slave drivers
AXI-Stream (AXIS)	`bin/CocoTBFramework/components/axis4/`	AXIS Master/Slave drivers
Network	`bin/CocoTBFramework/components/network/`	Network Master/Sl

Interface Type	Framework Component Location	Usage
		ave drivers
MonBus	`bin/CocoTBFramework/components/monbus/`	MonBus drivers

Critical Rules:

1. **NEVER manually drive valid/ready handshakes** - Use GAXI BFM components
2. **NEVER create custom protocol drivers** - Framework components already exist
3. **NEVER embed simple handshake logic** - Extract to GAXI Master/Slave

Example - Program Engine Interface:

```
# ✗ WRONG: Manual handshake driving
async def send_program_request(self, addr, data):
    # Manually driving program_valid/program_ready - DON'T DO THIS!
    self.dut.program_valid.value = 1
    self.dut.program_pkt_addr.value = addr
    self.dut.program_pkt_data.value = data
    while int(self.dut.program_ready.value) == 0:
        await self.wait_clocks(self.clk_name, 1)
    await self.wait_clocks(self.clk_name, 1)
    self.dut.program_valid.value = 0

# ✓ CORRECT: Use GAXI Master BFM
from CocoTBFramework.components.gaxi.gaxi_master import GAXIMaster

class ProgramEngineTB(TBBase):
    def __init__(self, dut):
        super().__init__(dut)
        # Create GAXI master for program interface
        self.program_master = GAXIMaster(
            dut=dut,
            clock=dut.clk,
            valid_signal='program_valid',
            ready_signal='program_ready',
            data_signals=['program_pkt_addr', 'program_pkt_data'],
            data_widths=[64, 32],
            name='program_interface'
        )

    async def send_program_request(self, addr, data):
        # Use GAXI master for proper handshaking
```

```
        await self.program_master.write({'program_pkt_addr': addr,
'program_pkt_data': data})
```

Why This Rule Exists:

1. **Consistency**: All tests use same handshake protocol
2. **Correctness**: GAXI BFMs handle complex timing scenarios correctly
3. **Reusability**: Same BFM used across all RAPIDS tests
4. **Maintainability**: Fix once in BFM, all tests benefit
5. **Coverage**: GAXI BFMs include timing randomization and backpressure

When Manual Driving is Acceptable:

- ✓ **Quick debug/prototype** - Temporary, will be replaced with BFM
- ✓ **Clock/reset initialization** - Not part of protocol handshaking
- ✗ **Production testbenches** - MUST use BFMs

Search Before Creating:

```
# Find existing GAXI components
find bin/CocoTBFramework/components/gaxi/ -name "*.py"

# Find example usage
grep -r "GAXIMaster\|GAXISlave"
projects/components/rapids/dv/tests/fub_tests/
grep -r "from.*gaxi" bin/CocoTBFramework/tbclasses/
```

BFM Selection Guide:

```
Need to drive interface with valid/ready?
├─ Standard protocol (AXI4, APB, AXIS, Network)?
│  └─ Use protocol-specific BFM (axi4/, apb/, axis4/, network/)
│
├─ Custom RAPIDS interface with valid/ready signals?
│  └─ Use GAXI Master/Slave BFM (gaxi/)
│
└─ MonBus monitor packet interface?
   └─ Use MonBus BFM (monbus/)
```

📖 **See:** - `bin/CocoTBFramework/components/gaxi/README.md` - GAXI BFM documentation - `bin/CocoTBFramework/components/axi4/README.md` - AXI4 BFM documentation - `val/amba/test_*.py` - Reference examples using framework BFMs

---

29.3.6 Rule #2: Always Reference Detailed Specification

This subsystem has extensive documentation in `projects/components/rapids/docs/rapids_spec/`

Before answering technical questions:

```
# Check complete specification
ls projects/components/rapids/docs/rapids_spec/
cat projects/components/rapids/docs/rapids_spec/miop_index.md
cat
projects/components/rapids/docs/rapids_spec/ch02_blocks/01_01_schedule
r.md
```

Your answer should: 1. Provide direct answer/code 2. **Then link to detailed spec:** "See `projects/components/rapids/docs/rapids_spec/{file}.md` for complete specification"

29.3.7 Rule #3: Know the Known Issues

Fixed Critical Issue: - √ **Scheduler Credit Counter - FIXED** (projects/components/rapids/rtl/rapids_fub/scheduler.sv:567-570) - Was: Credit counter hardcoded to 0 - Now: Implements exponential encoding (0→1, 1→2, 2→4, ..., 15→∞) - Impact: Credit-based flow control now functional with proper encoding - Tests: Verify exponential decoding works correctly - Priority: P0 (Critical - COMPLETED)

Always check: `projects/components/rapids/known_issues/` before diagnosing bugs

```
ls projects/components/rapids/known_issues/
cat projects/components/rapids/known_issues/scheduler.md
```

29.3.8 Rule #4: RAPIDS is Complex - Understand Block Interactions

Key Interaction Patterns:

1. **Descriptor Flow:**

   ```
   Software → AXIL4 → Descriptor Engine → Scheduler → Data Paths
   ```

2. **Sink Data Path (Network → Memory):**

   ```
   Network Slave → Sink SRAM Control → Sink AXI Write Engine →
   System Memory
   ```

3. **Source Data Path (Memory → Network):**

```
System Memory → Source AXI Read Engine → Source SRAM Control →
Network Master
```

4. **Monitoring:**

```
All Blocks → MonBus Reporter → MonBus Output
```

Never work on one block in isolation without understanding its upstream/downstream dependencies!

