Table of Contents

1 Rapids Beats Has Index

Generated: 2026-01-17

2 Document Information

Document Title: RAPIDS Beats Hardware Architecture Specification (HAS)
Document Number: RAPIDS-HAS-001 **Version:** 1.0 **Date:** 2026-01-17 **Status:** Released

---

2.1 Purpose

This Hardware Architecture Specification (HAS) defines the external interfaces, system integration requirements, and high-level architecture of the RAPIDS Beats accelerator. It is intended for system architects and integration engineers who need to understand RAPIDS Beats from an external perspective without requiring knowledge of internal implementation details.

2.2 Scope

This document covers:

- External interface specifications (AXI4, AXIS, APB, MonBus)
- System-level block diagrams and data flows
- Programming model and descriptor format
- Use cases and operational sequences
- Performance characteristics and constraints

This document does NOT cover:

- Internal module architecture (see MAS)
- RTL implementation details (see MAS)
- Verification test plans
- Physical design constraints

2.3 Audience

Target Audience

Role	Relevance
System Architect	Primary - system integration
Hardware Integration Engineer	Primary - interface connection
Software Engineer	Primary - driver development
Verification Engineer	Reference - interface protocols
RTL Designer	Reference - external constraints

2.4 Document Conventions

2.4.1 Requirement Levels

Term	Meaning
SHALL	Mandatory requirement
SHOULD	Recommended but not mandatory
MAY	Optional feature

2.4.2 Signal Directions

- **Input:** Signal driven by external logic into RAPIDS
- **Output:** Signal driven by RAPIDS to external logic
- **Bidirectional:** Signal can be driven by either side (rare)

2.4.3 Timing Diagrams

All timing diagrams use WaveDrom format with the following conventions:

- `p` = Positive clock edge
- `n` = Negative clock edge
- `0/1` = Logic low/high
- `x` = Unknown/don't care
- `=` = Data value (with label)
- `.` = Previous value continues

2.5 References

Reference Documents

Document	Description
RAPIDS Beats MAS	Module Architecture Specification
ARM AMBA AXI4 Specification	AXI4 protocol reference
ARM AMBA AXI-Stream Specification	AXIS protocol reference
RAPIDS PRD	Product Requirements Document

3 Revision History

Document Revision History

Version	Date	Author	Changes
1.0	2026-01-17	RTL Design Sherpa	Initial release

3.1 Change Summary

3.1.1 Version 1.0 (2026-01-17)

Initial Release

- Complete HAS for RAPIDS Beats Phase 1 architecture
- External interface specifications (AXI4, AXIS, APB, MonBus)
- System block diagrams with Mermaid
- Timing diagrams with WaveDrom
- Programming model and descriptor format
- Use case documentation

4 Acronyms and Definitions

4.1 Acronyms

Acronym Definitions

Acronym	Definition
AXI	Advanced eXtensible Interface
AXIS	AXI-Stream
APB	Advanced Peripheral Bus
DMA	Direct Memory Access
FIFO	First-In First-Out
FSM	Finite State Machine
HAS	Hardware Architecture Specification
MAS	Module Architecture Specification
MonBus	Monitor Bus
RAPIDS	Rapid AXI Programmable In-band Descriptor System
SRAM	Static Random Access Memory

4.2 Definitions

Term Definitions

Term	Definition
Beat	Single data transfer unit (512 bits = 64 bytes in default configuration)
Channel	Independent DMA context with dedicated descriptor processing
Descriptor	256-bit control structure defining a single DMA transfer
Descriptor Chain	Linked list of descriptors for multi-transfer operations
Sink Path	Data flow from network (AXIS slave) to memory (AXI write)
Source Path	Data flow from memory (AXI read) to network (AXIS

Term	Definition
	master)
Fill	Operation writing data into SRAM buffer (sink path input)
Drain	Operation reading data from SRAM buffer (source path output)

4.3 Signal Naming Conventions

Signal Naming Conventions

Prefix/Suffix	Meaning
`m_axi_*`	AXI Master signals
`s_axis_*`	AXIS Slave signals
`m_axis_*`	AXIS Master signals
`cfg_*`	Configuration signals
`*_valid`	Handshake valid signal
`*_ready`	Handshake ready signal
`*_n`	Active-low signal

5 Product Overview

5.1 Introduction

RAPIDS Beats is a high-performance network-to-memory accelerator designed for descriptor-based data movement operations. It bridges network interfaces (AXI-Stream) with system memory (AXI4), enabling efficient DMA transfers without CPU intervention.

The “Beats” architecture represents Phase 1 of the RAPIDS family, optimized for simplicity and throughput in streaming applications.

5.2 System Context

System Context Diagram

System Context Diagram

Source: 01_system_context.mmd

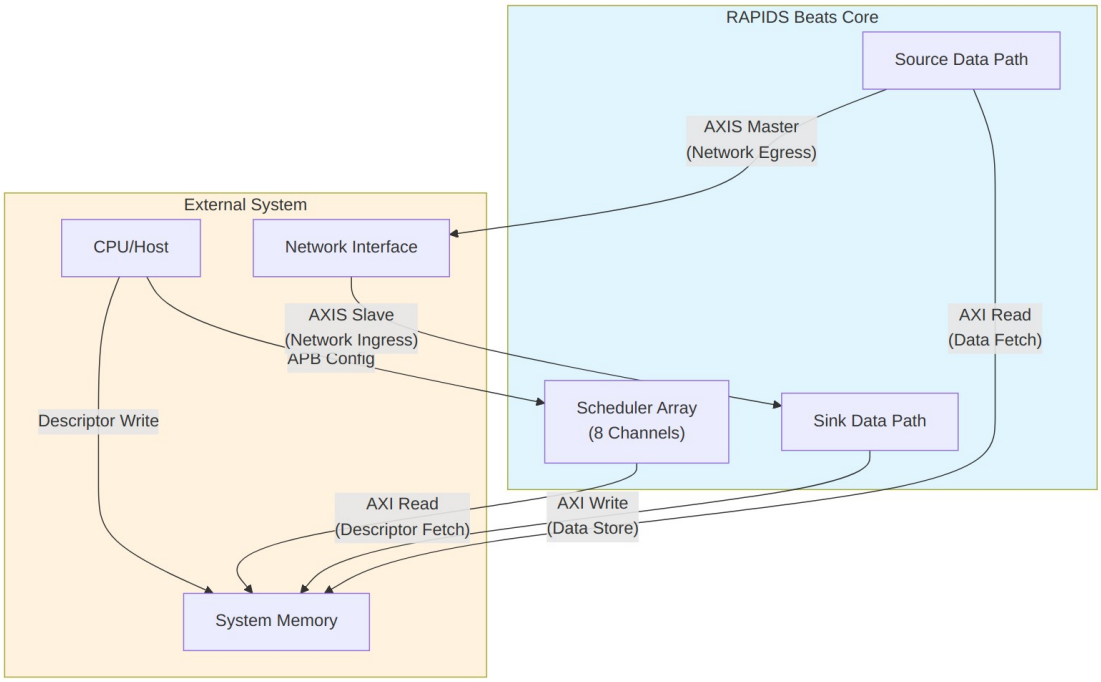


Diagram 1

5.3 Target Applications

Target Applications

Application	Description	Key Requirements
Network Offload	Receive packets from network, store to memory	Low latency, high throughput
Scatter-Gather DMA	Multi-descriptor data movement	Descriptor chaining
Streaming Accelerator	Front-end for compute engines	Continuous data flow
Protocol Processing	Header/payload separation	Multi-channel isolation

5.4 Architecture Highlights

5.4.1 Phase 1 “Beats” Simplifications

The Beats architecture prioritizes simplicity over feature completeness:

Phase 1 Simplifications

Feature	Beats (Phase 1)	Future Phases
Transfer Unit	Full beats only	Chunk-level alignment
Flow Control	Backpressure-based	Credit management
Address Alignment	Pre-aligned required	Hardware alignment fixup
Network Interface	Direct AXIS	Protocol conversion

5.4.2 Key Capabilities

1. **8 Independent Channels** - Concurrent transfers with dedicated resources
2. **Descriptor Chaining** - Automatic next-descriptor fetch
3. **Dual Data Paths** - Simultaneous sink (write) and source (read)
4. **Monitor Integration** - Real-time event reporting via MonBus
5. **Error Detection** - Address range checking, AXI error propagation

6 Key Features

6.1 Feature Summary

6.1.1 Multi-Channel Architecture

- **8 Independent Channels** with dedicated schedulers
- **Per-Channel State Machines** for concurrent operation
- **Shared AXI Infrastructure** with arbitrated access
- **Channel Isolation** prevents cross-channel interference

6.1.2 Descriptor-Based Control

- **256-bit Descriptor Format** with all transfer parameters
- **Automatic Chaining** follows next_descriptor_ptr links
- **Last Descriptor Flag** terminates chain processing
- **IRQ Generation** via gen_irq descriptor flag

6.1.3 High-Performance Data Paths

Data Path Summary

Path	Direction	Bandwidth	Features
Sink	Network to Memory	512-bit @ clock	SRAM buffering,

Path	Direction	Bandwidth	Features
			AXI burst writes
Source	Memory to Network	512-bit @ clock	AXI burst reads, SRAM buffering

6.1.4 Monitoring and Debug

- **64-bit MonBus Packets** for all significant events
- **State Transition Reports** track FSM activity
- **Error Event Codes** for fault diagnosis
- **Performance Metrics** for throughput analysis

6.2 Feature Details

6.2.1 Descriptor Processing

Descriptor Flow

Descriptor Flow

Source: 02_descriptor_flow.mmd

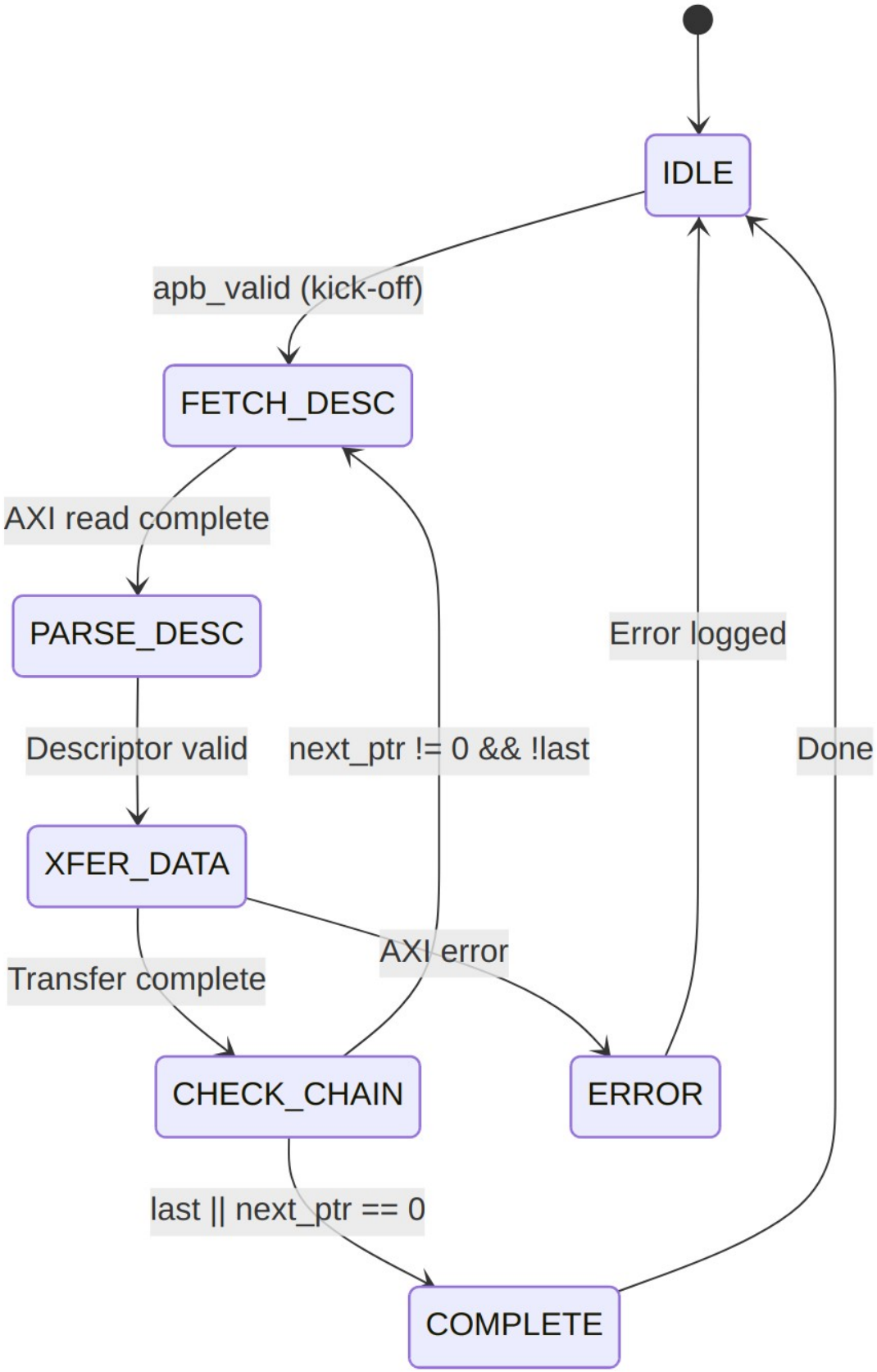

IDLE
apb_valid (kick-off)
FETCH_DESC
AXI read complete
PARSE_DESC
Descriptor valid
XFER_DATA
Transfer complete
AXI error
CHECK_CHAIN
next_ptr != 0 && !last
ERROR
Error logged
last || next_ptr == 0
COMPLETE
Done

Diagram 2

6.2.2 SRAM Buffering

Each data path includes dedicated SRAM buffering:

SRAM Buffer Parameters

Parameter	Default	Range	Description
SRAM_DEPTH	512	64-4096	Entries per channel
Data Width	512	Fixed	Bits per entry
Total Size	32KB	4KB-256KB	Per data path

Buffer Operation:

- **Sink Path:** Fill from AXIS, drain to AXI write engine
- **Source Path:** Fill from AXI read engine, drain to AXIS
- **Flow Control:** Backpressure when buffer full/empty

6.2.3 AXI Burst Optimization

RAPIDS optimizes AXI transactions for maximum bandwidth:

AXI Optimization Features

Feature	Implementation
Burst Length	Up to 256 beats (AXI4 max)
Outstanding Transactions	Configurable (default 8)
Address Alignment	64-byte aligned (beat size)
4KB Boundary	Automatic burst splitting

6.2.4 Error Handling

Error Handling Summary

Error Type	Detection	Response
Address Range	Pre-fetch validation	Descriptor rejected
AXI SLVERR	Response check	Transfer aborted
AXI DECERR	Response check	Transfer aborted
Timeout	Watchdog timer	Channel reset

7 System Context

7.1 Integration Overview

RAPIDS Beats integrates into a system as a DMA engine bridging network interfaces and system memory. This section describes the external connections and system-level requirements.

7.2 System Block Diagram

System Integration

System Integration

Source: 03_system_integration.mmd

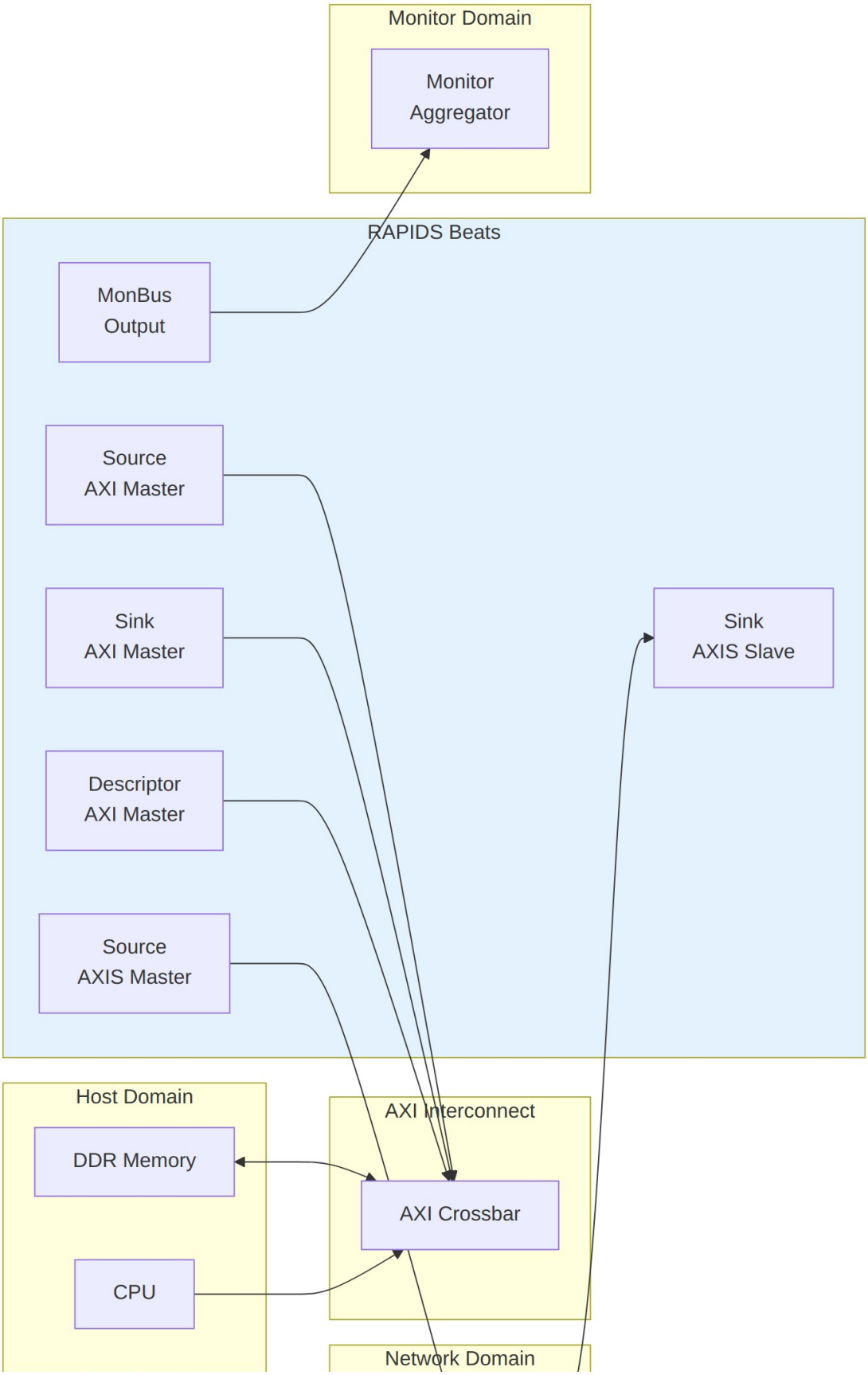

Monitor Domain
Monitor Aggregator
RAPIDS Beats
MonBus Output
Source AXI Master
Sink AXI Master
Descriptor AXI Master
Source AXIS Master
Sink AXIS Slave
Host Domain
DDR Memory
CPU
AXI Interconnect
AXI Crossbar
Network Domain

Diagram 3

7.3 Interface Connections

7.3.1 Memory Interface

RAPIDS requires three AXI4 master ports connected to system memory:

AXI Memory Interface Requirements

Port	Access Pattern	Typical Bandwidth
Descriptor AXI	Random read (256-bit)	Low (<100 MB/s)
Sink AXI	Sequential write (512-bit)	High (up to interface max)
Source AXI	Sequential read (512-bit)	High (up to interface max)

Interconnect Requirements:

- All three ports MAY share a single physical AXI port with arbitration
- Descriptor port has lowest bandwidth requirement
- Sink and Source ports should have balanced bandwidth allocation

7.3.2 Network Interface

RAPIDS uses standard AXI-Stream for network connectivity:

AXIS Network Interface

Interface	Role	Signals
Sink AXIS	Receives data from network	TDATA, TVALID, TREADY, TLAST
Source AXIS	Sends data to network	TDATA, TVALID, TREADY, TLAST

Protocol Notes:

- TKEEP/TSTRB supported for partial beat handling
- TID/TDEST may be used for channel routing (optional)
- TUSER available for sideband information

7.3.3 Configuration Interface

Software configures RAPIDS through register writes:

Configuration Methods

Method	Interface	Usage
Direct Write	Memory-mapped	Register configuration
Descriptor Kick	APB-style	Start descriptor processing

7.4 Clock and Reset

7.4.1 Clock Domains

Clock Domains

Clock	Frequency	Usage
`aclk`	100-500 MHz	All RAPIDS logic

Note: RAPIDS Beats uses a single clock domain. CDC (clock domain crossing) to different frequency domains must be handled externally.

7.4.2 Reset Requirements

Reset Signals

Signal	Type	Description
`aresetn`	Active-low	Async assert, sync deassert

Reset Sequence:

1. Assert `aresetn` low for minimum 4 clock cycles
2. Ensure all AXI interfaces are idle before reset
3. Release reset synchronously to `aclk` rising edge
4. Wait 16 clock cycles before initiating transfers

7.5 Power Considerations

7.5.1 Clock Gating

RAPIDS supports optional clock gating for power reduction:

- **Idle Detection:** Automatic when no transfers pending
- **Per-Channel Gating:** Individual channels can be clock-gated
- **External Control:** `cfg_clock_enable` override available

7.5.2 Power States

Power States

State	Description	Exit Latency
Active	Normal operation	0 cycles
Idle	No pending transfers	1 cycle
Gated	Clock disabled	4 cycles

8 Block Diagram

8.1 Top-Level Architecture

RAPIDS Beats Block Diagram

RAPIDS Beats Block Diagram

Source: 04_block_diagram.mmd

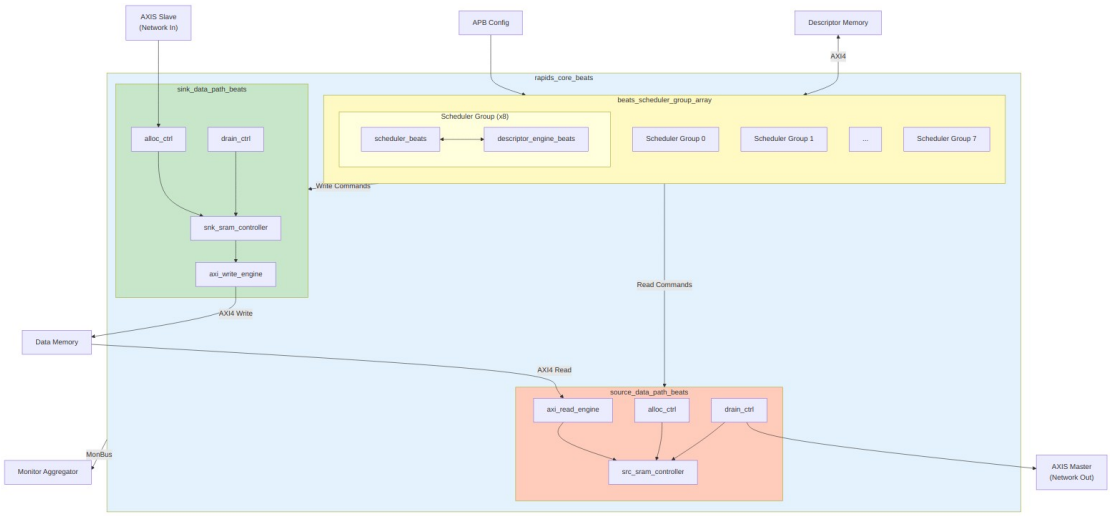


Diagram 4

8.2 Component Summary

8.2.1 Scheduler Group Array

The scheduler group array manages 8 independent channels:

Scheduler Array Components

Component	Instance Count	Purpose
`scheduler_beats`	8	Transfer

Component	Instance Count	Purpose
		coordination per channel
descriptor_engine_beats	8	Descriptor fetch and parse
Descriptor AXI Arbiter	1	Shared AXI4 for descriptor fetch

8.2.2 Sink Data Path

Network-to-memory data flow:

Sink Path Components

Component	Purpose
snk_sram_controller_beats	Multi-channel SRAM management
axi_write_engine_beats	AXI4 burst write generation
alloc_ctrl_beats	Space allocation tracking
drain_ctrl_beats	Data availability tracking

8.2.3 Source Data Path

Memory-to-network data flow:

Source Path Components

Component	Purpose
axi_read_engine_beats	AXI4 burst read generation
src_sram_controller_beats	Multi-channel SRAM management
alloc_ctrl_beats	Space allocation tracking
drain_ctrl_beats	Data availability tracking

8.3 Hierarchy

```
rapids_core_beats
├── beats_scheduler_group_array
│   ├── beats_scheduler_group [0]
│   │   ├── scheduler_beats
│   │   └── descriptor_engine_beats
│   ├── beats_scheduler_group [1..6]
```

```
│   │   └── ...
│   └── beats_scheduler_group [7]
│       └── ...
├── sink_data_path_beats
│   ├── snk_sram_controller_beats
│   │   └── snk_sram_controller_unit_beats [0..7]
│   ├── axi_write_engine_beats
│   ├── alloc_ctrl_beats
│   └── drain_ctrl_beats
└── source_data_path_beats
    ├── axi_read_engine_beats
    ├── src_sram_controller_beats
    │   └── src_sram_controller_unit_beats [0..7]
    ├── alloc_ctrl_beats
    └── drain_ctrl_beats
```

8.4 Internal Buses

Internal Bus Summary

Bus	Width	Description
Descriptor Bus	256-bit	Parsed descriptor to scheduler
Scheduler Command	Variable	Transfer parameters to data paths
SRAM Data	512-bit	Data to/from SRAM buffers
MonBus	64-bit	Monitor packets (aggregated)

9 Data Flow

9.1 Overview

RAPIDS Beats implements two independent data paths that can operate concurrently:

1. **Sink Path:** Network to Memory (AXIS Slave -> AXI Write)
2. **Source Path:** Memory to Network (AXI Read -> AXIS Master)

9.2 Sink Path Data Flow

Sink Data Flow

Sink Data Flow

Source: 05_sink_flow.mmd

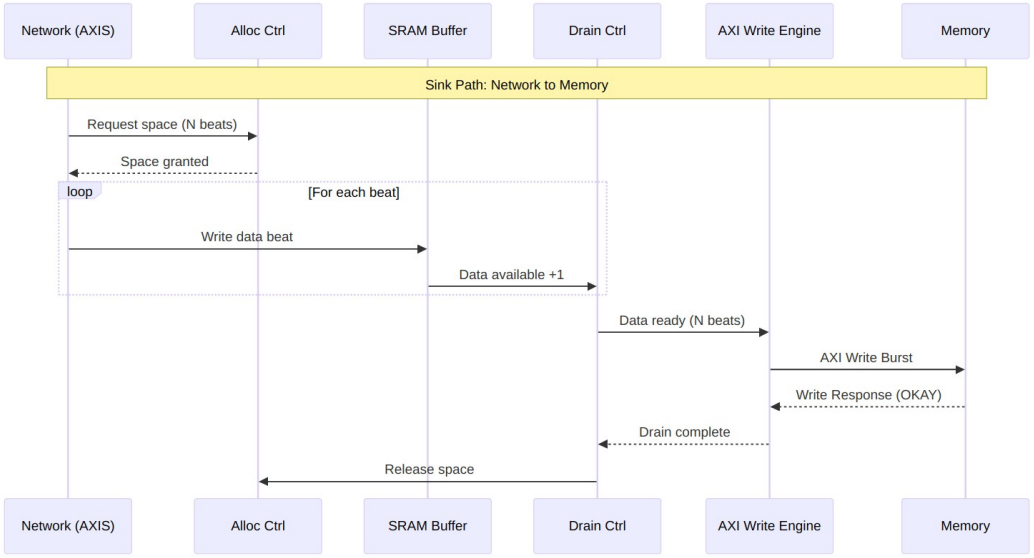


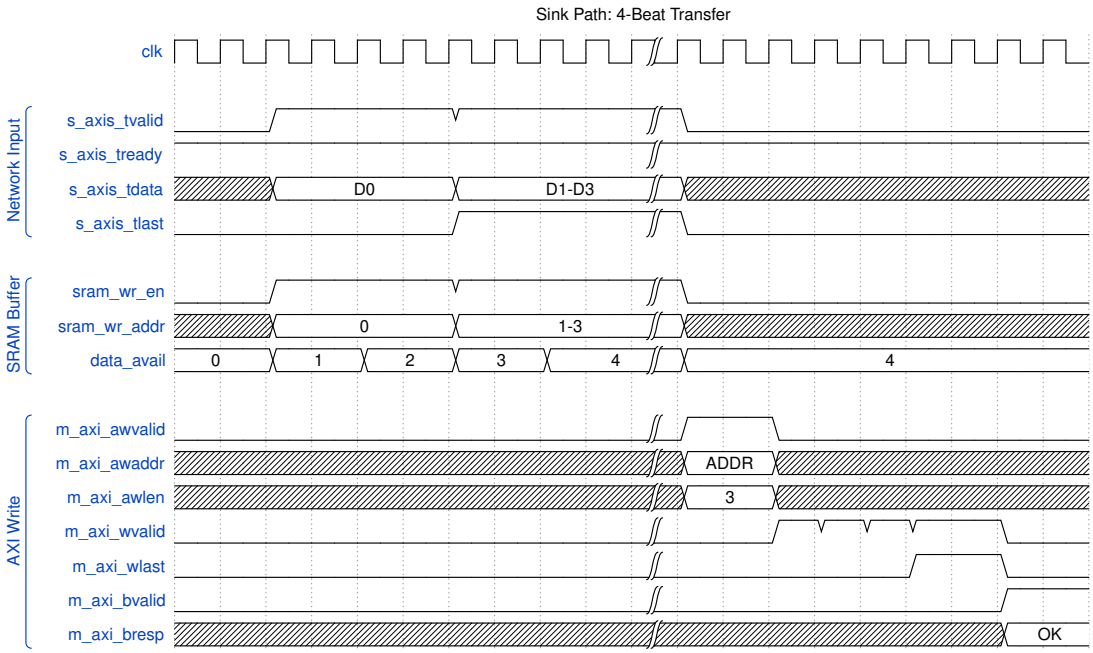
Diagram 5

9.2.1 Sink Path Timing

Sink Path Timing

Sink Path Timing

Source: sink_path_timing.json



Waveform 1

9.3 Source Path Data Flow

Source Data Flow

Source Data Flow

Source: 06_source_flow.mmd

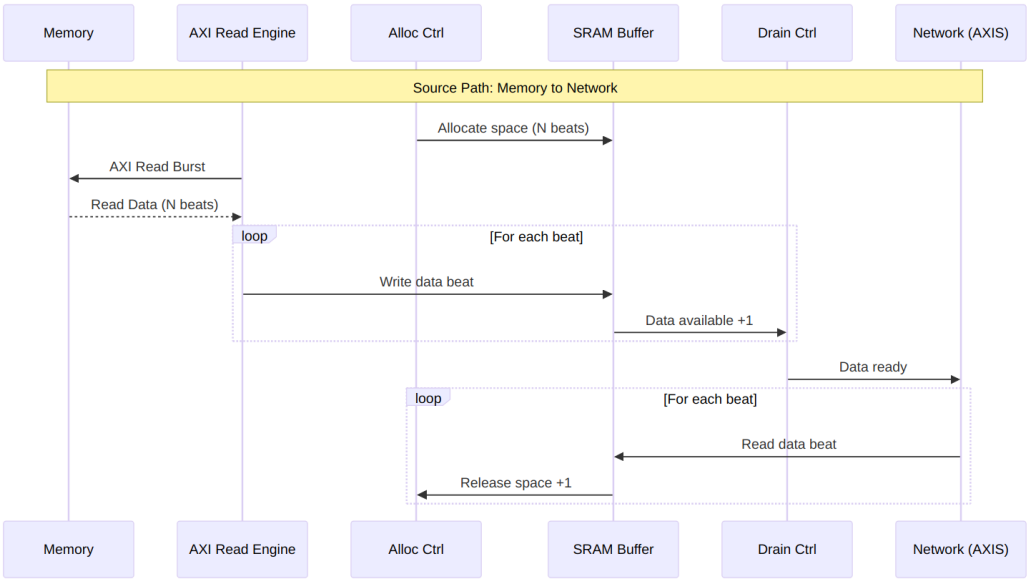


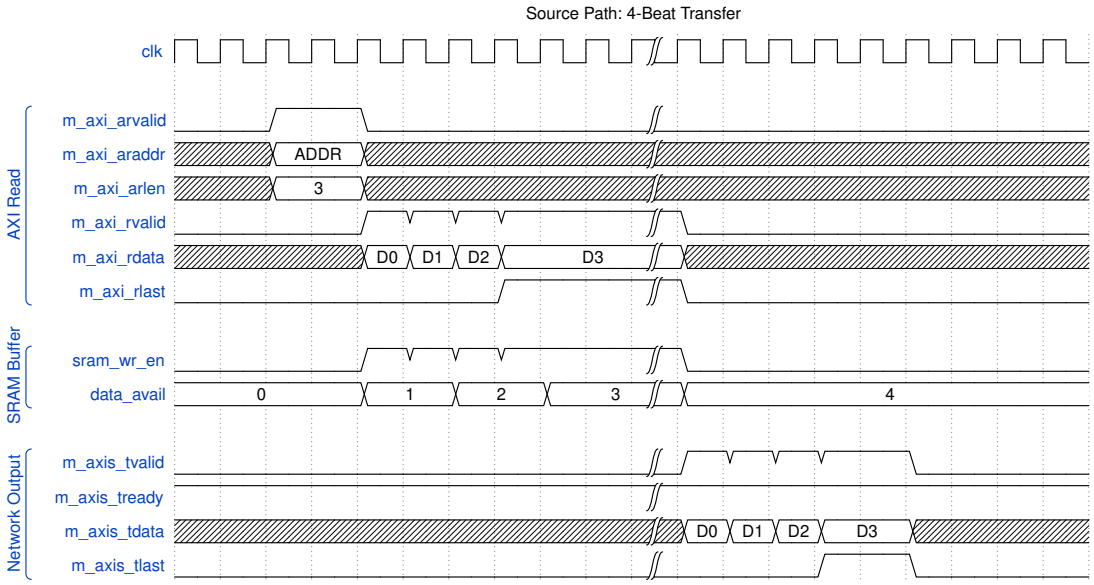
Diagram 6

9.3.1 Source Path Timing

Source Path Timing

Source Path Timing

Source: source_path_timing.json



Waveform 2

9.4 Concurrent Operation

Both data paths operate independently and can run simultaneously:

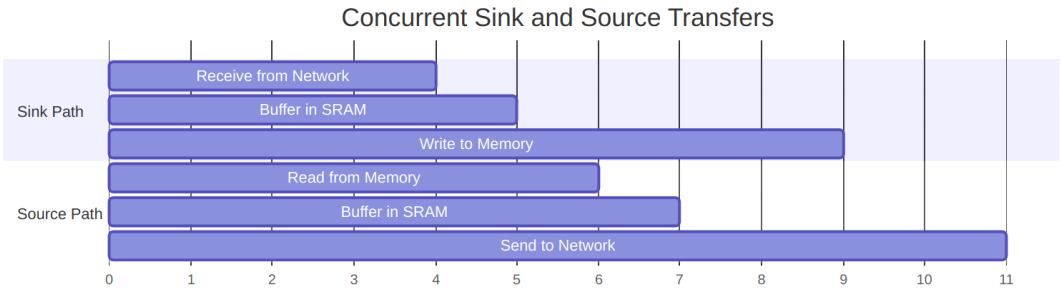


Diagram 7

9.4.1 Deadlock Prevention

RAPIDS prevents deadlock through independent resource allocation:

Resource Isolation

Resource	Sink Path	Source Path	Shared
SRAM Buffer	Dedicated	Dedicated	No
AXI Port	Write only	Read only	No
Scheduler	Per-channel	Per-channel	No

10 Channel Architecture

10.1 Multi-Channel Design

RAPIDS Beats supports 8 independent DMA channels, each with dedicated resources for concurrent operation.

10.2 Channel Resource Allocation

Channel Architecture

Channel Architecture

Source: 07_channel_arch.mmd

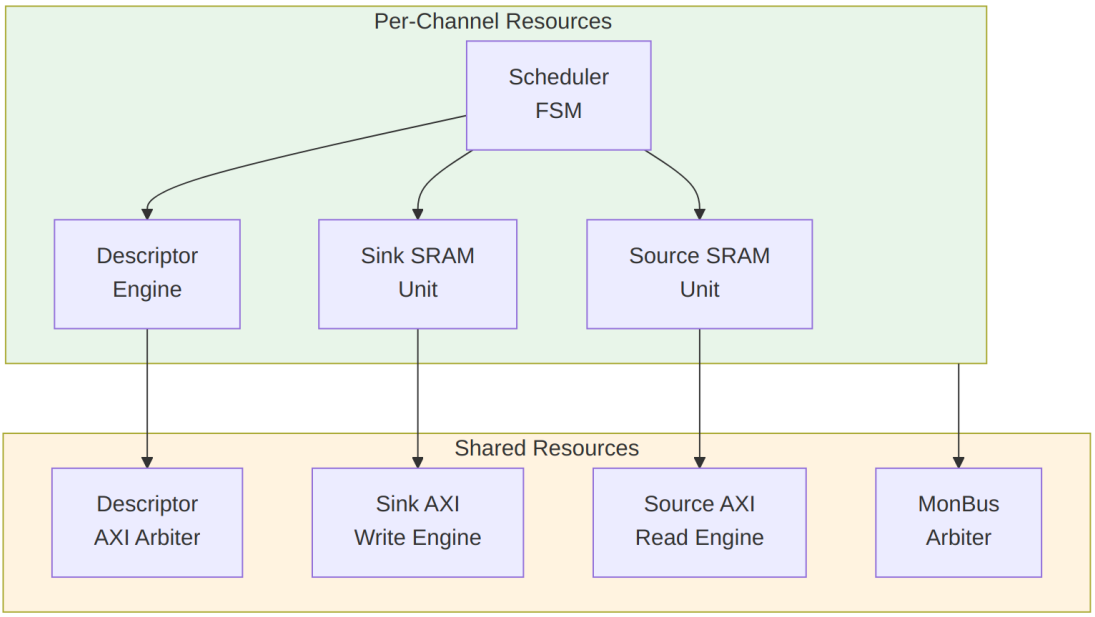


Diagram 8

10.3 Per-Channel Resources

Each channel has dedicated:

Per-Channel Resources

Resource	Description	Size/Capacity
Scheduler	Transfer coordination FSM	1 instance
Descriptor Engine	Descriptor fetch/parse	8-deep FIFO
Sink SRAM Unit	Sink buffer partition	SRAM_DEPTH/8 entries

Resource	Description	Size/Capacity
Source SRAM Unit	Source buffer partition	SRAM_DEPTH/8 entries
Configuration Registers	Per-channel config	~16 registers