29.3.9 Rule #5: Test Strategy is Multi-Layered

RAPIDS testing follows this hierarchy:

1. **FUB (Functional Unit Block) Tests:** `projects/components/rapids/dv/tests/fub_tests/`
   - Individual block testing
   - Focus: Module-level functionality
   - Example: Scheduler FSM, Descriptor Engine FIFO
2. **Integration Tests:** `projects/components/rapids/dv/tests/integration_tests/`
   - Multi-block scenarios
   - Focus: Block-to-block interfaces
   - Example: Scheduler + Descriptor Engine, Sink data path end-to-end
3. **System Tests:** `projects/components/rapids/dv/tests/system_tests/`
   - Full RAPIDS operation
   - Focus: Realistic traffic patterns
   - Example: Complete DMA transfer with monitoring

When creating/modifying tests, ensure appropriate test layer is used!

29.4 Architecture Quick Reference

29.4.1 Block Organization

```
RAPIDS Architecture (17 SystemVerilog modules)
├── Scheduler Group
│   ├── scheduler.sv              (Task FSM, credit management)
│   ├── descriptor_engine.sv      (Descriptor FIFO, parsing)
│   └── program_engine.sv         (Program sequencing, alignment)
│
├── Sink Data Path (Network → Memory)
│   ├── network_slave.sv             (Network ingress)
```

```
│   ├── sink_sram_control.sv      (Buffer management)
│   └── sink_axi_write_engine.sv  (AXI4 write to system memory)
│
├── Source Data Path (Memory → Network)
│   ├── source_axi_read_engine.sv (AXI4 read from system memory)
│   ├── source_sram_control.sv    (Buffer management)
│   └── network_master.sv            (Network egress)
│
└── MonBus AXIL Group
    ├── axil4_slave.sv            (Control/status registers)
    └── monbus_reporter.sv        (Monitor packet generation)
```

📖 **See:**
`projects/components/rapids/docs/rapids_spec/ch02_blocks/00_overview.md` for detailed block descriptions

29.4.2 Module Quick Reference

Module	Location	Purpose	Documentation
scheduler.sv	`rapids_fub/`	Task FSM, credit management	`rapids_spec/ch02_blocks/01_01_scheduler.md`
descriptor_engine.sv	`rapids_fub/`	Descriptor FIFO, parsing	`rapids_spec/ch02_blocks/01_02_descriptor_engine.md`
program_engine.sv	`rapids_fub/`	Program sequencing	`rapids_spec/ch02_blocks/01_03_program_engine.md`
network_slave.sv	`rapids_fub/`	Network ingress	`rapids_spec/ch02_blocks/02_01_network_slave.md`
sink_sram_control.sv	`rapids_fub/`	Sink buffer management	`rapids_spec/ch02_blocks/02_02_sink_sram_control.md`
sink_axi_write_engine.sv	`rapids_fub/`	Memory writes	`rapids_spec/ch02_blocks/02_03_sink_axi_write_engine.md`
source_axi_read_engine.sv	`rapids_fub/`	Memory reads	`rapids_spec/ch02_blocks/03_03_source_axi_read_engine.md`
source_sram_control.sv	`rapids_fub/`	Source buffer management	`rapids_spec/ch02_blocks/03_02_source_sram_contr`

Module	Location	Purpose	Documentation
			`ol.md`
network_master.sv	`rapids_fub/`	Network egress	`rapids_spec/ch02_blocks/03_01_network_master.md`
miop_top.sv	`rapids_macro/`	Top-level integration	`rapids_spec/ch03_interfaces/01_top_level.md`

29.4.3 Interface Summary

Interface	Type	Width	Purpose	Specification
AXIL4	Slave	32-bit	Control/status registers	`rapids_spec/ch03_interfaces/02_axil_interface_spec.md`
AXI4 (Sink)	Master	Configurable	Write to system memory	`rapids_spec/ch03_interfaces/03_axi4_interface_spec.md`
AXI4 (Source)	Master	Configurable	Read from system memory	`rapids_spec/ch03_interfaces/03_axi4_interface_spec.md`
Network (Sink)	Slave	Configurable	Network ingress	`rapids_spec/ch03_interfaces/04_network_interface_spec.md`
Network (Source)	Master	Configurable	Network egress	`rapids_spec/ch03_interfaces/04_network_interface_spec.md`
MonBus	Master	64-bit	Monitor packet output	`rapids_spec/ch03_interfaces/05_monbus_interface_spec.md`

29.5 Common User Questions and Responses

29.5.1 Q: "How does the scheduler work?"

A: Direct answer:

The scheduler is a complex FSM that coordinates RAPIDS operations:

1. **Idle State:** Waits for descriptors from Descriptor Engine

2. **Parse State:** Extracts descriptor fields (address, length, control)
3. **Credit Check:** Verifies credit availability (if credit mode enabled)
4. **Execute State:** Activates appropriate data path (sink or source)
5. **Monitor State:** Tracks operation progress
6. **Complete State:** Generates completion packets, updates credits