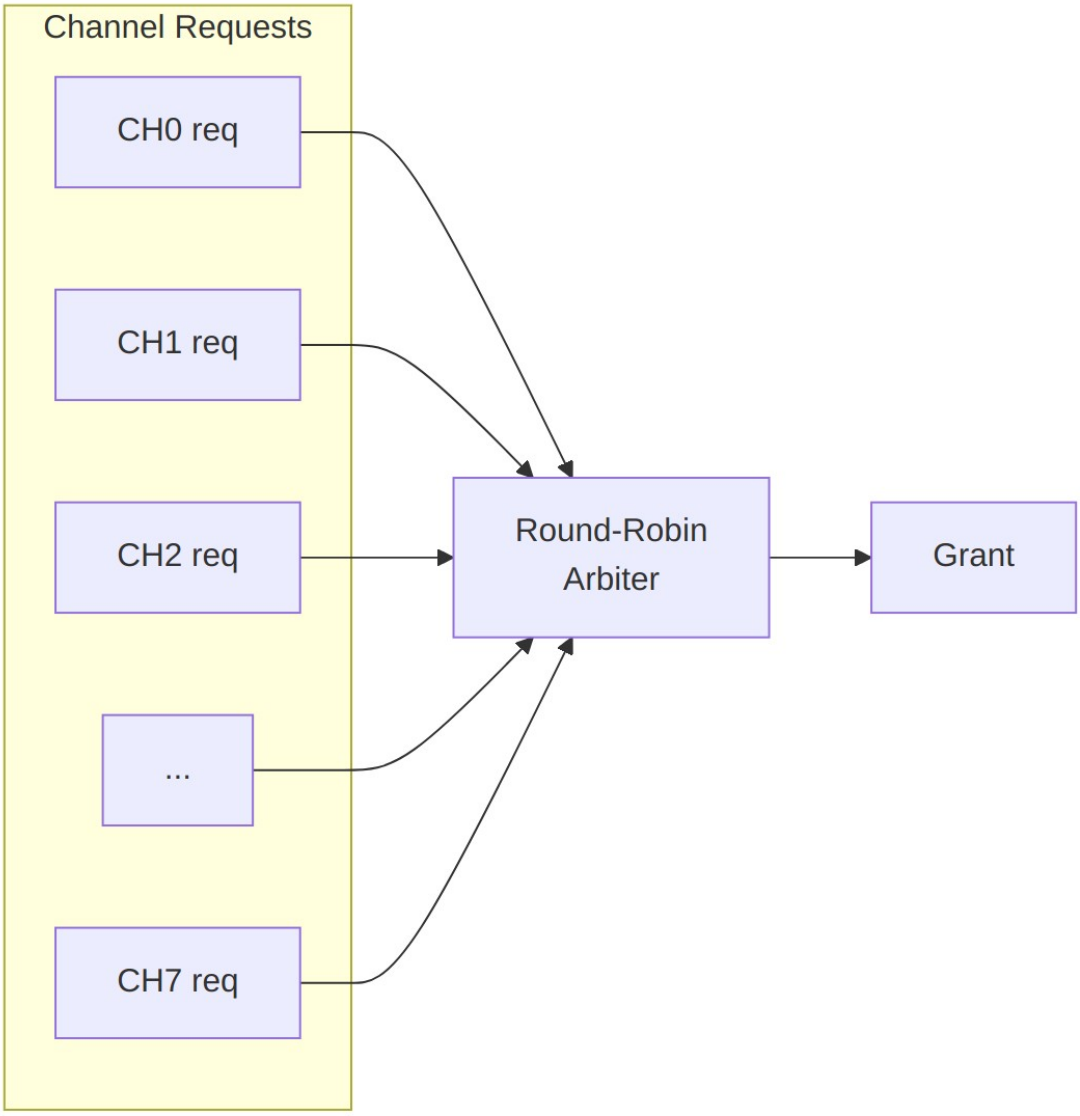
10.4 Shared Resources

Resources shared across all channels:

Shared Resources

Resource	Arbitration	Max Outstanding
Descriptor AXI	Round-robin	8 (1 per channel)
Sink AXI Write	Round-robin	8 configurable
Source AXI Read	Round-robin	8 configurable
MonBus Output	Priority	1 (serialized)

10.5 Channel Scheduling



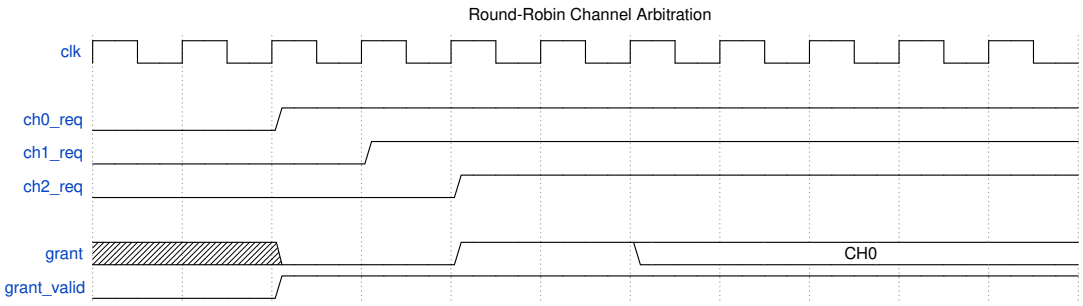
Arbitration Policy

Arbitration Timing:

Channel Arbitration

Channel Arbitration

Source: channel_arbitration.json



Waveform 3

10.6 Channel State Machine

Each channel scheduler follows this state machine:

Channel FSM

Channel FSM

Source: 08_channel_fsm.mmd

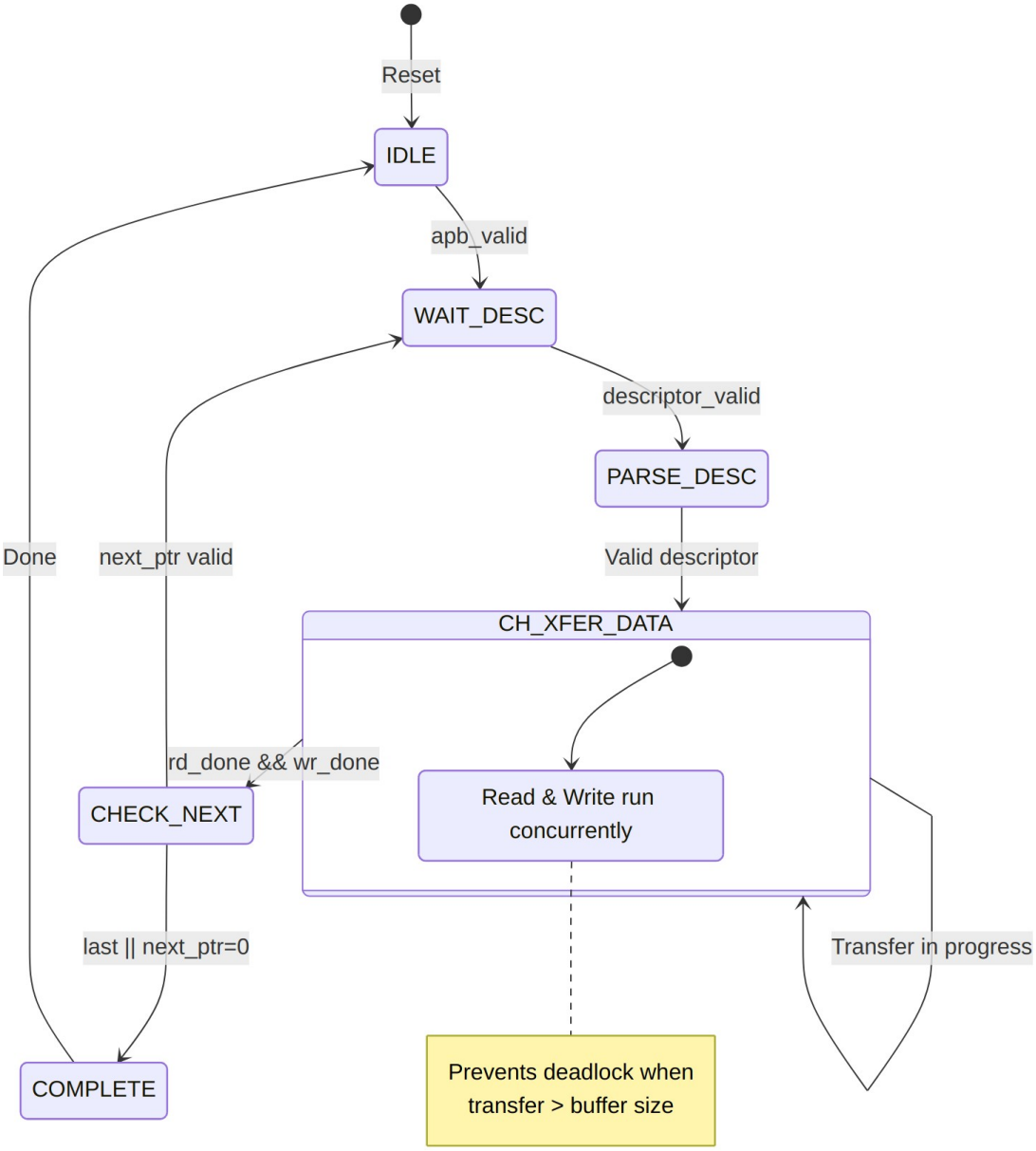


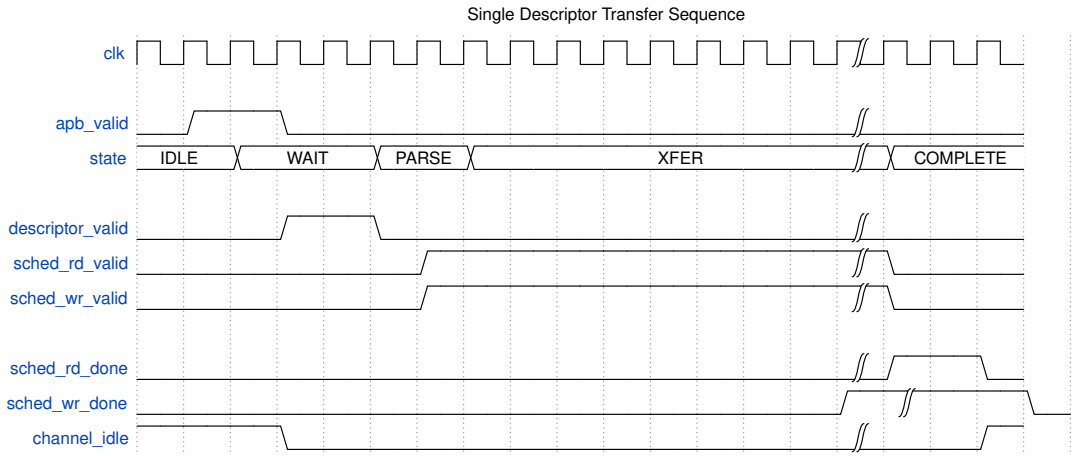
Diagram 10

10.6.1 Channel State Timing

Channel State Timing

Channel State Timing

Source: channel_state_timing.json



Waveform 4

10.7 Channel Isolation

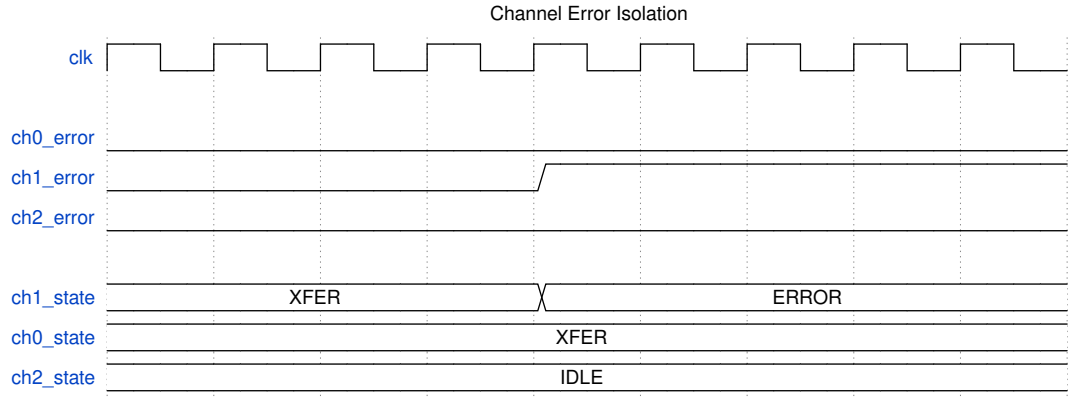
Channels are isolated to prevent interference:

Channel Isolation Features

Isolation Type	Implementation
State Isolation	Separate FSM per channel
Buffer Isolation	Partitioned SRAM per channel
Error Isolation	Per-channel error flags
Reset Isolation	Per-channel soft reset

10.7.1 Per-Channel Error Handling

Each channel maintains independent error state:



Waveform 5

Channel 1 error does not affect Channel 0 or Channel 2 operation.

11 Interface Summary

11.1 External Interface Overview

RAPIDS Beats exposes the following external interfaces:

Interface Overview

Source: 09_interface_overview.mmd

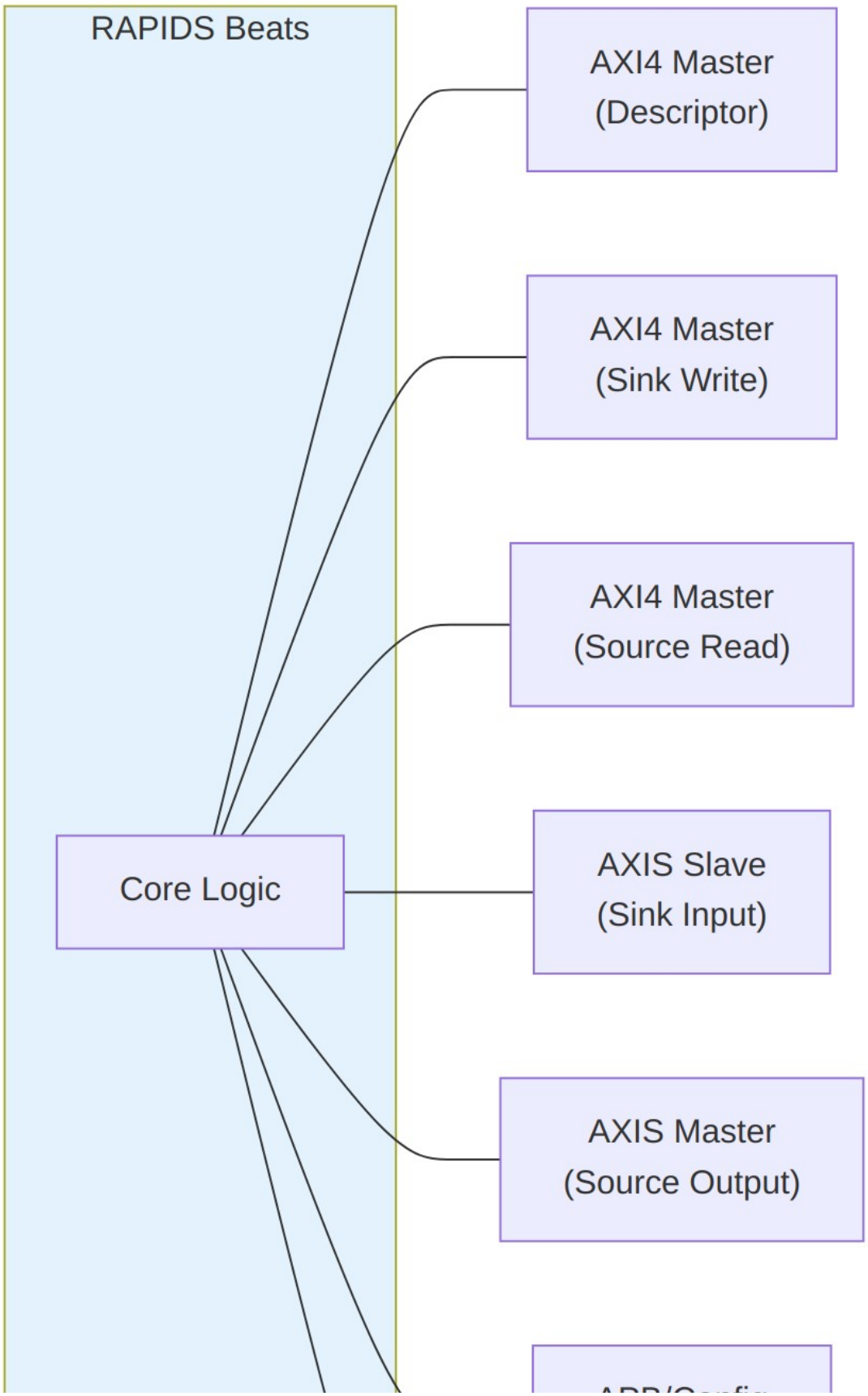

RAPIDS Beats
AXI4 Master
(Descriptor)
AXI4 Master
(Sink Write)
AXI4 Master
(Source Read)
Core Logic
AXIS Slave
(Sink Input)
AXIS Master
(Source Output)

Diagram 11

11.2 Interface Port Summary

External Interface Summary

Interface	Type	Direction	Data Width	Address Width
Descriptor AXI	AXI4 Master	Read	256-bit	64-bit
Sink AXI	AXI4 Master	Write	512-bit	64-bit
Source AXI	AXI4 Master	Read	512-bit	64-bit
Sink AXIS	AXIS Slave	Input	512-bit	-
Source AXIS	AXIS Master	Output	512-bit	-
APB Config	APB-like	Input	32-bit	12-bit
MonBus	Custom	Output	64-bit	-

11.3 Clock and Reset

All interfaces operate in the same clock domain:

Clock and Reset Signals

Signal	Type	Description
`aclk`	input	System clock (100-500 MHz typical)
`aresetn`	input	Active-low reset (async assert, sync deassert)

11.4 Interface Feature Matrix

Interface Feature Matrix

Feature	Desc AXI	Sink AXI	Src AXI	Sink AXIS	Src AXIS
Protocol	AXI4	AXI4	AXI4	AXIS	AXIS
Directio	Read	Write	Read	Slave	Master

Feature	Desc AXI	Sink AXI	Src AXI	Sink AXIS	Src AXIS
n					
Burst Support	INCR	INCR	INCR	N/A	N/A
Max Burst	1	256	256	N/A	N/A
Outstanding	8	8	8	N/A	N/A
ID Width	8-bit	8-bit	8-bit	Optional	Optional

11.5 Signal Count Summary

Signal Count Summary

Interface	Input Signals	Output Signals	Total
Descriptor AXI	~20	~30	~50
Sink AXI	~15	~40	~55
Source AXI	~25	~25	~50
Sink AXIS	~20	~5	~25
Source AXIS	~5	~20	~25
APB Config	~15	~5	~20
MonBus	~2	~5	~7
Total	**~102**	**~130**	**~232**

Note: Signal counts are approximate and vary with configuration parameters.

12 AXI4 Interface Specification

12.1 Overview

RAPIDS Beats uses three AXI4 master interfaces for memory access:

1. **Descriptor AXI** - Fetches 256-bit descriptors (read-only)
2. **Sink AXI** - Writes data to memory (write-only)
3. **Source AXI** - Reads data from memory (read-only)

12.2 Descriptor AXI Master

12.2.1 Purpose

Fetches descriptor structures from system memory for all 8 channels.

12.2.2 Configuration

Descriptor AXI Configuration

Parameter	Value	Description
Data Width	256-bit	Fixed descriptor size
Address Width	64-bit	Configurable
ID Width	8-bit	Configurable
Burst Length	1	Single-beat only
Burst Type	INCR	Fixed

12.2.3 Signal List

Descriptor AXI Signals

Signal	Width	Direction	Description
`m_axi_desc_aclk`	1	input	Clock
`m_axi_desc_aresetn`	1	input	Reset (active-low)
`m_axi_desc_arid`	8	output	Read ID (channel ID)
`m_axi_desc_araddr`	64	output	Read address
`m_axi_desc_arlen`	8	output	Burst length (always 0)
`m_axi_desc_arsize`	3	output	Beat size (5 = 32 bytes)
`m_axi_desc_arburst`	2	output	Burst type (INCR)
`m_axi_desc_arvalid`	1	output	Address valid
`m_axi_desc_arready`	1	input	Address ready
`m_axi_desc_rid`	8	input	Response ID

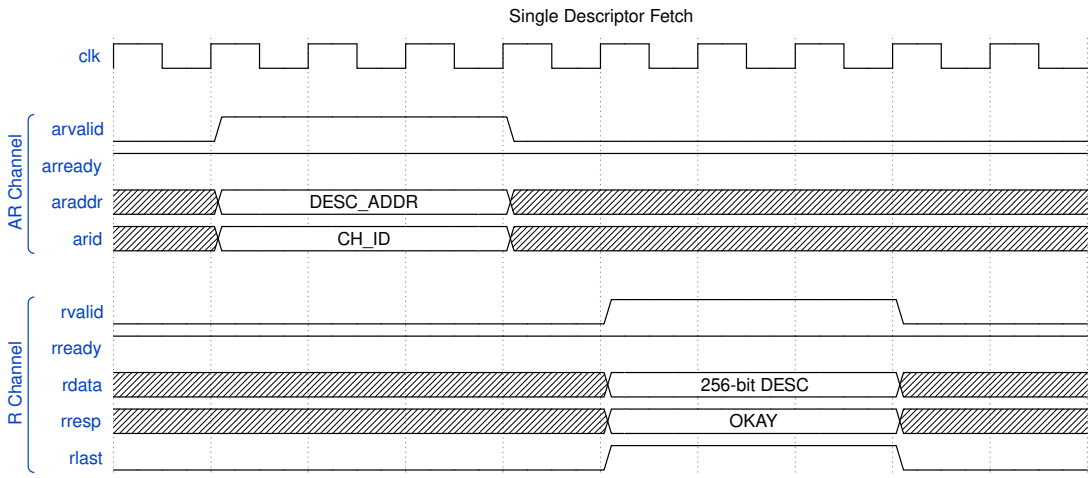
Signal	Width	Direction	Description
`m_axi_desc_rdata`	256	input	Read data
`m_axi_desc_resp`	2	input	Read response
`m_axi_desc_rlast`	1	input	Last beat
`m_axi_desc_rvalid`	1	input	Data valid
`m_axi_desc_rready`	1	output	Data ready

12.2.4 Timing Diagram

Descriptor Fetch Timing

Descriptor Fetch Timing

Source: desc_axi_timing.json



Waveform 6

12.3 Sink AXI Master (Write)

12.3.1 Purpose

Writes data from SRAM buffer to system memory.