Key FSM States: - `IDLE` → `PARSE` → `CREDIT_CHECK` → `EXECUTE` → `MONITOR` → `COMPLETE` → `IDLE`

Credit Management: - √ **FIXED:** Exponential credit encoding now implemented (lines 567-570) - **Encoding:** 0→1, 1→2, 2→4, 3→8, ..., 14→16384, 15→∞ (unlimited) - **See Q&A below** for complete encoding table and details

📖 **See:** - `projects/components/rapids/docs/rapids_spec/ch02_blocks/01_01_scheduler.md` - Complete FSM specification - `projects/components/rapids/known_issues/scheduler.md` - Known bugs and workarounds

29.5.2 Q: "How do I configure RAPIDS?"

A: Configuration via AXIL4 interface:

```
// 1. Initialize RAPIDS via AXIL4 writes
// Configure SRAM depths
write_axil(ADDR_SINK_SRAM_DEPTH, 1024);
write_axil(ADDR_SOURCE_SRAM_DEPTH, 1024);

// Set timeout thresholds
write_axil(ADDR_TIMEOUT_THRESHOLD, 1000);

// Configure initial credits (exponential encoding)
// 0→1, 1→2, 2→4, 3→8, 4→16, etc.
write_axil(ADDR_INITIAL_CREDIT, 4);  // 4 = 16 credits (2^4)
write_axil(ADDR_CREDIT_ENABLE, 1);   // ✓ Enable credit mode (now
fixed!)

// 2. Load descriptors
write_descriptor(addr, length, control_bits);

// 3. Enable operation
write_axil(ADDR_ENABLE, 1);
```

📖 **See:** - `projects/components/rapids/docs/rapids_spec/ch04_programming_models/`

`01_programming.md` - Programming model - `projects/components/rapids/docs/rapids_spec/ch05_registers/01_registers.md` - Register definitions

29.5.3 Q: "What's the data flow for network to memory transfer?"

A: Sink path data flow:

```
1. Network Packet Arrives
   ↓
2. Network Slave receives packet
   - Validates packet format
   - Handshakes with Network interface
   ↓
3. Sink SRAM Control buffers data
   - Writes to SRAM
   - Manages write pointers
   - Handles backpressure
   ↓
4. Sink AXI Write Engine
   - Reads from SRAM
   - Generates AXI4 write transactions
   - Bursts data to system memory
   ↓
5. Completion Reporting
   - Generates MonBus completion packet
   - Updates scheduler state
```

Key Considerations: - SRAM acts as buffer to decouple network from memory timing - AXI4 bursts used for efficient memory access - Backpressure propagates from memory to network

📖 **See:** `projects/components/rapids/docs/rapids_spec/ch02_blocks/02_00_sink_data_path.md`

29.5.4 Q: "What's the credit counter bug and how does exponential encoding work?"

A: Scheduler credit counter exponential encoding:

The scheduler uses **exponential credit encoding** to provide a wide range of credit values (1 to 16384) with a compact 4-bit configuration:

Encoding Table: | `cfg_initial_credit` | Actual Credits | Use Case | | ———————|———————-|———-| | 0 | 1 | Minimum (2^0) | | 1 | 2 | Very low traffic (2^1) | | 2 | 4 | Low traffic (2^2) | | 3 | 8 | (2^3) | | 4 | 16 | Typical (2^4) | | 5 | 32 | (2^5) | | 6 | 64 | Medium traffic (2^6) | | 7 | 128 | (2^7) | | 8 | 256 |

High traffic (2^8) | | 10 | 1024 | Very high traffic (2^10) | | 14 | 16384 | Maximum finite (2^14) | | 15 | ∞ (0xFFFFFFFF) | Unlimited credits |

√ FIXED Implementation:

```
// projects/components/rapids/rtl/rapids_fub/scheduler.sv:567-570
// Exponential credit encoding: 0→1, 1→2, 2→4, ..., 14→16384, 15→∞
r_descriptor_credit_counter <= (cfg_initial_credit == 4'hF) ?
32'hFFFFFFFF :
                               (cfg_initial_credit == 4'h0) ?
32'h00000001 :
                               (32'h1 << cfg_initial_credit);
```

Was Broken (Before Fix):

```
r_descriptor_credit_counter <= 32'h0;  // ✗ WRONG: Hardcoded to 0
```

Impact (Before Fix): - Credit-based flow control didn't work - All operations blocked if credit mode enabled - Descriptors never processed

Encoding Rationale: - Compact 4-bit config covers wide range: 1 to 16384 credits - Fine-grained control for low traffic (1, 2, 4, 8) - High-throughput support (256, 1024, 16384) - Special unlimited mode (15 → ∞)

Important: Exponential encoding applies **only at initialization**. Once running, the counter operates linearly (increment/decrement by 1).