12.3.2 Configuration

Sink AXI Configuration

Parameter	Default	Range	Description
Data Width	512-bit	Configurable	Match SRAM width
Address Width	64-bit	Configurable	System address space
ID Width	8-bit	Configurable	Transaction tracking
Max Burst	256	1-256	AXI4 maximum
Outstanding	8	1-16	Concurrent AW requests

12.3.3 Signal List

Sink AXI Signals

Signal	Width	Direction	Description
`m_axi_sink_awid`	8	output	Write ID
`m_axi_sink_awaddr`	64	output	Write address
`m_axi_sink_awlen`	8	output	Burst length - 1
`m_axi_sink_awsize`	3	output	Beat size (6 = 64 bytes)
`m_axi_sink_awburst`	2	output	Burst type (INCR)
`m_axi_sink_awvalid`	1	output	Address valid
`m_axi_sink_awready`	1	input	Address ready
`m_axi_sink_wdata`	512	output	Write data
`m_axi_sink_wstrb`	64	output	Write strobes
`m_axi_sink_wlast`	1	output	Last beat
`m_axi_sink_wvalid`	1	output	Data valid

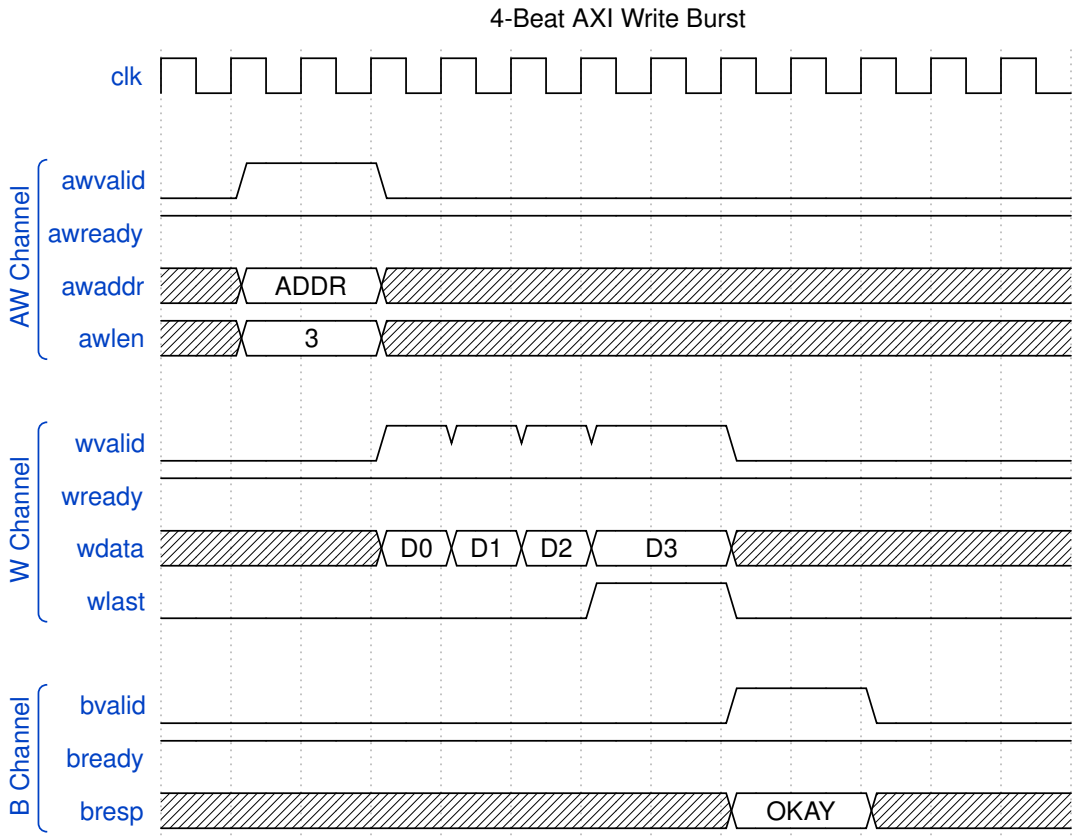
Signal	Width	Direction	Description
`m_axi_sink_wready`	1	input	Data ready
`m_axi_sink_bid`	8	input	Response ID
`m_axi_sink_bresp`	2	input	Write response
`m_axi_sink_bvalid`	1	input	Response valid
`m_axi_sink_bready`	1	output	Response ready

12.3.4 Timing Diagram

Sink AXI Write Timing

Sink AXI Write Timing

Source: sink_axi_timing.json



Waveform 7

12.4 Source AXI Master (Read)

12.4.1 Purpose

Reads data from system memory to SRAM buffer.

12.4.2 Configuration

Source AXI Configuration

Parameter	Default	Range	Description
Data Width	512-bit	Configurable	Match SRAM width
Address Width	64-bit	Configurable	System address space
ID Width	8-bit	Configurable	Transaction tracking
Max Burst	256	1-256	AXI4 maximum
Outstanding	8	1-16	Concurrent AR requests

12.4.3 Signal List

Source AXI Signals

Signal	Width	Direction	Description
`m_axi_src_arid`	8	output	Read ID
`m_axi_src_araddr`	64	output	Read address
`m_axi_src_arlen`	8	output	Burst length - 1
`m_axi_src_arsize`	3	output	Beat size (6 = 64 bytes)
`m_axi_src_arburst`	2	output	Burst type (INCR)
`m_axi_src_arvalid`	1	output	Address valid
`m_axi_src_arready`	1	input	Address ready
`m_axi_src_rid`	8	input	Response ID
`m_axi_src_rda`	512	input	Read data

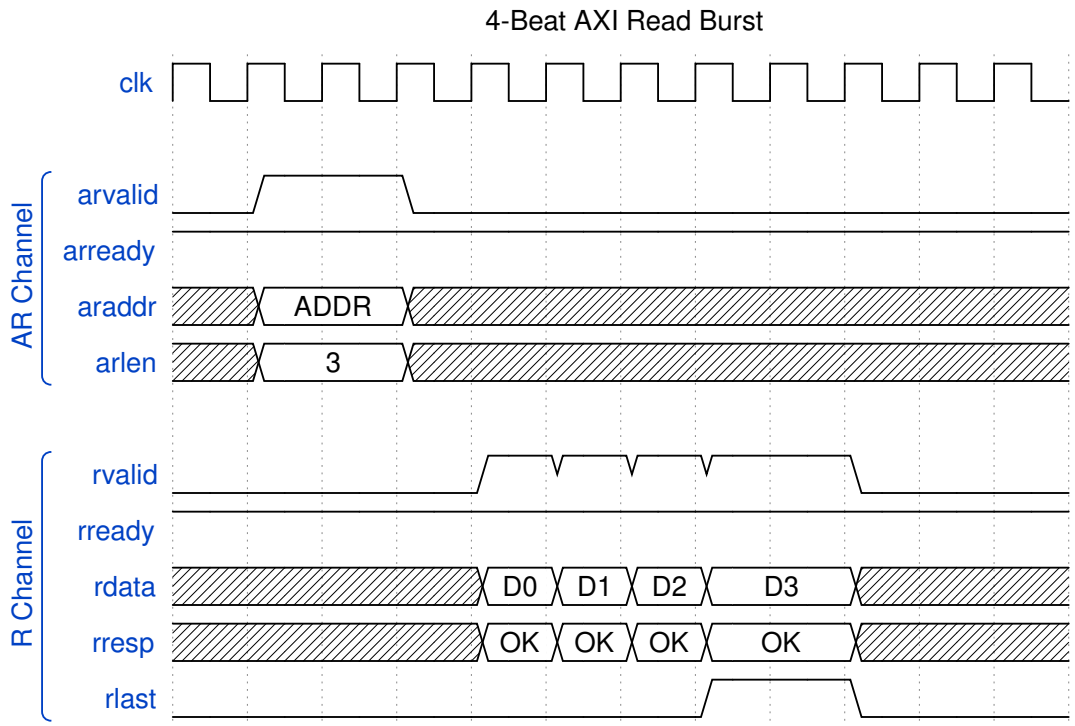
Signal	Width	Direction	Description
ta			
m_axi_src_rresp	2	input	Read response
m_axi_src_rlast	1	input	Last beat
m_axi_src_rvalid	1	input	Data valid
m_axi_src_rready	1	output	Data ready

12.4.4 Timing Diagram

Source AXI Read Timing

Source AXI Read Timing

Source: source_axi_timing.json



Waveform 8

12.5 AXI Protocol Constraints

12.5.1 Address Alignment

Address Alignment Requirements

Transfer Type	Alignment Requirement
Descriptor	32-byte aligned
Sink Data	64-byte aligned
Source Data	64-byte aligned

12.5.2 4KB Boundary Handling

AXI4 requires burst transactions not to cross 4KB boundaries. RAPIDS automatically splits bursts at 4KB boundaries:

```
Example: 8-beat burst starting at 0x0FE0 (64 bytes/beat)
  Beat 0-1: 0x0FE0 - 0x0FFF (to 4KB boundary)
  Beat 2-7: 0x1000 - 0x117F (after boundary)

Split into two bursts:
  Burst 1: ARLEN=1, ARADDR=0x0FE0
  Burst 2: ARLEN=5, ARADDR=0x1000
```

12.5.3 Response Handling

AXI Response Handling

Response	Code	RAPIDS Behavior
OKAY	2'b00	Normal completion
EXOKAY	2'b01	Treated as OKAY
SLVERR	2'b10	Error, transfer aborted
DECERR	2'b11	Error, transfer aborted

13 AXIS Interface Specification

13.1 Overview

RAPIDS Beats uses AXI-Stream interfaces for network data transfer:

1. **Sink AXIS Slave** - Receives data from network (ingress)
2. **Source AXIS Master** - Sends data to network (egress)

13.2 Sink AXIS Slave

13.2.1 Purpose

Receives streaming data from external network interface and writes to SRAM buffer.

13.2.2 Configuration

Sink AXIS Configuration

Parameter	Default	Range	Description
Data Width	512-bit	Configurable	TDATA width
ID Width	8-bit	0-8	TID width (optional)
DEST Width	8-bit	0-8	TDEST width (optional)
USER Width	1-bit	0-16	TUSER width (optional)

13.2.3 Signal List

Sink AXIS Signals

Signal	Width	Direction	Description
`s_axis_sink_tdata`	512	input	Data payload
`s_axis_sink_tkeep`	64	input	Byte valid mask
`s_axis_sink_tstrb`	64	input	Byte position (optional)
`s_axis_sink_tlast`	1	input	End of packet
`s_axis_sink_tid`	8	input	Transaction ID (channel select)
`s_axis_sink_tdest`	8	input	Destination (optional)
`s_axis_sink_tuser`	1	input	User sideband (optional)
`s_axis_sink_tvalid`	1	input	Data valid

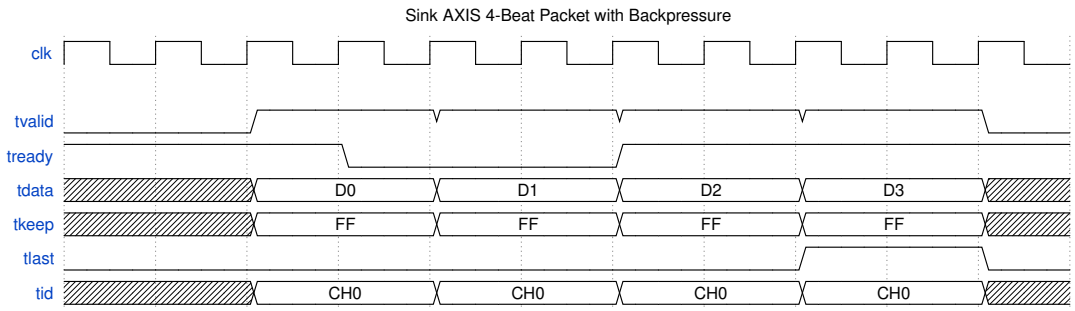
Signal	Width	Direction	Description
`s_axis_sink_tready`	1	output	Ready to accept

13.2.4 Timing Diagram

Sink AXIS Timing

Sink AXIS Timing

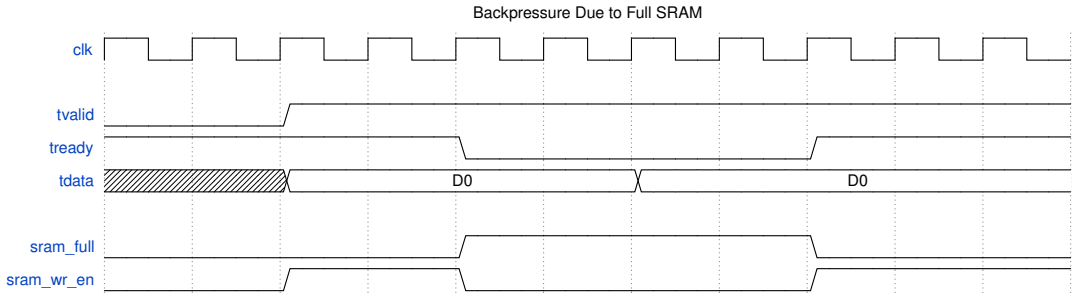
Source: sink_axis_timing.json



Waveform 9

13.2.5 Backpressure Behavior

When SRAM buffer is full, `tready` deasserts:



Waveform 10

13.3 Source AXIS Master

13.3.1 Purpose

Sends streaming data from SRAM buffer to external network interface.

13.3.2 Configuration

Source AXIS Configuration

Parameter	Default	Range	Description
Data Width	512-bit	Configurable	TDATA width
ID Width	8-bit	0-8	TID width (optional)
DEST Width	8-bit	0-8	TDEST width (optional)
USER Width	1-bit	0-16	TUSER width (optional)

13.3.3 Signal List

Source AXIS Signals

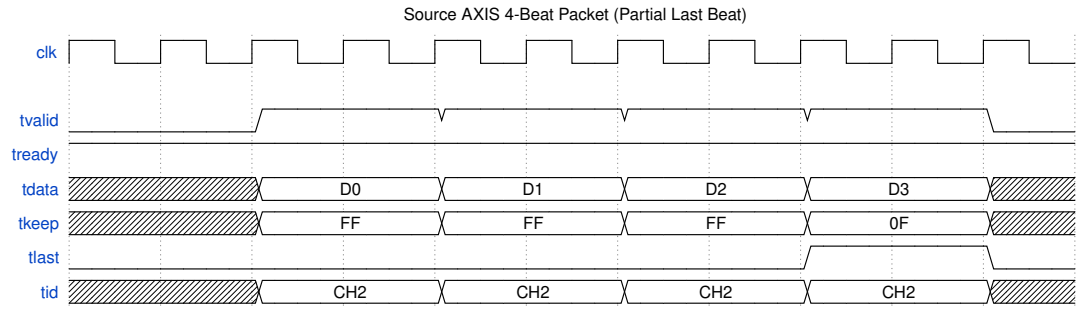
Signal	Width	Direction	Description
`m_axis_src_tdata`	512	output	Data payload
`m_axis_src_tkeep`	64	output	Byte valid mask
`m_axis_src_tstrb`	64	output	Byte position (optional)
`m_axis_src_tlast`	1	output	End of packet
`m_axis_src_tid`	8	output	Transaction ID (channel)
`m_axis_src_tdest`	8	output	Destination (optional)
`m_axis_src_tuser`	1	output	User sideband (optional)
`m_axis_src_tvalid`	1	output	Data valid
`m_axis_src_tready`	1	input	Downstream ready

13.3.4 Timing Diagram

Source AXIS Timing

Source AXIS Timing

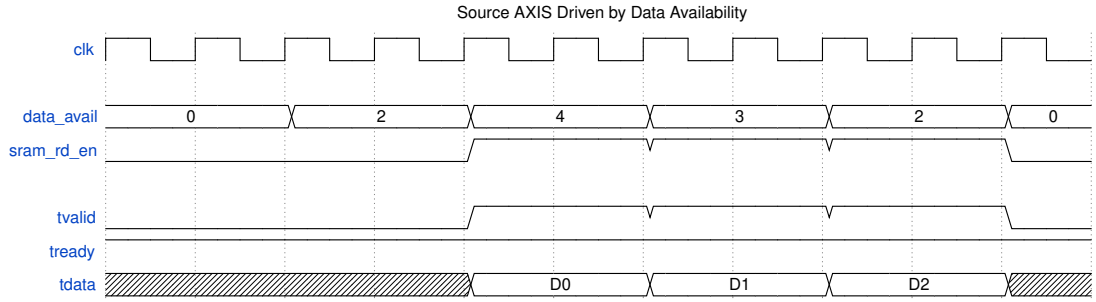
Source: source_axis_timing.json



Waveform 11

13.3.5 Data Availability

Source AXIS only asserts `tvalid` when data is available in SRAM:



Waveform 12

13.4 AXIS Protocol Notes

13.4.1 Handshake Rules

Standard AXI-Stream handshake applies:

1. `tvalid` asserts when data is available
2. `tready` asserts when receiver can accept
3. Transfer occurs on clock edge when both are high
4. `tvalid` SHALL NOT depend on `tready`
5. `tready` MAY depend on `tvalid`

13.4.2 TLAST Semantics

TLAST Usage

Scenario	TLAST Behavior
Descriptor transfer complete	Assert on last beat
Packet boundary	Assert on last beat of packet
Continuous stream	Not used (always 0)

13.4.3 TKEEP Usage

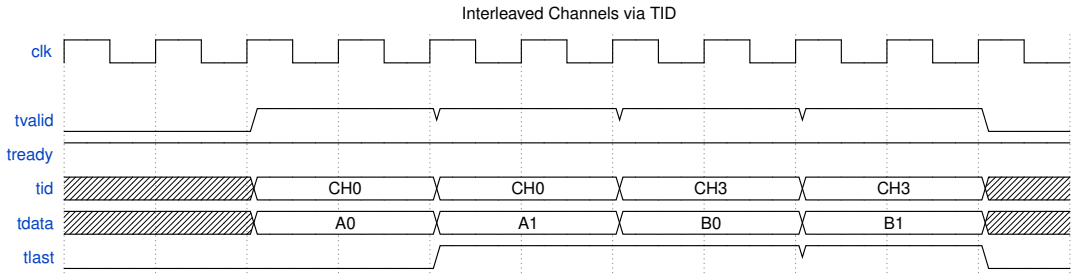
TKEEP indicates valid bytes within each beat:

TKEEP Examples

TKEEP Value	Meaning
64'hFFFF_FFFF_FFFF_FFFF	All 64 bytes valid
64'h0000_FFFF_FFFF_FFFF	Upper 16 bytes invalid
64'h0000_0000_0000_000F	Only lower 4 bytes valid

13.4.4 Channel Multiplexing via TID

Multiple channels can share a single AXIS interface using TID:



Waveform 13

14 APB Configuration Interface

14.1 Overview

RAPIDS Beats uses an APB-like interface for configuration and descriptor kick-off. This interface allows software to:

1. Configure channel parameters
2. Initiate descriptor processing (kick-off)

3. Read status and error information

14.2 Configuration Interface

14.2.1 Signal List

APB Configuration Signals (NC = NUM_CHANNELS)

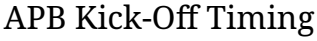
Signal	Width	Direction	Description
`apb_valid`	NC	input	Per-channel kick-off valid
`apb_ready`	NC	output	Per-channel ready
`apb_addr`	64	input	Descriptor address
`cfg_channel_enable`	NC	input	Per-channel enable
`cfg_addr0_base`	64	input	Address range 0 base
`cfg_addr0_limit`	64	input	Address range 0 limit
`cfg_addr1_base`	64	input	Address range 1 base
`cfg_addr1_limit`	64	input	Address range 1 limit

14.2.2 Descriptor Kick-Off

To start descriptor processing on a channel:

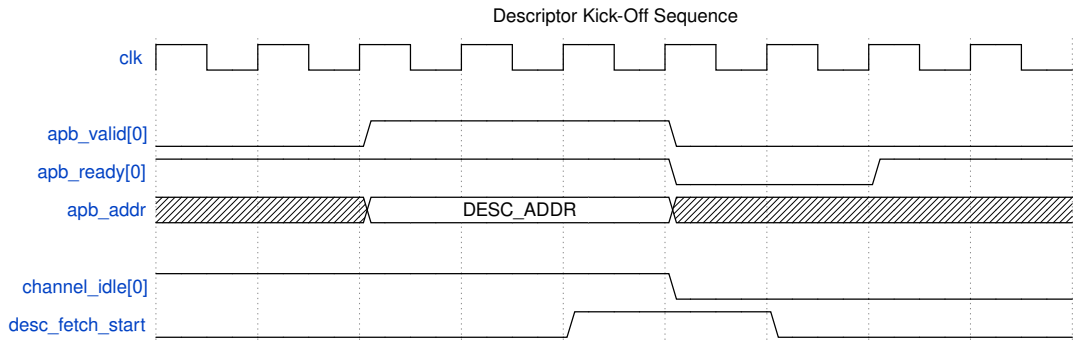
1. Write descriptor to memory
2. Assert `apb_valid[ch]` with descriptor address on `apb_addr`
3. Wait for `apb_ready[ch]` assertion
4. Deassert `apb_valid[ch]`

14.2.3 Timing Diagram



APB Kick-Off Timing

Source: apb_kickoff_timing.json



Waveform 14

14.3 Status Interface

14.3.1 Per-Channel Status Signals

Status Signals

Signal	Width	Direction	Description
`channel_idle`	NC	output	Channel idle status
`channel_error`	NC	output	Channel error flag
`scheduler_state`	4*NC	output	FSM state per channel

14.3.2 Channel State Encoding

Scheduler State Encoding

State	Value	Description
IDLE	4'h0	Waiting for kick-off
WAIT_DESC	4'h1	Waiting for descriptor
PARSE_DESC	4'h2	Parsing descriptor
CH_XFER_DATA	4'h3	Transfer in progress
CHECK_NEXT	4'h4	Checking next descriptor
COMPLETE	4'h5	Transfer complete
ERROR	4'hF	Error state

14.4 Configuration Registers

14.4.1 Address Range Validation

RAPIDS validates descriptor addresses against configurable ranges:

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

14.4.2 Configuration Parameters

Configuration Register Map

Register	Offset	Width	Description
CTRL	0x00	32	Global control
STATUS	0x04	32	Global status
CH_ENABLE	0x08	8	Channel enable bitmap
CH_STATUS	0x0C	8	Channel busy bitmap
ADDR0_BASE	0x10	64	Address range 0 base
ADDR0_LIMIT	0x18	64	Address range 0 limit
ADDR1_BASE	0x20	64	Address range 1 base
ADDR1_LIMIT	0x28	64	Address range 1 limit
IRQ_ENABLE	0x30	8	IRQ enable per channel
IRQ_STATUS	0x34	8	IRQ status per channel

14.4.3 Control Register (CTRL)

CTRL Register Bits

Bit	Name	Description
0	ENABLE	Global enable
1	SOFT_RESET	Soft reset (self-clearing)

Bit	Name	Description
7:2	Reserved	-

14.4.4 Status Register (STATUS)

STATUS Register Bits

Bit	Name	Description
0	BUSY	Any channel busy
1	ERROR	Any channel error
7:2	Reserved	-

14.5 Programming Sequence

14.5.1 Initialization

```
// 1. Assert soft reset
CTRL = 0x02;
while (CTRL & 0x02);  // Wait for reset complete

// 2. Configure address ranges
ADDR0_BASE = 0x80000000;
ADDR0_LIMIT = 0x90000000;

// 3. Enable desired channels
CH_ENABLE = 0xFF;  // Enable all 8 channels

// 4. Enable IRQs if needed
IRQ_ENABLE = 0xFF;

// 5. Global enable
CTRL = 0x01;
```

14.5.2 Descriptor Kick-Off

```
// Prepare descriptor in memory
desc->src_addr = src;
desc->dst_addr = dst;
desc->length = beats;
desc->last = 1;
desc->valid = 1;

// Kick off channel 0
apb_kickoff(0, desc_addr);

// Wait for completion
while (!(channel_idle & 0x01));
```

```
// Check for errors
if (channel_error & 0x01) {
    handle_error(0);
}
```

15 Monitor Bus Interface

15.1 Overview

RAPIDS Beats outputs a 64-bit Monitor Bus (MonBus) for real-time event reporting. The MonBus provides visibility into:

- State machine transitions
- Descriptor processing events
- Error conditions
- Performance metrics

15.2 MonBus Packet Format

15.2.1 64-bit Packet Structure

```
Bit Field           Width   Description
──────────────────────────────────────────────────────
[63:60]             4-bit   packet_type (error, completion, etc.)
[59:57]             3-bit   protocol (CORE for RAPIDS)
[56:53]             4-bit   event_code (specific event)
[52:47]             6-bit   channel_id
[46:43]             4-bit   unit_id (subsystem)
[42:35]             8-bit   agent_id (module)
[34:0]              35-bit  event_data (event-specific)
```

15.2.2 Signal List

MonBus Signals

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

15.3 Packet Types

MonBus Packet Types

Type	Code	Description
ERROR	4'h0	Error event
COMPLETION	4'h1	Transfer/operation complete
THRESHOLD	4'h2	Threshold crossed
TIMEOUT	4'h3	Timeout event
PERF	4'h4	Performance metric
DEBUG	4'hF	Debug/trace event

15.4 Agent IDs

RAPIDS modules use the following Agent IDs:

RAPIDS Agent IDs

Agent ID	Module	Description
0x10-0x17	Descriptor Engine	Channels 0-7
0x30-0x37	Scheduler	Channels 0-7
0x40-0x47	Sink Data Path	Channels 0-7
0x50-0x57	Source Data Path	Channels 0-7

15.5 Event Codes

15.5.1 Scheduler Events (Agent 0x30-0x37)

Scheduler Event Codes

Event Code	Type	Description
0x0	COMPLETION	Descriptor complete
0x1	COMPLETION	Chain complete
0x2	ERROR	Timeout
0x3	ERROR	AXI error
0x4	DEBUG	State transition

15.5.2 Descriptor Engine Events (Agent 0x10-0x17)

Descriptor Engine Event Codes

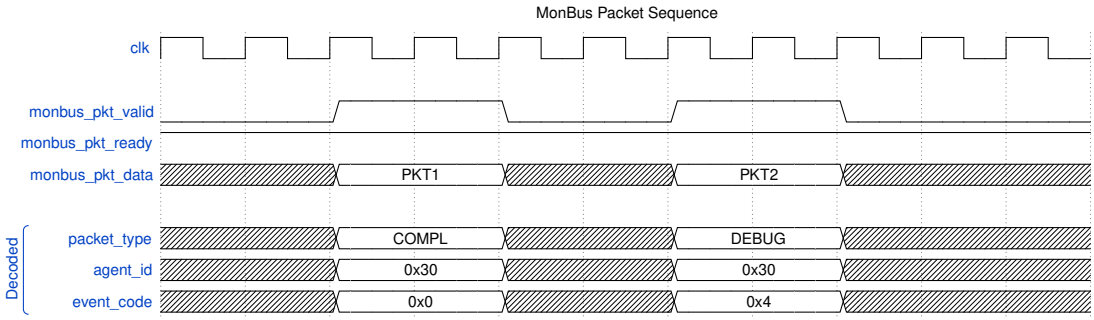
Event Code	Type	Description
0x0	COMPLETION	Descriptor fetched
0x1	ERROR	Address range error
0x2	ERROR	AXI read error
0x3	DEBUG	Fetch started

15.6 Timing Diagram

MonBus Timing

MonBus Timing

Source: monbus_timing.json



Waveform 15

15.7 Event Data Encoding

15.7.1 Completion Event Data

```
[34:32] - Reserved
[31:0]  - Transfer length (beats completed)
```

15.7.2 Error Event Data

```
[34:32] - Error type
[31:16] - Reserved
[15:0]  - Error details
```

15.7.3 State Transition Event Data

```
[34:32] - Reserved
[31:28] - Previous state
[27:24] - New state
[23:0]  - Timestamp (lower bits)
```

15.8 MonBus Arbitration

Multiple RAPIDS modules generate MonBus packets. Internal arbitration ensures ordered delivery:

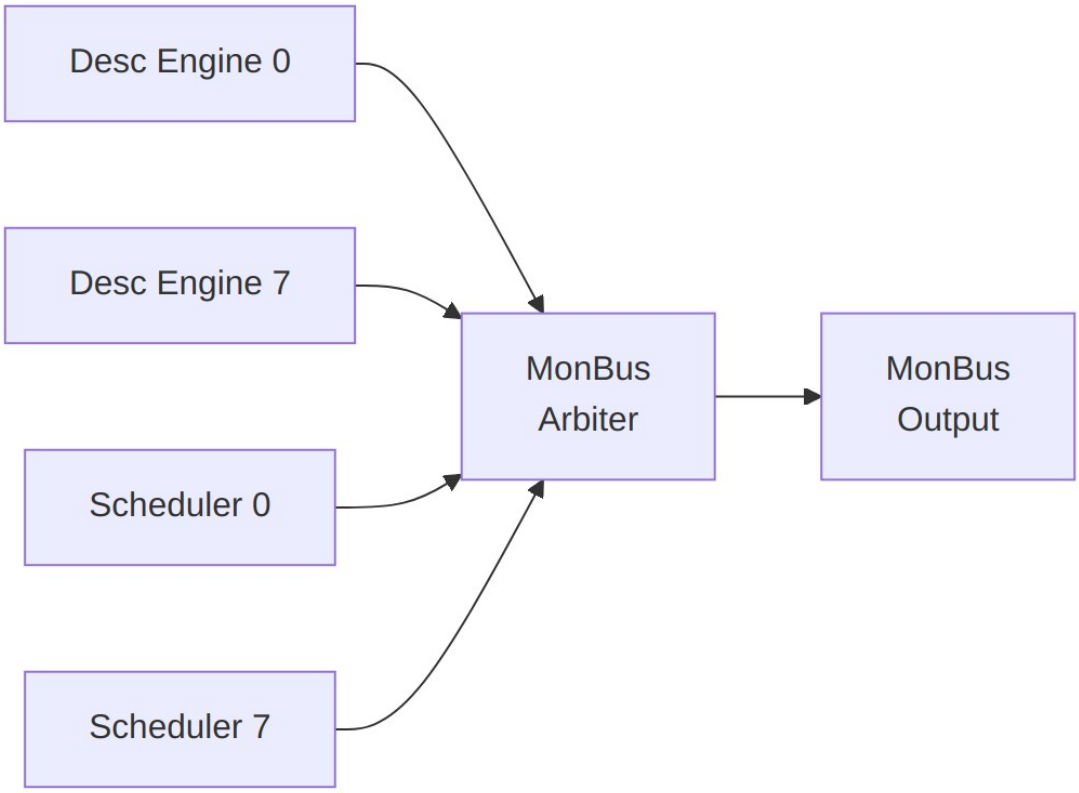


Diagram 12

15.8.1 Arbitration Policy

- Round-robin across modules
- No packet loss (backpressure if output blocked)
- Priority boosting for ERROR packets

15.9 Integration Notes

15.9.1 Downstream Connection

MonBus output typically connects to:

1. **MonBus Aggregator** - Combines multiple sources
2. **AXIL Converter** - Maps to AXI-Lite for CPU access
3. **Debug FIFO** - Buffered access for debug tools

15.9.2 Packet Rate

MonBus Packet Rates

Scenario	Approximate Rate
Idle	0 packets/cycle
Single channel active	~1 packet/100 cycles
All channels active	~1 packet/20 cycles
Error storm	Up to 1 packet/cycle

16 Use Case: Network to Memory (Sink Path)

16.1 Overview

The Sink Path transfers data from a network interface (AXI-Stream slave) to system memory (AXI4 write master). This is the primary receive data path for network applications.

16.2 Operation Flow

Sink Flow

Sink Flow

Source: 09_sink_flow.mmd

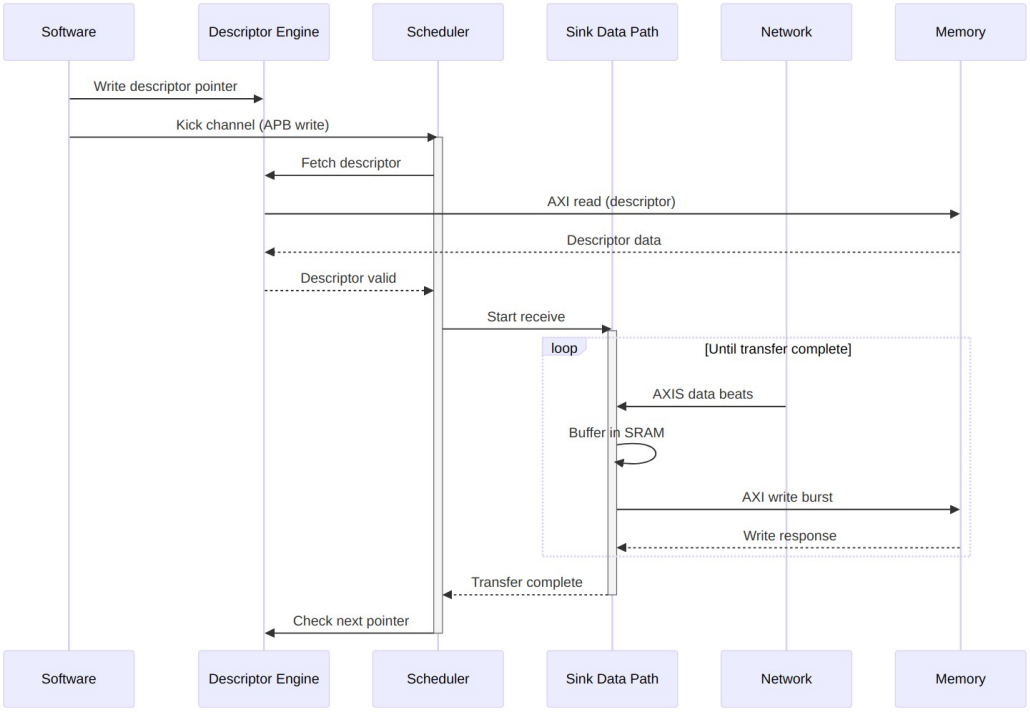


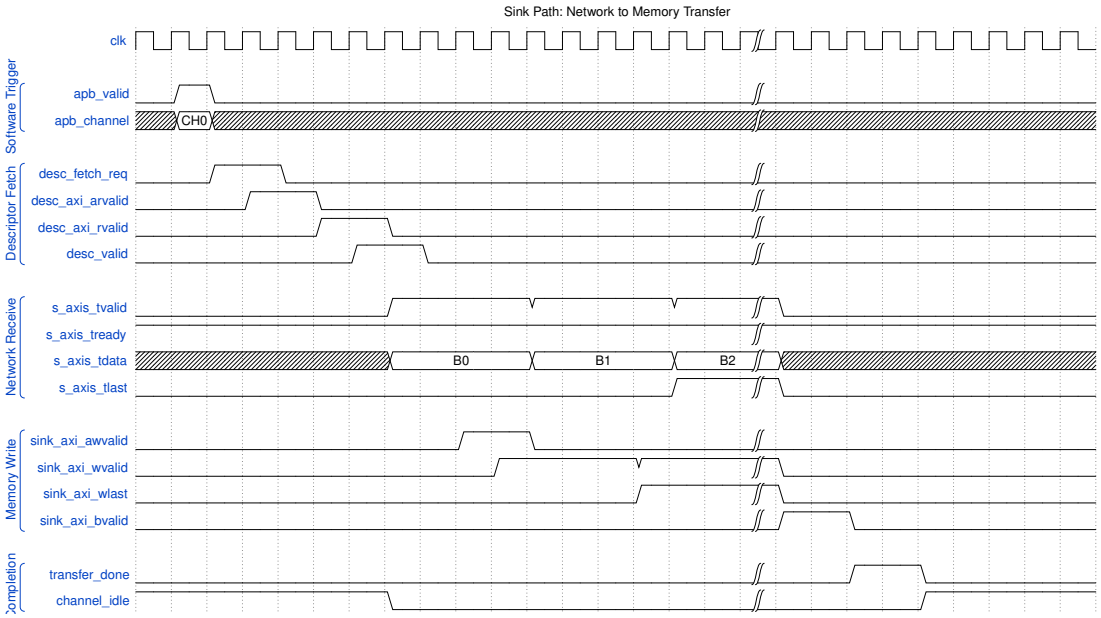
Diagram 13

16.3 Timing Diagram

Sink Transfer Timing

Sink Transfer Timing

Source: sink_transfer.json



Waveform 16

16.4 Buffer Management

16.4.1 SRAM Buffering Strategy

The sink path uses internal SRAM to decouple network timing from memory timing:

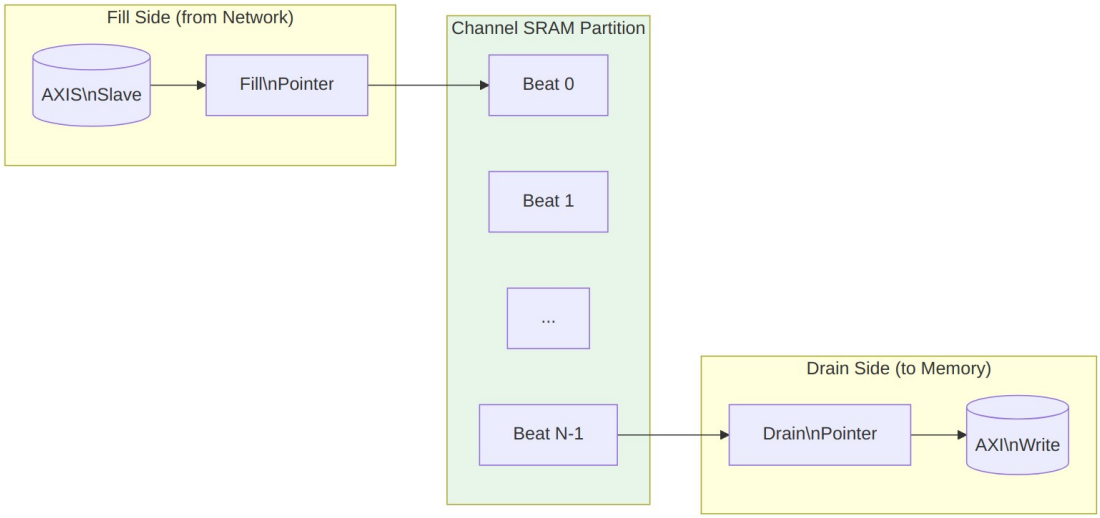
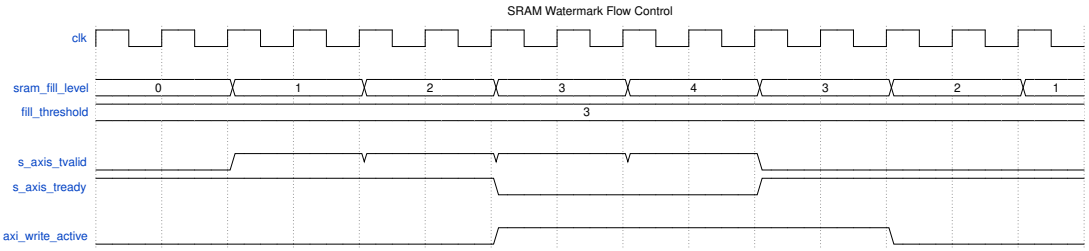


Diagram 14

16.4.2 Flow Control

Sink Path Flow Control

Condition	Behavior
SRAM full	Backpressure network (TREADY=0)
SRAM empty	Pause AXI writes
SRAM threshold	Optional early write start



Watermark Timing

16.5 Performance Considerations

16.5.1 Throughput

Sink Path Throughput Factors

Factor	Impact
Network bandwidth	Upper bound on receive rate
Memory bandwidth	Must match or exceed network
SRAM depth	Burst absorption capability
AXI outstanding	Overlapped write latency hiding