📖 **See:** - `projects/components/rapids/docs/rapids_spec/ch02_blocks/01_01_scheduler.md` - Complete encoding table - `projects/components/rapids/known_issues/scheduler.md` - Bug details and fix verification

29.5.5 Q: "How do I run RAPIDS tests?"

A: Multi-layered test approach:

```
# 1. FUB (Functional Unit Block) Tests - Individual blocks
pytest projects/components/rapids/dv/tests/fub_tests/scheduler/ -v
pytest
projects/components/rapids/dv/tests/fub_tests/descriptor_engine/ -v
pytest projects/components/rapids/dv/tests/fub_tests/ -v  # All FUB
tests


# 2. Integration Tests - Multi-block scenarios
pytest projects/components/rapids/dv/tests/integration_tests/ -v


# 3. System Tests - Full RAPIDS operation
```

```
pytest projects/components/rapids/dv/tests/system_tests/ -v

# Run with waveforms for debugging
pytest projects/components/rapids/dv/tests/fub_tests/scheduler/ --
vcd=debug.vcd
gtkwave debug.vcd
```

Test Organization: - **FUB tests:** Focus on individual block functionality - **Integration tests:** Verify block-to-block interfaces - **System tests:** Validate complete data flows

Current Status: ~80% functional coverage, basic scenarios validated

📖 **See:** `docs/RAPIDS_Validation_Status_Report.md` for detailed test results

29.6 Integration Patterns

29.6.1 Pattern 1: Basic RAPIDS Instantiation

```
miop_top #(
    .AXI_ADDR_WIDTH(32),
    .AXI_DATA_WIDTH(64),
    .Network_DATA_WIDTH(64),
    .SRAM_DEPTH(1024),
    .MAX_DESCRIPTORS(16)
) u_miop (
    // Clock and Reset
    .aclk               (system_clk),
    .aresetn            (system_rst_n),

    // AXIL4 Control Interface
    .s_axil_awaddr      (ctrl_awaddr),
    .s_axil_awvalid     (ctrl_awvalid),
    .s_axil_awready     (ctrl_awready),
    .s_axil_wdata       (ctrl_wdata),
    .s_axil_wstrb       (ctrl_wstrb),
    .s_axil_wvalid      (ctrl_wvalid),
    .s_axil_wready      (ctrl_wready),
    .s_axil_bresp       (ctrl_bresp),
    .s_axil_bvalid      (ctrl_bvalid),
    .s_axil_bready      (ctrl_bready),
    .s_axil_araddr      (ctrl_araddr),
    .s_axil_arvalid     (ctrl_arvalid),
    .s_axil_arready     (ctrl_arready),
    .s_axil_rdata       (ctrl_rdata),
    .s_axil_rresp       (ctrl_rresp),
    .s_axil_rvalid      (ctrl_rvalid),
    .s_axil_rready      (ctrl_rready),
```

```
    // AXI4 Memory Interface (Sink - Write)
    .m_axi_sink_awaddr  (mem_sink_awaddr),
    .m_axi_sink_awlen   (mem_sink_awlen),
    .m_axi_sink_awsize  (mem_sink_awsize),
    .m_axi_sink_awburst (mem_sink_awburst),
    .m_axi_sink_awvalid (mem_sink_awvalid),
    .m_axi_sink_awready (mem_sink_awready),
    // ... additional AXI4 sink write channel signals

    // AXI4 Memory Interface (Source - Read)
    .m_axi_source_araddr  (mem_source_araddr),
    .m_axi_source_arlen   (mem_source_arlen),
    .m_axi_source_arsize  (mem_source_arsize),
    .m_axi_source_arburst (mem_source_arburst),
    .m_axi_source_arvalid (mem_source_arvalid),
    .m_axi_source_arready (mem_source_arready),
    // ... additional AXI4 source read channel signals

    // Network Network Interface (Sink - Receive)
    .s_network_tdata       (net_rx_data),
    .s_network_tvalid      (net_rx_valid),
    .s_network_tready      (net_rx_ready),
    .s_network_tlast       (net_rx_last),
    // ... additional Network sink signals

    // Network Network Interface (Source - Transmit)
    .m_network_tdata       (net_tx_data),
    .m_network_tvalid      (net_tx_valid),
    .m_network_tready      (net_tx_ready),
    .m_network_tlast       (net_tx_last),
    // ... additional Network source signals

    // MonBus Output
    .monbus_pkt_valid   (miop_mon_valid),
    .monbus_pkt_ready   (miop_mon_ready),
    .monbus_pkt_data    (miop_mon_data)
);
```

29.6.2 Pattern 2: Configuration Sequence

```
// Recommended initialization sequence
initial begin
    // 1. Assert reset
    aresetn = 0;
    repeat(10) @(posedge aclk);
    aresetn = 1;

    // 2. Configure RAPIDS via AXIL4
```

```
    axil_write(ADDR_SINK_SRAM_DEPTH, 1024);
    axil_write(ADDR_SOURCE_SRAM_DEPTH, 1024);
    axil_write(ADDR_TIMEOUT_THRESHOLD, 1000);
    axil_write(ADDR_INITIAL_CREDIT, 4);  // 4 = 16 credits (2^4,
exponential encoding)
    axil_write(ADDR_CREDIT_ENABLE, 1);  // ✓ Enable credit mode (now
fixed!)

    // 3. Load descriptors
    for (int i = 0; i < num_descriptors; i++) begin
        axil_write(ADDR_DESC_ADDR, descriptor[i].addr);
        axil_write(ADDR_DESC_LEN, descriptor[i].length);
        axil_write(ADDR_DESC_CTRL, descriptor[i].control);
        axil_write(ADDR_DESC_COMMIT, 1);
    end

    // 4. Enable RAPIDS operation
    axil_write(ADDR_ENABLE, 1);
end
```

29.6.3 Pattern 3: MonBus Integration

```
// Always add downstream FIFO for MonBus
gaxi_fifo_sync #(
    .DATA_WIDTH(64),
    .DEPTH(256)
) u_miop_mon_fifo (
    .i_clk      (aclk),
    .i_rst_n    (aresetn),
    .i_data     (monbus_pkt_data),
    .i_valid    (monbus_pkt_valid),
    .o_ready    (monbus_pkt_ready),
    .o_data     (fifo_mon_data),
    .o_valid    (fifo_mon_valid),
    .i_ready    (consumer_ready)
);
```