16.5.2 Latency

Sink Path Latency Breakdown

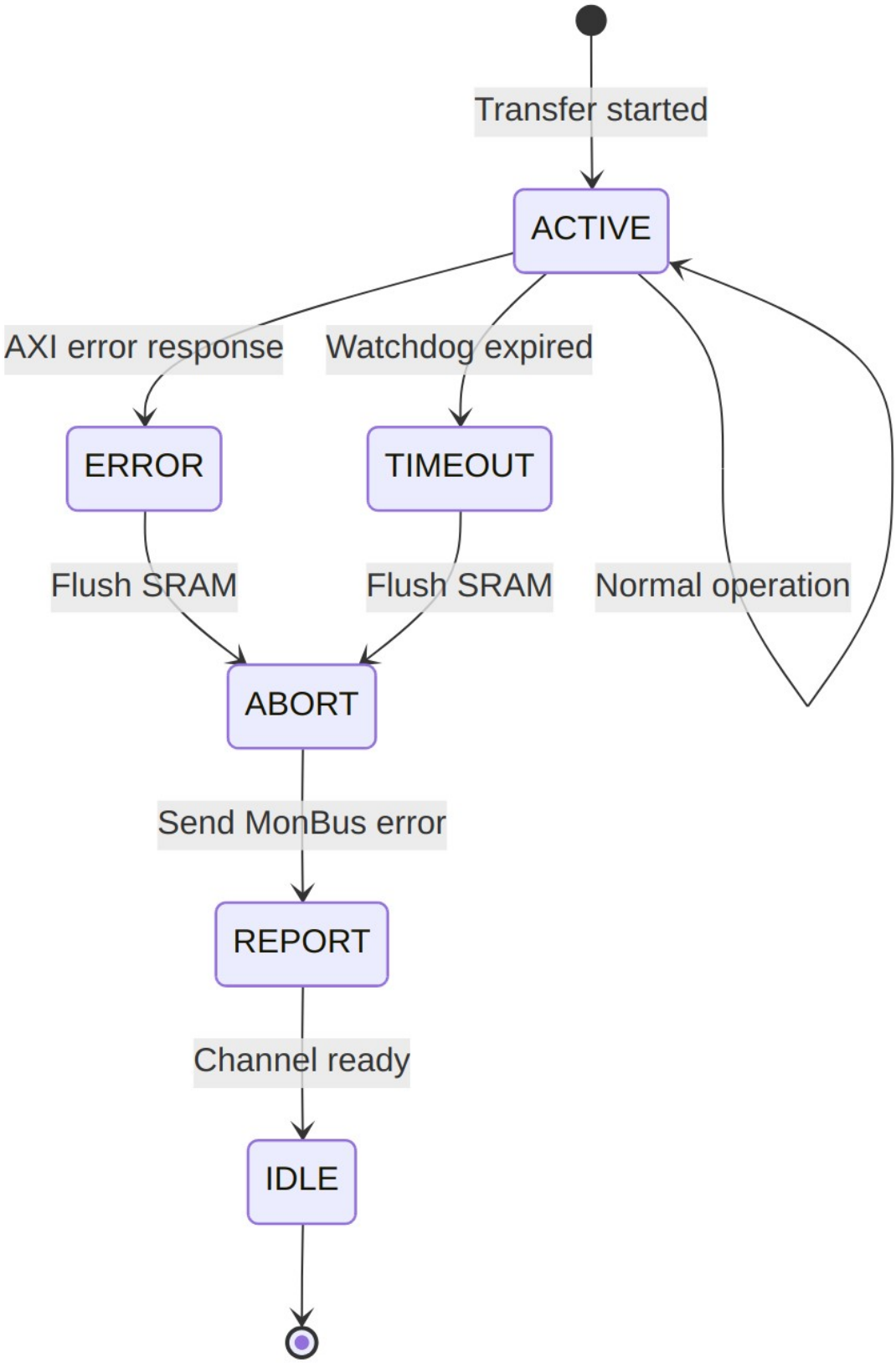
Phase	Typical Cycles
Descriptor fetch	10-50 (memory dependent)
First beat to SRAM	1
SRAM to AXI write	2-4
AXI write completion	10-100 (memory dependent)

16.6 Error Handling

16.6.1 Error Sources

Sink Path Error Handling

Error	Detection	Response
AXI write error	BRESP != OKAY	Stop transfer, report via MonBus
SRAM overflow	Fill pointer catches drain	Should never occur (TREADY control)
Timeout	Watchdog timer	Abort transfer, report error


Transfer started
ACTIVE
AXI error response
Watchdog expired
ERROR
TIMEOUT
Flush SRAM
Flush SRAM
Normal operation
ABORT
Send MonBus error
REPORT
Channel ready
IDLE

Error Recovery Sequence

17 Use Case: Memory to Network (Source Path)

17.1 Overview

The Source Path transfers data from system memory (AXI4 read master) to a network interface (AXI-Stream master). This is the primary transmit data path for network applications.

17.2 Operation Flow

Source Flow

Source Flow

Source: 10_source_flow.mmd

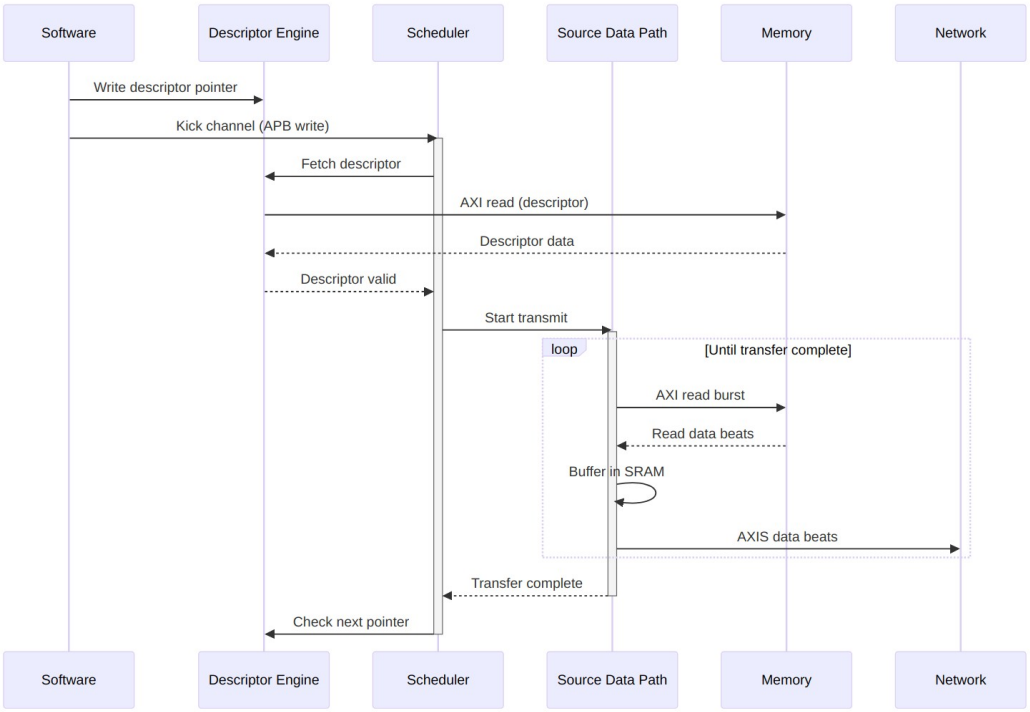


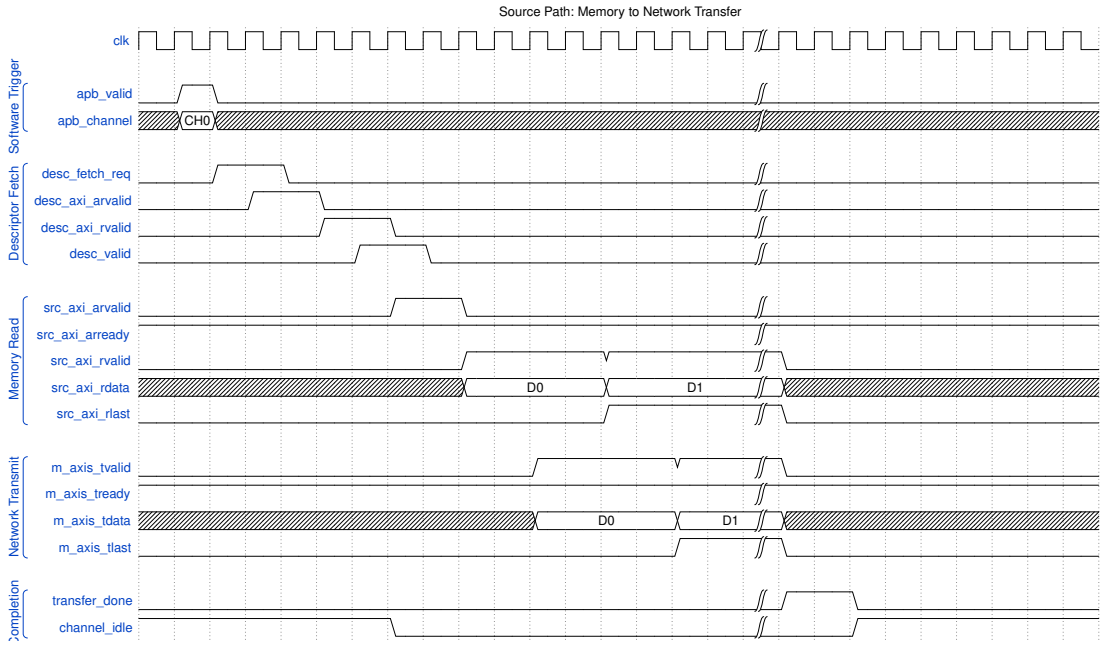
Diagram 16

17.3 Timing Diagram

Source Transfer Timing

Source Transfer Timing

Source: source_transfer.json



Waveform 18

17.4 Buffer Management

17.4.1 SRAM Buffering Strategy

The source path uses internal SRAM to prefetch data and maintain network throughput:

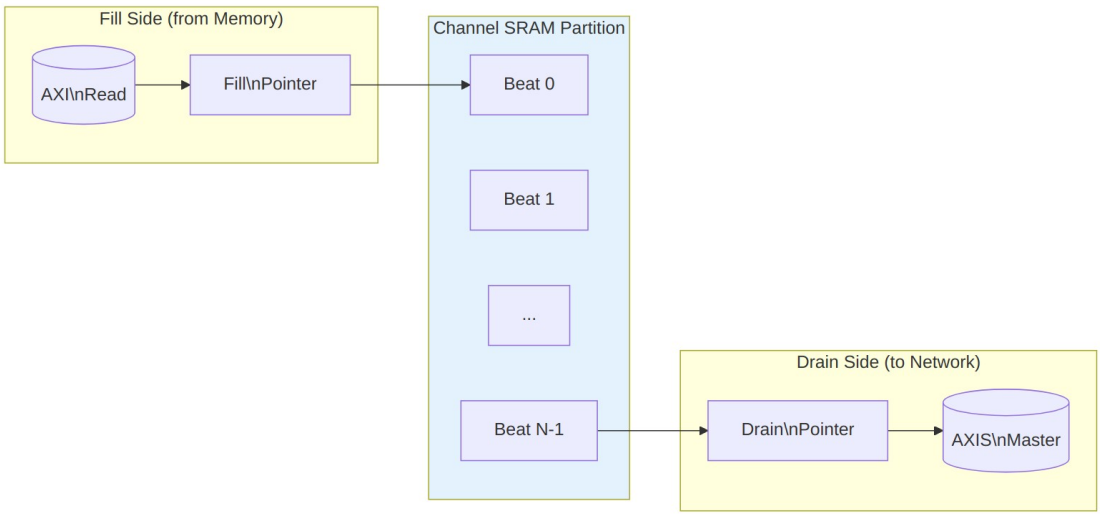
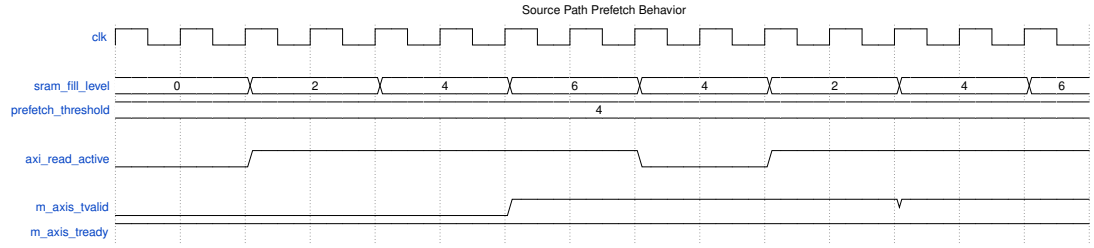


Diagram 17

17.4.2 Flow Control

Source Path Flow Control

Condition	Behavior
SRAM full	Pause AXI reads
SRAM empty	Hold TVALID low
Network backpressure	Stop draining SRAM

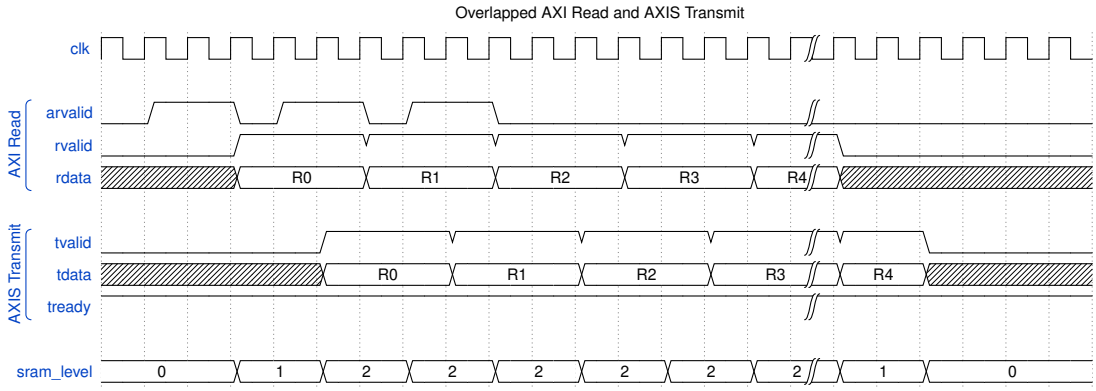


Prefetch Strategy

17.5 Concurrent Operation

17.5.1 Read-Ahead Pipeline

The source path can issue AXI reads ahead of network consumption:



Waveform 20

17.6 Performance Considerations

17.6.1 Throughput

Source Path Throughput Factors

Factor	Impact
Memory bandwidth	Upper bound on transmit rate
Network acceptance	Must keep pace with reads

Factor	Impact
SRAM depth	Read-ahead capability
AXI outstanding	Overlapped read latency hiding

17.6.2 Latency

Source Path Latency Breakdown

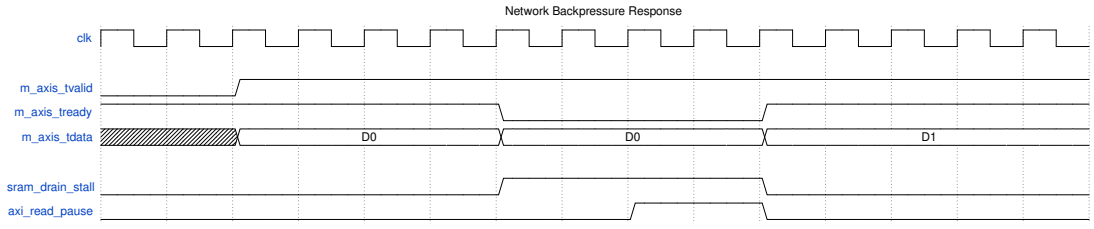
Phase	Typical Cycles
Descriptor fetch	10-50 (memory dependent)
AXI read issue	1-2
Memory read latency	10-100 (memory dependent)
SRAM to network	1-2

17.7 Error Handling

17.7.1 Error Sources

Source Path Error Handling

Error	Detection	Response
AXI read error	RRESP != OKAY	Stop transfer, report via MonBus
Network timeout	Watchdog timer	Abort transfer, report error
Address range	Descriptor validation	Reject descriptor



Backpressure Handling

When the network deasserts TREADY: 1. SRAM drain stalls immediately 2. SRAM continues filling until threshold 3. AXI reads pause when SRAM approaches full 4. Resume automatically when TREADY returns

18 Use Case: Descriptor Chaining

18.1 Overview

Descriptor chaining enables multiple transfers to execute sequentially without software intervention. Each descriptor contains a pointer to the next descriptor, forming a linked list that RAPIDS processes automatically.

18.2 Chain Structure

Descriptor Chain

Descriptor Chain

Source: 11_descriptor_chain.mmd

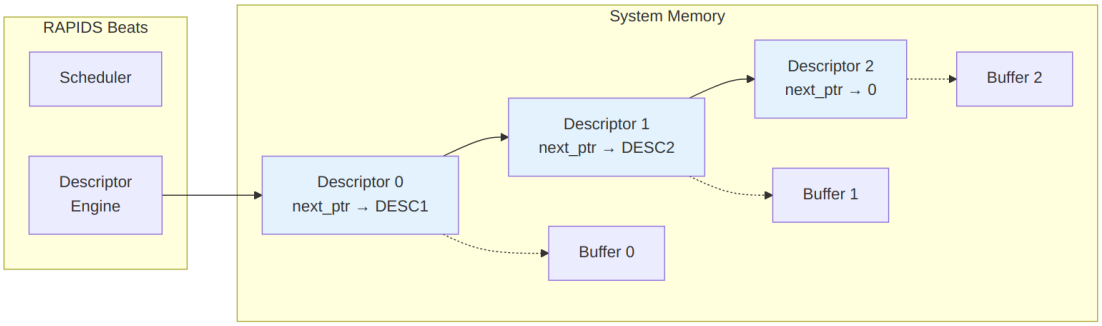
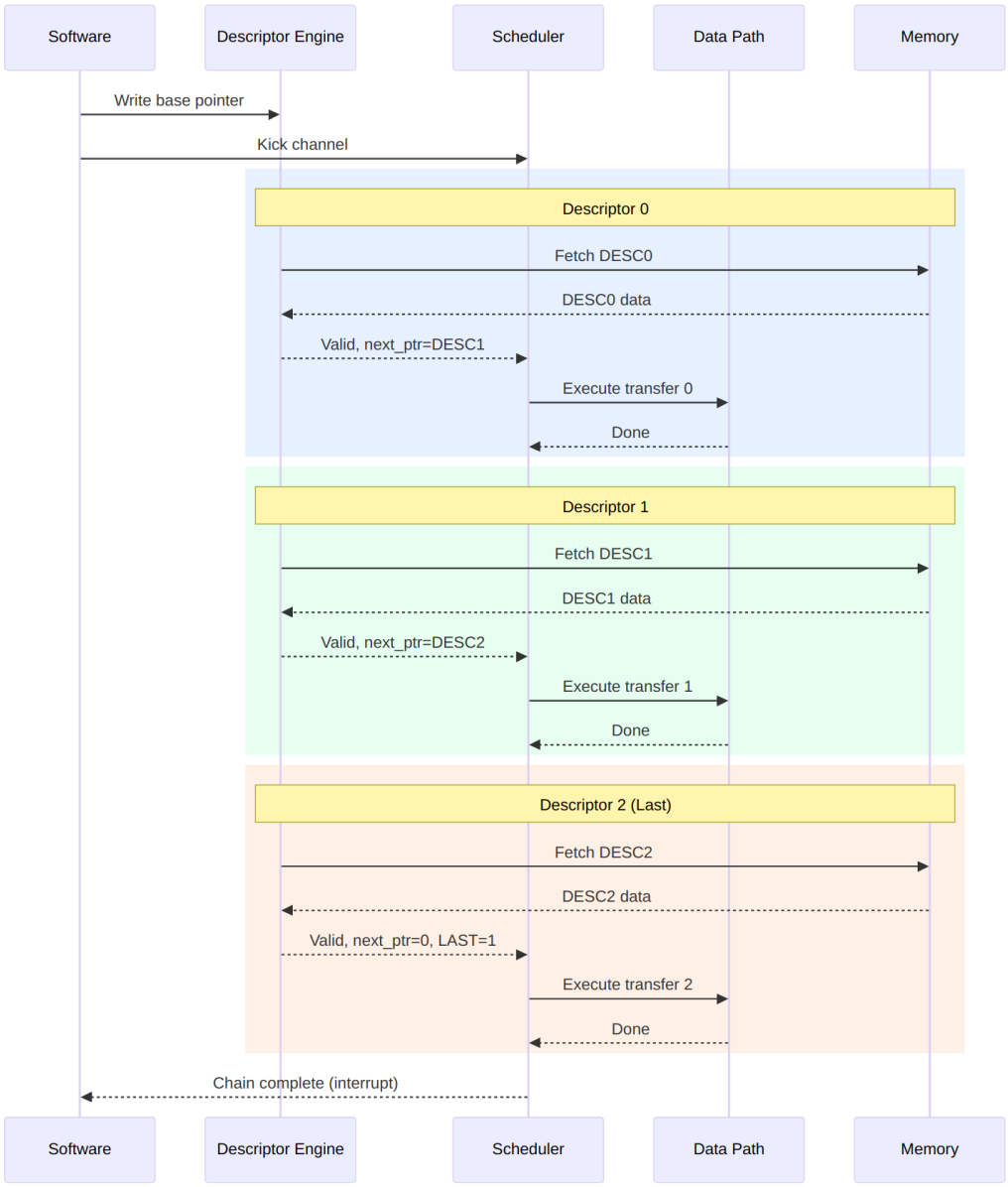