29.7 Anti-Patterns to Catch

29.7.1 ✗ Anti-Pattern 1: Not Understanding Exponential Credit Encoding

```
✗ WRONG:
.cfg_initial_credit(4'd16),  // Thinks this gives 16 credits - it
doesn't!

✓ CORRECTED:
"Credits use exponential encoding:
- cfg_initial_credit = 4 → 16 credits (2^4)
- cfg_initial_credit = 8 → 256 credits (2^8)
```

```
- cfg_initial_credit = 15 → ∞ credits (unlimited)
See
projects/components/rapids/docs/rapids_spec/ch02_blocks/01_01_schedule
r.md for encoding table."
```

29.7.2 ✗ Anti-Pattern 2: Insufficient SRAM Depth

```
✗ WRONG:
.SRAM_DEPTH(16)  // Too small for realistic packets

✓ CORRECTED:
"SRAM depth should match typical packet sizes.
Recommended: 1024-4096 entries depending on data width and packet
sizes."
```

29.7.3 ✗ Anti-Pattern 3: No MonBus Downstream Handling

```
✗ WRONG:
assign monbus_pkt_ready = 1'b1;  // Always ready = potential packet
loss

✓ CORRECTED:
"Connect to FIFO or proper consumer:
gaxi_fifo_sync #(.DATA_WIDTH(64), .DEPTH(256)) u_mon_fifo (
    .i_valid(monbus_pkt_valid),
    .i_data(monbus_pkt_data),
    .o_ready(monbus_pkt_ready),
    ...
);"
```

29.7.4 ✗ Anti-Pattern 4: Testing Individual Blocks in Isolation

```
✗ WRONG:
"Only test scheduler without descriptor engine"

✓ CORRECTED:
"RAPIDS blocks are tightly coupled. Always test:
1. FUB tests for basic block functionality
2. Integration tests for block interactions
3. System tests for complete flows"
```

29.8 Debugging Workflow

29.8.1 Issue: Scheduler Not Processing Descriptors

Check in order: 1. √ Credit configuration correct? (Remember exponential encoding!) 2. √ Descriptors loaded via AXIL4? 3. √ RAPIDS enabled? (`ADDR_ENABLE` = 1) 4. √ Reset properly deasserted? 5. √ Descriptor engine FIFO not empty?

Debug commands:

```
pytest projects/components/rapids/dv/tests/fub_tests/scheduler/ -v -s
# Verbose test
pytest projects/components/rapids/dv/tests/fub_tests/scheduler/ --
vcd=debug.vcd
gtkwave debug.vcd  # Inspect FSM state transitions
```

29.8.2 Issue: Data Path Stalls

Check in order: 1. √ SRAM depth sufficient? 2. √ Downstream interfaces ready? 3. √ AXI4 backpressure handling? 4. √ Network flow control? 5. √ Buffer overflow/underflow detection?

Waveform Analysis: - Check SRAM read/write pointers - Verify AXI4 handshakes - Inspect Network TREADY signals

29.8.3 Issue: MonBus Packets Not Generated

Check in order: 1. √ Operations completing successfully? 2. √ MonBus ready signal asserted? 3. √ MonBus reporter enabled? 4. √ Downstream FIFO not full?

29.9 Testing Guidance

29.9.1 Test Organization

```
projects/components/rapids/dv/tests/
├── fub_tests/                  # Individual block tests
│   ├── scheduler/
│   │   ├── test_scheduler.py   # FSM, credit management
│   │   └── conftest.py
│   ├── descriptor_engine/
│   │   ├── test_desc_engine.py # FIFO, parsing
│   │   └── conftest.py
│   ├── program_engine/
│   ├── network_interfaces/
│   └── simple_sram/
│
├── integration_tests/          # Multi-block scenarios
│   ├── test_scheduler_desc.py  # Scheduler + Descriptor Engine
│   ├── test_sink_path.py       # Complete sink data path
│   └── test_source_path.py     # Complete source data path
│
└── system_tests/               # Full RAPIDS operation
    ├── test_full_dma.py        # Complete DMA transfer
    └── test_bidirectional.py   # Simultaneous sink/source
```

29.9.2 Running Tests

```
# Single block test
pytest projects/components/rapids/dv/tests/fub_tests/scheduler/ -v

# All FUB tests
pytest projects/components/rapids/dv/tests/fub_tests/ -v

# Integration tests
pytest projects/components/rapids/dv/tests/integration_tests/ -v

# System tests
pytest projects/components/rapids/dv/tests/system_tests/ -v

# All RAPIDS tests
pytest projects/components/rapids/dv/tests/ -v

# With waveforms
pytest projects/components/rapids/dv/tests/fub_tests/scheduler/ --
vcd=waves.vcd
```

29.9.3 Test Coverage Status

Current Status: ~80% functional coverage

Component	Coverage	Status
Scheduler	~90%	Exponential credit encoding implemented
Descriptor Engine	~80%	Basic tests passing
Program Engine	~85%	Alignment tested
Sink Data Path	~75%	Basic flows working
Source Data Path	~70%	Basic flows working
Integration	~60%	More stress testing needed

29.10 Key Documentation Links

29.10.1 Always Reference These

Primary Technical Specification: - `projects/components/rapids/docs/rapids_spec/miop_index.md` - Complete specification index -

`projects/components/rapids/docs/rapids_spec/ch01_overview/` - Architecture overview - `projects/components/rapids/docs/rapids_spec/ch02_blocks/` - Block specifications - `projects/components/rapids/docs/rapids_spec/ch03_interfaces/` - Interface specifications - `projects/components/rapids/docs/rapids_spec/ch04_programming_models/` - Programming model - `projects/components/rapids/docs/rapids_spec/ch05_registers/` - Register definitions

This Subsystem: - `projects/components/rapids/PRD.md` - Requirements overview - `projects/components/rapids/README.md` - Quick start guide (if exists) - `projects/components/rapids/TASKS.md` - Current work items - `projects/components/rapids/known_issues/` - Bug tracking

Validation: - `docs/RAPIDS_Validation_Status_Report.md` - Test results and status

Root: - `/PRD.md` - Master requirements - `/CLAUDE.md` - Repository guide

29.11 Quick Commands

```
# View complete specification
cat projects/components/rapids/docs/rapids_spec/miop_index.md
cat
projects/components/rapids/docs/rapids_spec/ch02_blocks/01_01_schedule
r.md

# Check known issues
ls projects/components/rapids/known_issues/
cat projects/components/rapids/known_issues/scheduler.md

# Run tests
pytest projects/components/rapids/dv/tests/fub_tests/ -v
pytest projects/components/rapids/dv/tests/integration_tests/ -v

# Lint
verilator --lint-only
projects/components/rapids/rtl/rapids_fub/scheduler.sv

# Search for modules
find projects/components/rapids/rtl/rapids_fub/ -name "*.sv" -exec
grep -H "^module" {} \;
```

29.12 Verification Patterns and Best Practices

29.12.1 Pattern: Continuous Background Monitoring

Problem: When testing components that output data asynchronously (descriptors, packets, etc.), tests that only check outputs at specific points will miss data that arrives during other operations.

Solution: Use continuous background monitoring coroutines that run throughout the test.

Example - Descriptor Engine Testing:

```
class DescriptorEngineTB(TBBase):
    async def run_apb_only_test(self, num_packets: int, profile:
DelayProfile):
        """Test with continuous descriptor monitoring"""
        descriptors_collected = []  # Shared list
        monitor_active = True  # Control flag

        # Background coroutine monitors continuously
        async def descriptor_monitor():
            """Continuously monitor for descriptors throughout the
test"""
            while monitor_active:
                await self.wait_clocks(self.clk_name, 1)
                if int(self.dut.descriptor_valid.value) == 1:
                    desc_data = int(self.dut.descriptor_packet.value)
                    descriptors_collected.append(desc_data)
                    if len(descriptors_collected) % 5 == 1:
                        self.log.info(f"📦 Descriptor
{len(descriptors_collected)}: 0x{desc_data:X}")

        # Start background monitor
        monitor_task = cocotb.start_soon(descriptor_monitor())

        # Send all requests (monitor captures outputs during this
phase)
        for i in range(num_packets):
            # Send APB request
            self.dut.apb_valid.value = 1
            self.dut.apb_addr.value = test_addr
            await self.wait_clocks(self.clk_name, 1)
            # ... wait for ready ...
            self.dut.apb_valid.value = 0
```

```
        # Wait for final descriptors (monitor still running)
        for cycle in range(final_window):
            await self.wait_clocks(self.clk_name, 1)
            if len(descriptors_collected) >= num_packets:
                break

        # Stop background monitor
        monitor_active = False
        await self.wait_clocks(self.clk_name, 2)  # Let monitor finish

        # Verify results
        return len(descriptors_collected) == num_packets
```

Key Points: 1. **Shared State:** Use list/dict accessible from both main test and monitor coroutine 2. **Control Flag:** `monitor_active` allows clean shutdown 3. **Background Task:** `cocotb.start_soon()` runs monitor concurrently 4. **Continuous Capture:** Monitor checks every clock cycle, never missing data 5. **Clean Shutdown:** Set flag to False, wait for monitor to finish

When to Use: - √ Asynchronous output interfaces (descriptors, packets, responses) - √ Tests with rapid-fire input operations - √ Multi-stage pipelines where outputs don't align with inputs - ✗ Simple synchronous request-response patterns

Benefits: - 100% capture rate (no missed transactions) - Decouples input stimulus from output monitoring - Realistic timing coverage (captures outputs at any point)

Results: Descriptor engine tests improved from 42% success (5/12 descriptors) to 100% success (14/14 tests passing) by applying this pattern.

29.12.2 Pattern: Using Existing CocoTBFramework Components

Critical Rule: Always search for existing components before creating new ones.

Framework Structure:

```
bin/CocoTBFramework/
├── components/           # Protocol-specific BFMs and drivers
│   ├── axi4/            # Complete AXI4 infrastructure
│   ├── axil4/           # AXI4-Lite components
│   ├── apb/             # APB drivers and monitors
│   ├── axis4/           # AXI-Stream components
│   └── rapids/            # RAPIDS-specific BFMs
├── tbclasses/           # Reusable testbench classes
│   ├── amba/            # AMBA testbenches
│   ├── rapids/            # RAPIDS testbenches
```

```
│   └── shared/           # Common utilities (TBBase, etc.)
└── scoreboards/          # Transaction checkers
```

Decision Tree: Create New BFM vs Use Existing?

```
Need to model protocol behavior?
├─ Is it a standard protocol (AXI4, APB, AXIS)?
│  └─ YES → Use components/axi4/, components/apb/, etc.
│           ✓ DO NOT create new implementation
│
├─ Is it RAPIDS-specific custom behavior?
│  ├─ Is it >100 lines?
│  │  └─ YES → Extract to components/rapids/
│  └─ Is it <50 lines and test-specific?
│     └─ YES → Keep embedded in testbench
│
└─ Is it generic helper (not protocol-specific)?
   └─ Add to tbclasses/shared/utilities.py
```

Example - Descriptor Engine AXI Responder:

```python
# ✗ WRONG: Creating new AXI4 read responder BFM
# File: components/rapids/axi_read_responder_bfm.py
class AXIReadResponderBFM:
    """New AXI4 read responder - DON'T DO THIS!"""
    # ... 200 lines of AXI4 protocol implementation ...

# ✓ CORRECT: Use existing AXI4 components
# File: tbclasses/rapids/descriptor_engine_tb.py
async def axi_read_responder(self):
    """Simple test-specific responder - 50 lines, embedded"""
    while True:
        await self.wait_clocks(self.clk_name, 1)
        if int(self.dut.ar_valid.value) == 1 and
int(self.dut.ar_ready.value) == 1:
            # Generate response data
            self.dut.r_valid.value = 1
            self.dut.r_data.value = response_data
            # ... wait for r_ready ...
            self.dut.r_valid.value = 0
```

Why This Matters: - **Existing AXI4 components:** Comprehensive, tested, feature-complete - **Simple embedded responder:** Test-specific, minimal protocol simulation - **No duplication:** Framework already has robust AXI4 infrastructure

BFM Extraction Criteria:

Factor	Extract to BFM	Keep Embedded
Lines of code	>100 lines	<50 lines

Factor	Extract to BFM	Keep Embedded
Complexity	Complex protocol logic	Simple stimulus/response
Reusability	Used across multiple tests	Test-specific
Protocol	Custom RAPIDS-specific	Standard protocol (use framework)
Dependencies	Standalone	Tightly coupled to one test

Examples:

✓ **Extracted to BFM:** `DataMoverBFM` (components/rapids/data_mover_bfm.py) - 150+ lines - Complex RAPIDS data mover protocol - Used across scheduler tests - Reusable for integration tests

✓ **Kept Embedded:** AXI read responder in descriptor engine - 50 lines - Simple test-specific behavior - Framework already has full AXI4 support - No need for extraction

29.12.3 Test Scalability Patterns

Principle: Tests should scale from basic validation to comprehensive stress testing.

Pattern: Delay Profiles for Timing Coverage

```
class DelayProfile(Enum):
    """Delay profiles for comprehensive timing coverage"""
    FAST_PRODUCER = "fast_producer"      # Producer faster than
consumer
    FAST_CONSUMER = "fast_consumer"      # Consumer faster than
producer
    FIXED_DELAY = "fixed_delay"          # Predictable timing
    MINIMAL_DELAY = "minimal_delay"      # Stress test - minimal
delays
    BACKPRESSURE = "backpressure"        # Heavy backpressure
scenarios

# Test method scales with profile
async def run_apb_only_test(self, num_packets: int, profile:
DelayProfile):
    """Scalable test with configurable timing"""
    params = self.delay_params[profile]

    for i in range(num_packets):
        # Apply profile-specific delays
        producer_delay =
```

```
self.get_delay_value(params['producer_delay'])
        await self.wait_clocks(self.clk_name, producer_delay)

        # Send request...

        # Profile-specific backpressure
        if random.random() < params.get('backpressure_freq', 0.1):
            backpressure_cycles = random.randint(5, 25)
            await self.wait_clocks(self.clk_name, backpressure_cycles)
```

Pattern: Hierarchical Test Levels

```
# Test runner with multiple levels
@pytest.mark.parametrize("test_level", ["basic", "medium", "full"])
def test_descriptor_engine(test_level, ...):
    """Hierarchical test levels for different coverage needs"""

    if test_level == "basic":
        # Quick validation: 10 packets, simple timing
        num_packets = 10
        test_class = TestClass.APB_ONLY

    elif test_level == "medium":
        # Moderate coverage: 3 packets × 4 profiles
        num_packets = 3
        test_classes = [TestClass.APB_ONLY, TestClass.RDA_ONLY,
TestClass.MIXED]

    elif test_level == "full":
        # Comprehensive: 5 packets × all profiles × all test classes
        num_packets = 5
        test_classes = [TestClass.APB_ONLY, TestClass.RDA_ONLY,
TestClass.MIXED]
```

Benefits: - **Basic:** Quick smoke tests (CI/CD) - **Medium:** Developer validation - **Full:** Comprehensive regression

Pattern: Parametrized Test Generation

```
def generate_test_params():
    """Generate comprehensive parameter combinations"""
    params = []

    # Basic configurations
    for num_channels in [8, 32, 64]:
        for data_width in [64, 512]:
            for addr_width in [32, 64]:
                params.append((num_channels, data_width, addr_width))
```

```
    return params

# Pytest automatically runs all combinations
@pytest.mark.parametrize("num_channels, data_width, addr_width",
generate_test_params())
def test_scheduler(num_channels, data_width, addr_width):
    """Test scales across all parameter combinations"""
    # ...
```

Test Success Criteria:

⚠️ **CRITICAL:** All tests must achieve 100% success rate.

```
# ✗ WRONG: Accepting partial success
success_rate = (descriptors_received / total_sent) * 100
return success_rate >= 70  # "Mostly good" - NOT ACCEPTABLE

# ✓ CORRECT: Require 100% success
success_rate = (descriptors_received / total_sent) * 100
return success_rate >= 100  # Must receive ALL expected outputs
```

Why 100% Required: - Partial success indicates bugs, not acceptable tolerance - RTL should be deterministic - 100% success is achievable - Lower thresholds mask real issues

---

29.13 Documentation Generation

29.13.1 Generating PDF/DOCX from Specification

Tool: `/mnt/data/github/rtldesignsherpa/bin/md_to_docx.py`

Use this tool to convert the linked specification index into a single all-inclusive PDF or DOCX file.

Basic Usage:

```
# Generate DOCX from rapids_spec index
python bin/md_to_docx.py \
    projects/components/rapids/docs/rapids_spec/rapids_index.md \
    -o projects/components/rapids/docs/RAPIDS_Specification_v0.25.docx
\
    --toc \
    --title-page

# Generate both DOCX and PDF
```

```
python bin/md_to_docx.py \
    projects/components/rapids/docs/rapids_spec/rapids_index.md \
    -o projects/components/rapids/docs/RAPIDS_Specification_v0.25.docx
\
    --toc \
    --title-page \
    --pdf

# With custom template (optional)
python bin/md_to_docx.py \
    projects/components/rapids/docs/rapids_spec/rapids_index.md \
    -o projects/components/rapids/docs/RAPIDS_Specification_v0.25.docx
\
    -t path/to/template.dotx \
    --toc \
    --title-page \
    --pdf
```

Key Features: - **Recursive Collection:** Follows all markdown links in the index file - **Heading Demotion:** Automatically adjusts heading levels for included files - **Table of Contents:** `--toc` flag generates automatic ToC - **Title Page:** `--title-page` flag creates title page from first heading - **PDF Export:** `--pdf` flag generates both DOCX and PDF - **Image Support:** Resolves images relative to source directory - **Template Support:** Optional custom DOCX/DOTX template via `-t` flag

Common Workflow:

```
# 1. Update version number in index file (rapids_index.md)
# 2. Generate documentation
cd /mnt/data/github/rtldesignsherpa
python bin/md_to_docx.py \
    projects/components/rapids/docs/rapids_spec/rapids_index.md \
    -o projects/components/rapids/docs/RAPIDS_Specification_v0.25.docx
\
    --toc --title-page --pdf

# 3. Output files created:
#    - RAPIDS_Specification_v0.25.docx
#    - RAPIDS_Specification_v0.25.pdf (if --pdf used)
```

Debug Mode:

```
# Generate debug markdown to see combined output
python bin/md_to_docx.py \
    projects/components/rapids/docs/rapids_spec/rapids_index.md \
    -o output.docx \
    --debug-md
```

```
# This creates debug.md showing the complete merged content
```

Tool Requirements: - Python 3.6+ - Pandoc installed and in PATH - For PDF generation: LaTeX (e.g., texlive) or use Pandoc's built-in PDF writer

📖 **See:** `/mnt/data/github/rtldesignsherpa/bin/md_to_docx.py` for complete implementation details

29.14 PDF Generation Location

IMPORTANT: PDF files should be generated in the docs directory:

`/mnt/data/github/rtldesignsherpa/projects/components/rapids/docs/`

Quick Command: Use the provided shell script:

```
cd /mnt/data/github/rtldesignsherpa/projects/components/rapids/docs
./generate_pdf.sh
```

The shell script will automatically: 1. Use the md_to_docx.py tool from bin/ 2. Process the rapids_spec index file 3. Generate both DOCX and PDF files in the docs/ directory 4. Create table of contents and title page

📖 **See:** `bin/md_to_docx.py` for complete implementation details

29.15 Remember

1. 🎛️ **MANDATORY: Use BFMs** - GAXI Master/Slave for custom valid/ready, protocol-specific BFMs for AXI4/APB/AXIS/Network
2. 📖 **Link to detailed spec** - `projects/components/rapids/docs/rapids_spec/` has complete architecture docs
3. 🔢 **Exponential credit encoding** - Remember: 0→1, 1→2, 2→4, not linear!
4. 🐛 **Check known issues** - Before diagnosing bugs
5. 🔗 **Block interactions** - RAPIDS blocks are tightly coupled
6. 🧪 **Multi-layered testing** - FUB → Integration → System tests
7. 🏗️ **Testbench reuse** - Always create TB classes in `bin/CocoTBFramework/tbclasses/rapids/`
8. 🎯 **Continuous monitoring** - Use background coroutines for asynchronous output capture

9. 🔍 **Search first** - Use existing CocoTBFramework components before creating new ones
10. 📊 **100% success** - All tests must achieve 100% success rate, no exceptions
11. 🏛️ **Three-layer architecture** - TB (infrastructure) + Test (intelligence) + Scoreboard (verification)
12. 🎪 **Queue-based verification** - Use `monitor._recvQ.popleft()` for simple tests, not memory models

---

Version: 1.2 **Last Updated:** 2025-10-14 **Maintained By:** RTL Design Sherpa Project