Diagram 18



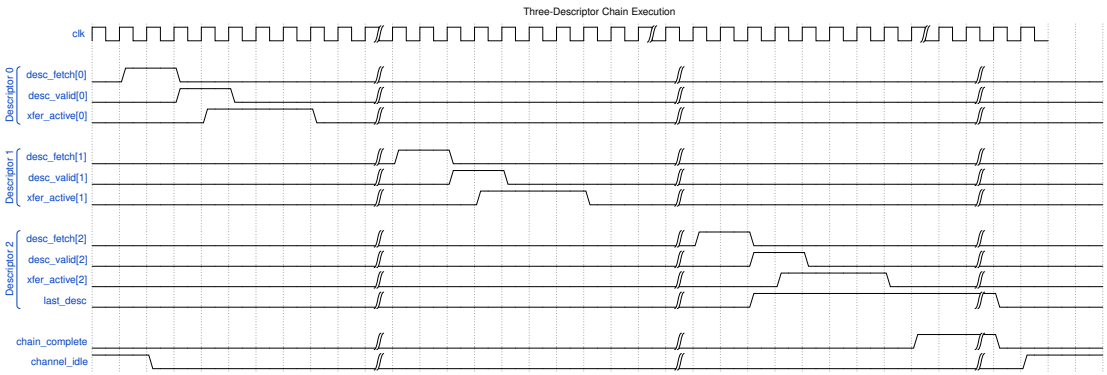
Chain Execution Flow

18.3 Timing Diagram

Chain Execution

Chain Execution

Source: chain_execution.json



Waveform 22

18.4 Descriptor Fields for Chaining

18.4.1 Relevant Fields

Chaining-Related Descriptor Fields

Field	Bits	Description
`next_ptr`	[191:128]	64-bit pointer to next descriptor
`last`	[255]	Last descriptor in chain flag
`irq_en`	[254]	Generate interrupt on completion

18.4.2 Chain Termination

A chain terminates when ANY of these conditions is true:

1. `next_ptr == 0` (null pointer)
2. `last == 1` (explicit last flag)
3. Error during transfer

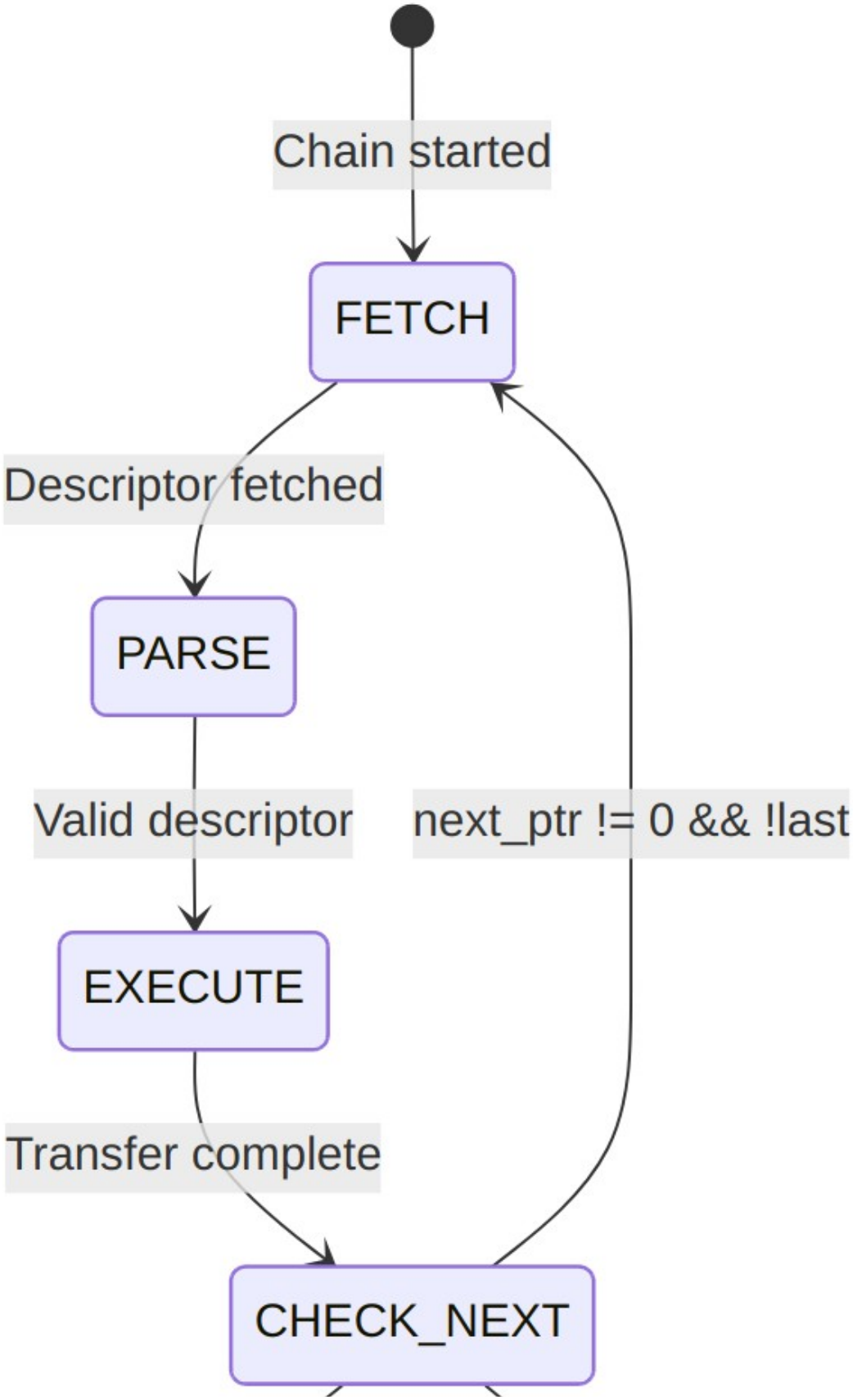
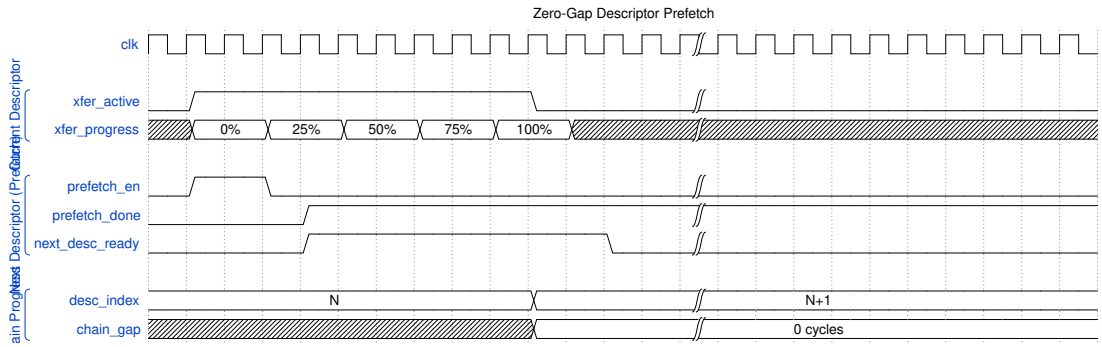

Chain started
FETCH
Descriptor fetched
PARSE
Valid descriptor
EXECUTE
Transfer complete
CHECK_NEXT
next_ptr != 0 && !last

Diagram 20

18.5 Prefetch Optimization

18.5.1 Descriptor Prefetching

RAPIDS can prefetch the next descriptor while the current transfer executes:



Waveform 23

Benefit: Zero-cycle gap between chained transfers when prefetch completes before current transfer.

18.6 Multi-Channel Chaining

Each channel maintains independent chains:

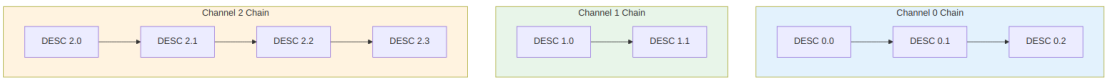


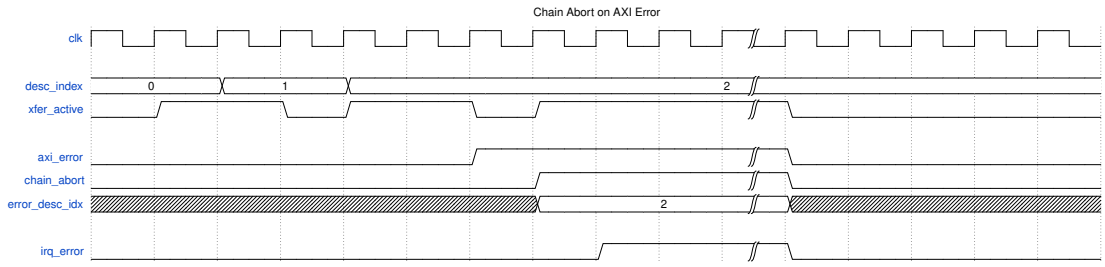
Diagram 21

Channels execute their chains concurrently and independently.

18.7 Error During Chain

18.7.1 Error Handling

When an error occurs mid-chain:



Waveform 24

Error Response: 1. Current transfer aborts 2. Chain processing stops 3. Error status captures failing descriptor index 4. MonBus reports error event 5. Interrupt generated (if enabled)

18.7.2 Recovery

Software must: 1. Read error status to identify failing descriptor 2. Fix the issue (descriptor content, memory mapping, etc.) 3. Restart chain from failing descriptor or beginning

19 Use Case: Multi-Channel Operation

19.1 Overview

RAPIDS Beats supports 8 independent DMA channels that can operate concurrently. This enables parallel data transfers for multiple network connections, priority-based scheduling, and efficient resource utilization.

19.2 Channel Independence

Multi-Channel Architecture

Multi-Channel Architecture

Source: 12_multi_channel.mmd

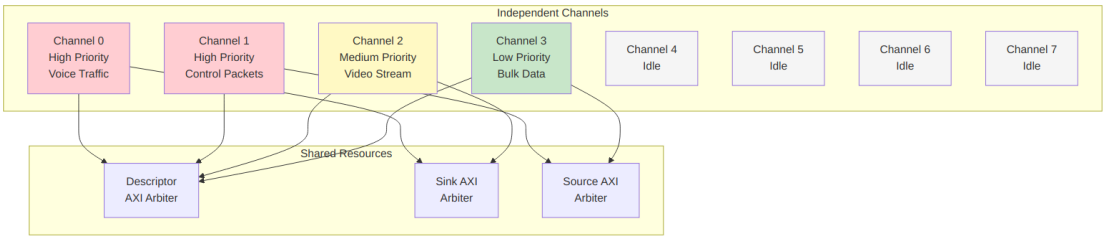


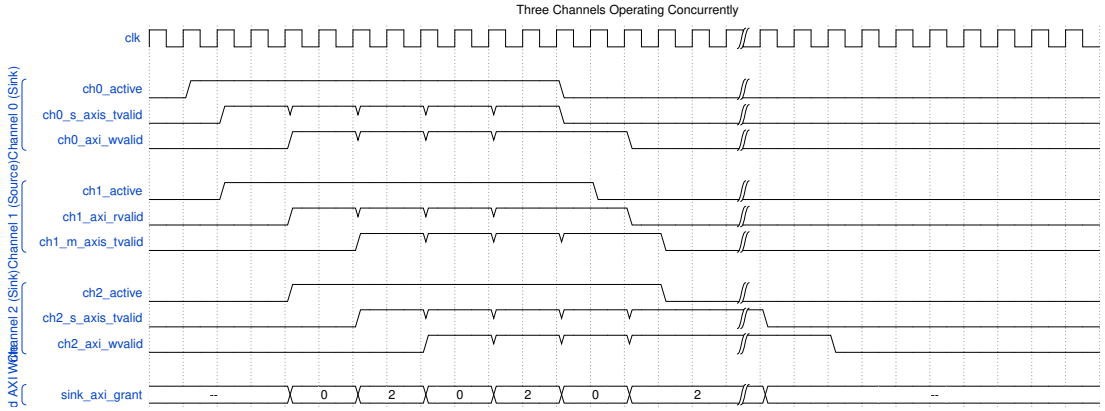
Diagram 22

19.3 Concurrent Operation Timing

Multi-Channel Timing

Multi-Channel Timing

Source: multi_channel_timing.json

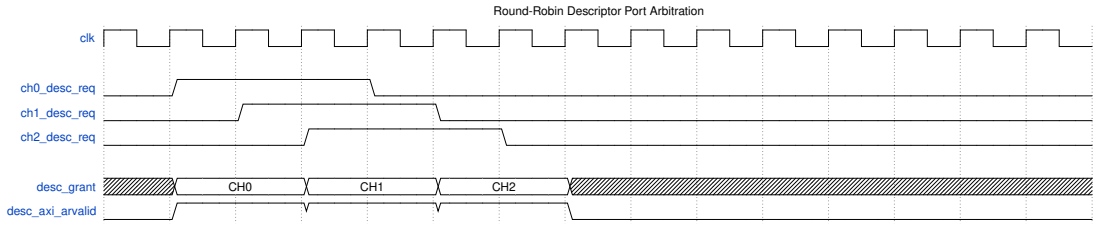


Waveform 25

19.4 Resource Arbitration

19.4.1 Descriptor Port Arbitration

All channels share a single AXI port for descriptor fetches:



Waveform 26

19.4.2 Sink/Source Arbitration

Data path AXI ports use round-robin arbitration with optional priority:

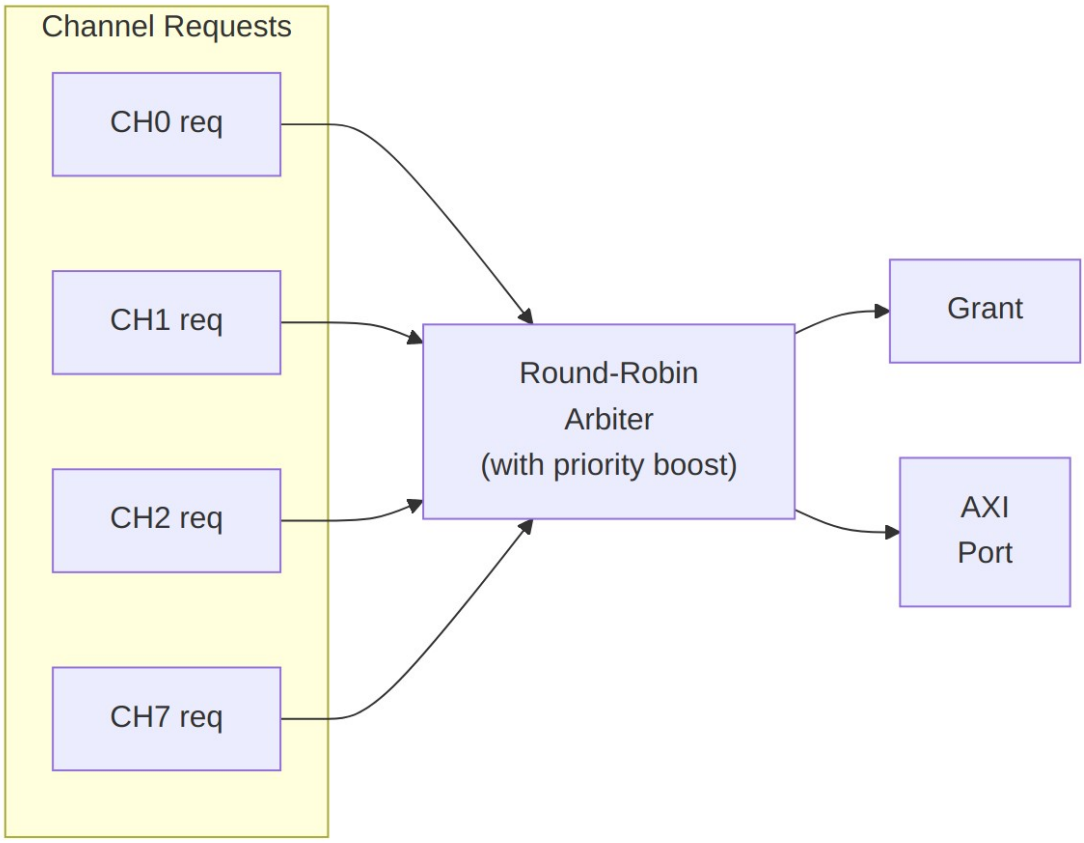
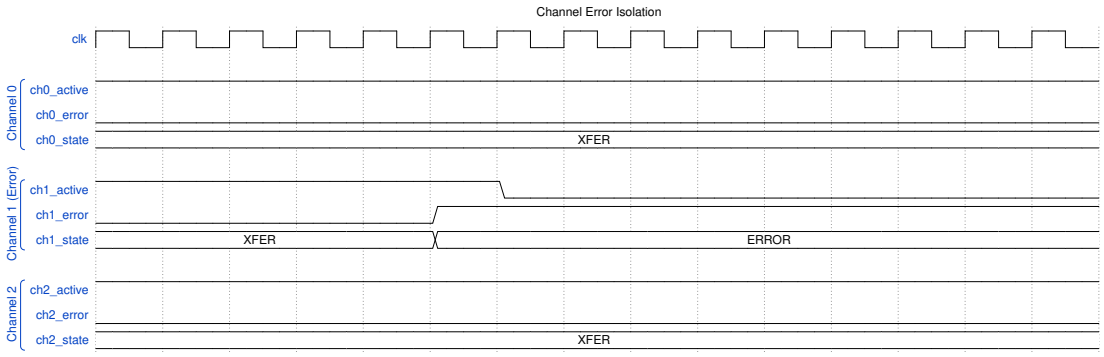


Diagram 23

19.5 Channel Isolation

19.5.1 Error Isolation

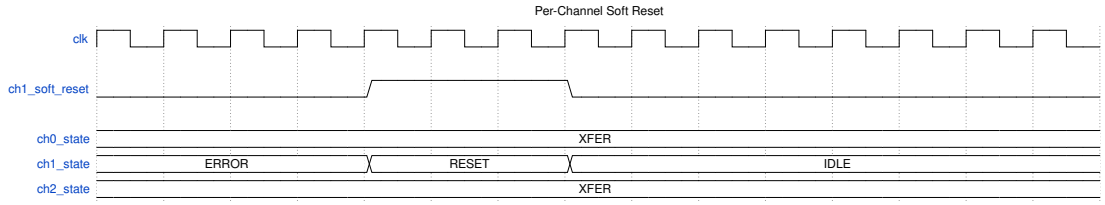
Errors in one channel do not affect other channels:



Waveform 27

19.5.2 Per-Channel Soft Reset

Individual channels can be reset without affecting others:



Waveform 28

19.6 Use Case Examples

19.6.1 Mixed Traffic Pattern

Typical network application with different traffic types:

Mixed Traffic Channel Assignment

Channel	Traffic Type	Direction	Priority	Notes
0	Voice/RTP	Sink	High	Low latency required
1	Voice/RTP	Source	High	Low latency required
2	Video	Sink	Medium	High bandwidth
3	Control	Source	Medium	Small packets
4-7	Bulk data	Both	Low	Background transfers

19.6.2 Bidirectional Transfer

Single network connection using paired channels:

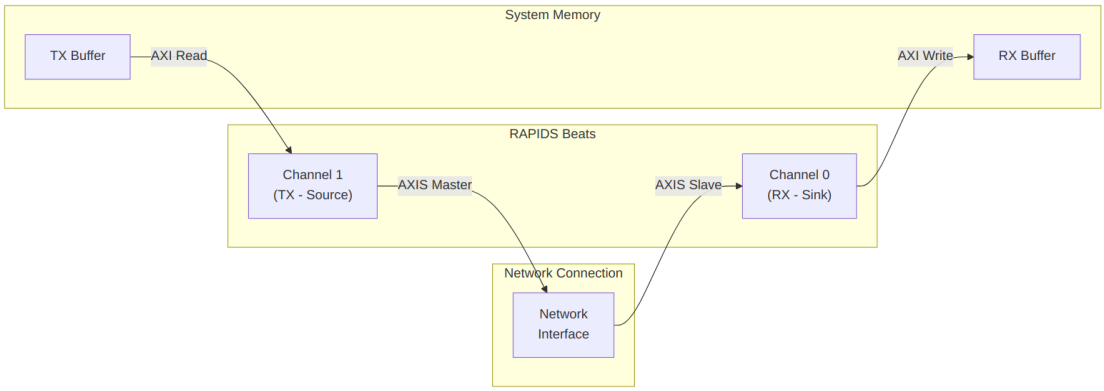


Diagram 24

19.6.3 Load Balancing

Multiple channels processing same traffic type:

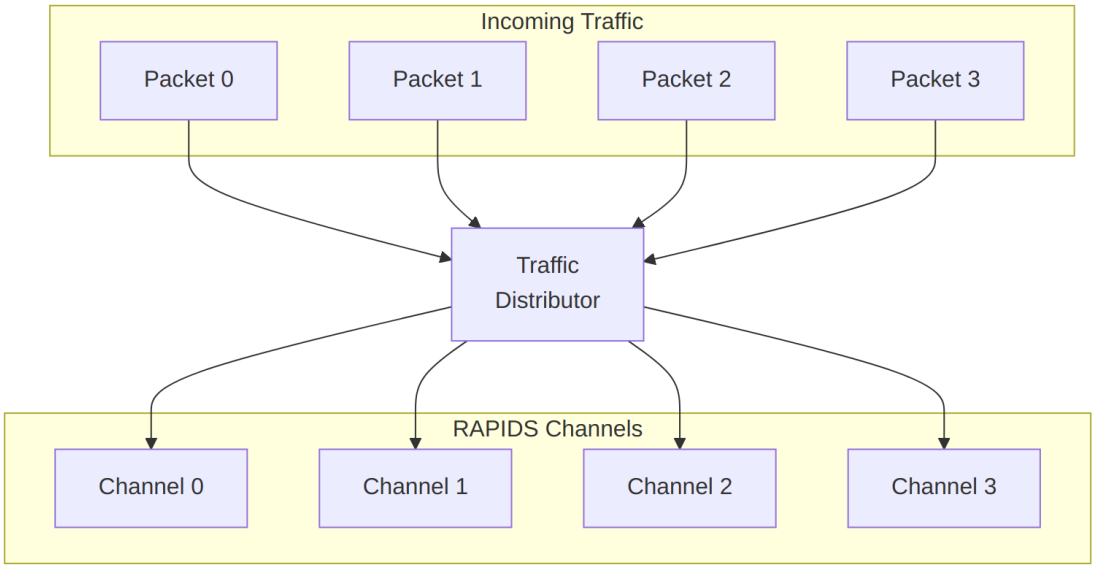


Diagram 25

19.7 Performance Considerations

19.7.1 Bandwidth Sharing

Bandwidth Distribution

Active Channels	Per-Channel BW (% of max)	Notes
1	100%	Full bandwidth available

Active Channels	Per-Channel BW (% of max)	Notes
2	~50% each	Round-robin sharing
4	~25% each	Fair distribution
8	~12.5% each	All channels active

Note: Actual distribution depends on transfer sizes and memory latency.

19.7.2 Latency Impact

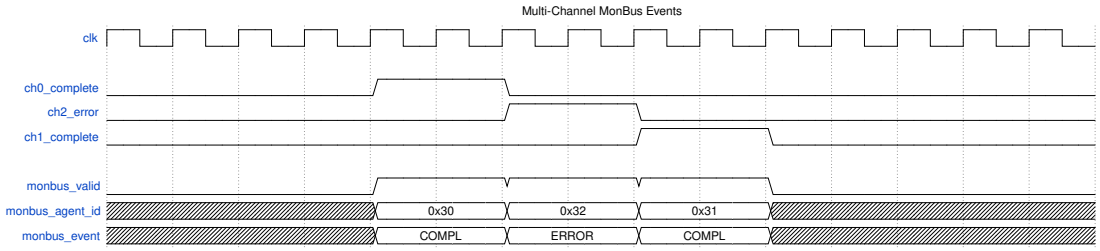
Multi-channel operation adds arbitration latency:

Arbitration Latency

Scenario	Additional Latency
Single channel	0 cycles
2 channels contending	1-2 cycles
8 channels contending	Up to 7 cycles

19.8 MonBus Events

Each channel generates independent MonBus events:



Waveform 29

Events are arbitrated and serialized on the shared MonBus output.

20 Descriptor Format

20.1 Overview

RAPIDS Beats uses 256-bit (32-byte) descriptors to define DMA transfers. Descriptors are stored in system memory and fetched by the Descriptor Engine when a channel is kicked.

20.2 Descriptor Layout

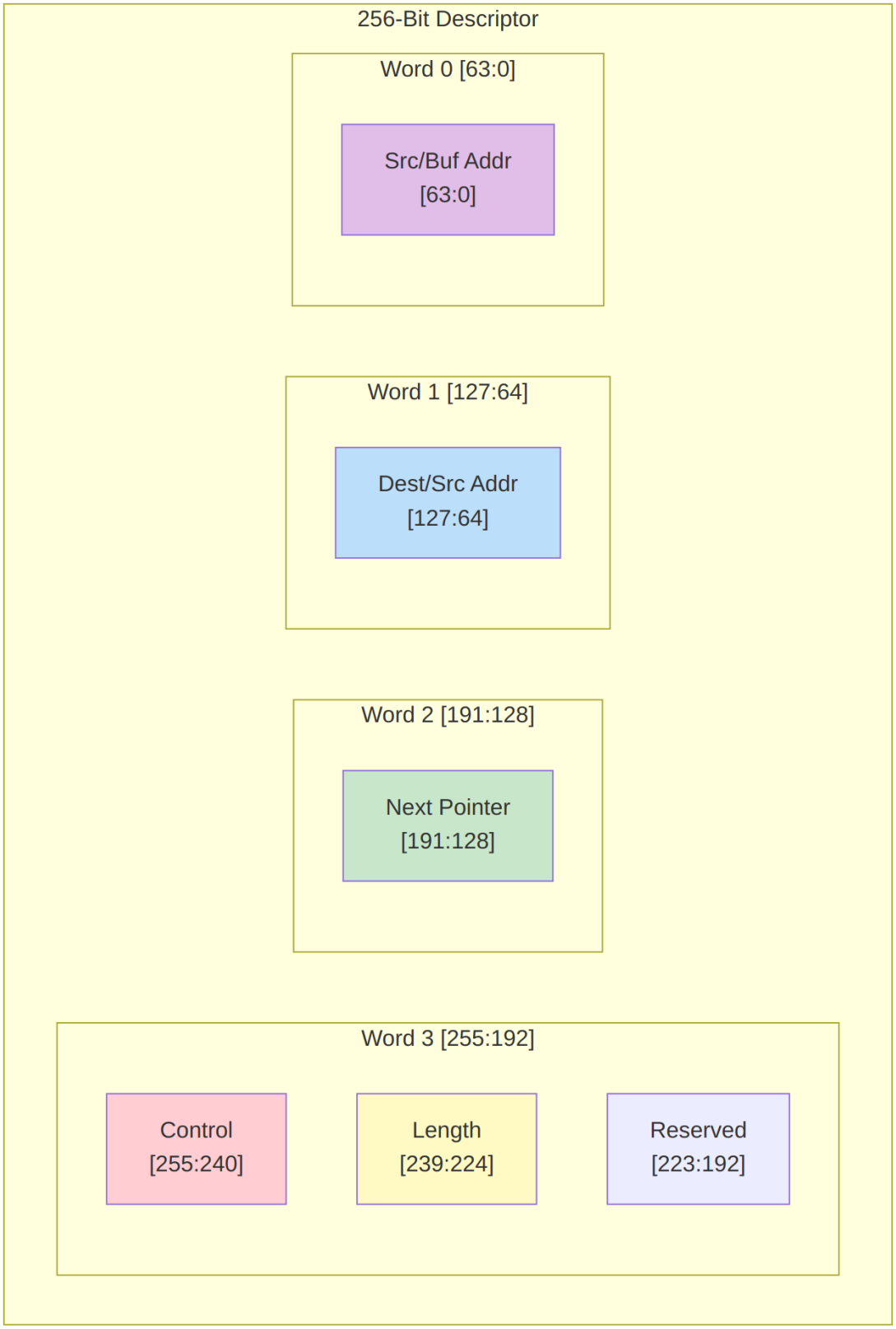
Descriptor Format

Descriptor Format

Source: 13_descriptor_format.mmd

20.2.1 256-Bit Structure

Bit Range	Field Name	Width	Description
[255]	last	1	Last descriptor in chain
[254]	irq_en	1	Generate interrupt on completion
[253:252]	reserved	2	Reserved (write 0)
[251:248]	direction	4	Transfer direction
[247:240]	reserved	8	Reserved (write 0)
[239:224]	transfer_length	16	Transfer length in beats
[223:192]	reserved	32	Reserved (write 0)
[191:128]	next_ptr	64	Pointer to next descriptor
[127:64]	dest_addr	64	Destination address (sink) or source address
[63:0]	src_addr	64	Source address (source path) or buffer addr



Visual Layout

20.3 Field Descriptions

20.3.1 Control Field [255:248]

Control Field Bits

Bit	Field	Description
255	`last`	Set to 1 for last descriptor in chain
254	`irq_en`	Set to 1 to generate completion interrupt
253:252	Reserved	Write as 0
251:248	`direction`	Transfer direction encoding

20.3.2 Direction Encoding

Direction Field Encoding

Value	Name	Description
4'h0	SINK	Network to memory (AXIS slave to AXI write)
4'h1	SOURCE	Memory to network (AXI read to AXIS master)
4'h2-F	Reserved	Reserved for future use

20.3.3 Transfer Length [239:224]

- 16-bit field specifying transfer size in beats
- 1 beat = DATA_WIDTH bits (default 512 bits = 64 bytes)
- Maximum transfer: 65,535 beats (4 MB at 64 bytes/beat)
- Length of 0 is reserved (no operation)

20.3.4 Next Pointer [191:128]

- 64-bit byte address of next descriptor
- Must be 32-byte aligned (bits [4:0] ignored)
- Value of 0 terminates chain (regardless of `last` bit)

20.3.5 Address Fields [127:0]

For SINK transfers (network to memory): | Field | Usage | |——-|——-| | `dest_addr` [127:64] | Memory write destination address | | `src_addr` [63:0] | Not used (write 0) |

For SOURCE transfers (memory to network): | Field | Usage | |——-|——-| | `dest_addr` [127:64] | Not used (write 0) | | `src_addr` [63:0] | Memory read source address |

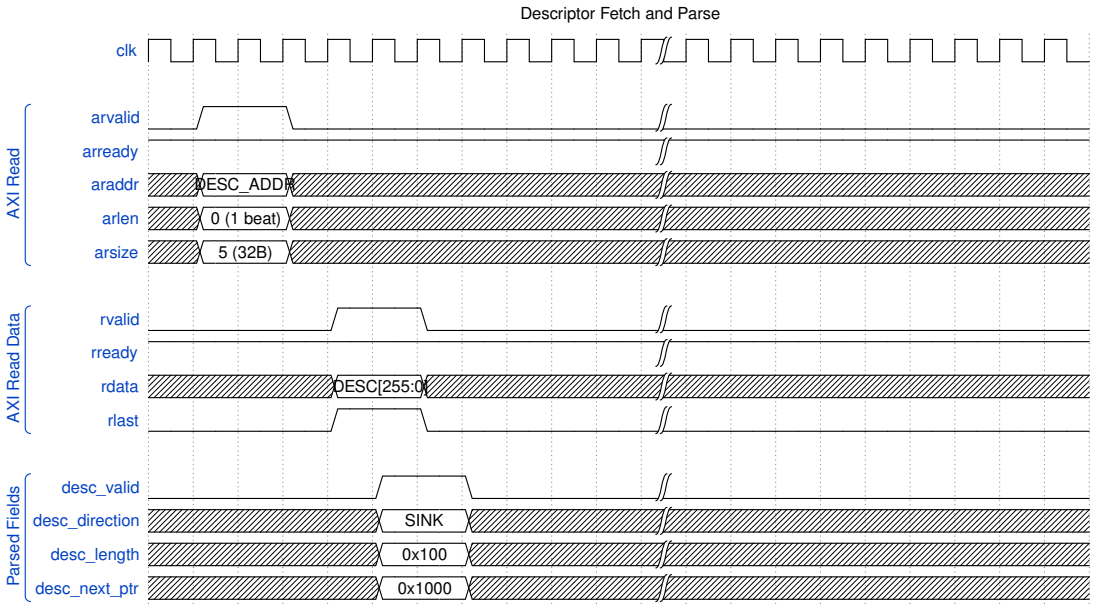
: Address Field Usage

20.4 Descriptor Fetch Timing

Descriptor Fetch

Descriptor Fetch

Source: descriptor_fetch.json



Waveform 30

20.5 Alignment Requirements

Address Alignment Requirements

Field	Alignment	Notes
Descriptor address	32-byte	Bits [4:0] ignored
`next_ptr`	32-byte	Bits [4:0] ignored
`dest_addr`	DATA_WIDTH/8	64-byte for 512-bit

Field	Alignment	Notes
		data
src_addr	DATA_WIDTH/8	64-byte for 512-bit data

Note: Unaligned addresses may cause unpredictable behavior or AXI protocol violations.

20.6 Descriptor Examples

20.6.1 Sink Descriptor (Network to Memory)

```
Hex representation (LSB to MSB):
[63:0]    = 0x0000_0000_0000_0000  // src_addr (unused)
[127:64]  = 0x0000_0001_0000_0000  // dest_addr = 0x1_0000_0000
[191:128] = 0x0000_0000_0000_2000  // next_ptr = 0x2000
[255:192] = 0xC000_0100_0000_0000  // last=1, irq=1, dir=SINK, len=256

Binary control field:
  [255] = 1      // Last descriptor
  [254] = 1      // Interrupt enabled
  [253:252] = 00 // Reserved
  [251:248] = 0000 // Direction = SINK
```

20.6.2 Source Descriptor (Memory to Network)

```
Hex representation (LSB to MSB):
[63:0]    = 0x0000_0002_0000_0000  // src_addr = 0x2_0000_0000
[127:64]  = 0x0000_0000_0000_0000  // dest_addr (unused)
[191:128] = 0x0000_0000_0000_0000  // next_ptr = 0 (last)
[255:192] = 0x4001_0080_0000_0000  // last=0, irq=1, dir=SOURCE,
len=128

Binary control field:
  [255] = 0      // Not last (but next_ptr=0 terminates)
  [254] = 1      // Interrupt enabled
  [253:252] = 00 // Reserved
  [251:248] = 0001 // Direction = SOURCE
```

20.7 Software Construction

20.7.1 C Structure Example

```c
typedef struct __attribute__((packed, aligned(32))) {
    uint64_t src_addr;      // [63:0]
    uint64_t dest_addr;     // [127:64]
    uint64_t next_ptr;      // [191:128]
    uint16_t reserved1;     // [207:192]
    uint16_t length;        // [223:208] - Note: shifted in actual
layout
```

```
    uint8_t  reserved2;     // [231:224]
    uint8_t  direction;     // [239:232] - lower 4 bits only
    uint8_t  reserved3;     // [247:240]
    uint8_t  control;       // [255:248] - last, irq_en, etc.
} rapids_descriptor_t;

// Helper macros
#define DESC_CTRL_LAST    (1 << 7)
#define DESC_CTRL_IRQ_EN  (1 << 6)
#define DESC_DIR_SINK     0x0
#define DESC_DIR_SOURCE   0x1
```

20.7.2 Descriptor Ring Setup

```
// Allocate descriptor ring (must be 32-byte aligned)
rapids_descriptor_t *desc_ring = aligned_alloc(32, NUM_DESC *
sizeof(rapids_descriptor_t));

// Initialize chain
for (int i = 0; i < NUM_DESC - 1; i++) {
    desc_ring[i].next_ptr = (uint64_t)&desc_ring[i + 1];
    desc_ring[i].control = DESC_CTRL_IRQ_EN;
}

// Last descriptor
desc_ring[NUM_DESC - 1].next_ptr = 0;
desc_ring[NUM_DESC - 1].control = DESC_CTRL_LAST | DESC_CTRL_IRQ_EN;
```

21 Register Map

21.1 Overview

RAPIDS Beats provides a memory-mapped register interface for configuration and status. Registers are accessed via the APB interface and organized by function.

21.2 Register Summary

21.2.1 Global Registers (Offset 0x000-0x0FF)

Global Registers

Offset	Name	Width	Access	Description
0x000	VERSION	32	RO	IP version register
0x004	CONFIG	32	RO	Configuration

Offset	Name	Width	Access	Description
				parameters
0x008	GLOBAL_CTRL	32	RW	Global control
0x00C	GLOBAL_STATUS	32	RO	Global status
0x010	IRQ_STATUS	32	RW1C	Interrupt status
0x014	IRQ_ENABLE	32	RW	Interrupt enable
0x018	IRQ_FORCE	32	WO	Force interrupt (debug)

21.2.2 Per-Channel Registers (Offset 0x100 + ch*0x40)

Per-Channel Registers

Offset	Name	Width	Access	Description
+0x00	CH_CTRL	32	RW	Channel control
+0x04	CH_STATUS	32	RO	Channel status
+0x08	DESC_PTR_LO	32	RW	Descriptor pointer [31:0]
+0x0C	DESC_PTR_HI	32	RW	Descriptor pointer [63:32]
+0x10	XFER_COUNT	32	RO	Beats transferred
+0x14	ERR_STATUS	32	RW1C	Error status
+0x18	CH_CONFIG	32	RW	Channel configuration

21.3 Register Descriptions

21.3.1 VERSION (0x000) - Read Only

VERSION Register

Bits	Field	Reset	Description
[31:24]	`major`	0x01	Major version
[23:16]	`minor`	0x00	Minor version
[15:0]	`patch`	0x0000	Patch level

21.3.2 CONFIG (0x004) - Read Only

CONFIG Register

Bits	Field	Reset	Description
[31:24]	`num_channels`	0x08	Number of channels
[23:16]	`data_width`	0x40	Data width / 8 (64 = 512-bit)
[15:8]	`sram_depth`	varies	SRAM depth / 16
[7:0]	`features`	varies	Feature bits

21.3.3 GLOBAL_CTRL (0x008) - Read/Write

GLOBAL_CTRL Register

Bits	Field	Reset	Description
[31]	`soft_reset`	0	Write 1 to reset all channels
[30:8]	Reserved	0	Reserved
[7:0]	`clock_gate_en`	0xFF	Per-channel clock gate enable

21.3.4 GLOBAL_STATUS (0x00C) - Read Only

GLOBAL_STATUS Register

Bits	Field	Reset	Description
[31:24]	Reserved	0	Reserved
[23:16]	`error_channels`	0	Channels with

Bits	Field	Reset	Description
			errors
[15:8]	`active_channels`	0	Channels in transfer
[7:0]	`idle_channels`	0xFF	Channels idle

21.3.5 IRQ_STATUS (0x010) - Read/Write-1-to-Clear

IRQ_STATUS Register

Bits	Field	Reset	Description
[31:24]	`ch_error`	0	Channel error interrupts
[23:16]	Reserved	0	Reserved
[15:8]	`ch_complete`	0	Channel completion interrupts
[7:0]	Reserved	0	Reserved

21.3.6 CH_CTRL (0x100 + ch*0x40) - Read/Write

CH_CTRL Register

Bits	Field	Reset	Description
[31]	`enable`	0	Channel enable
[30]	`kick`	0	Write 1 to start (self-clearing)
[29]	`abort`	0	Write 1 to abort (self-clearing)
[28]	`soft_reset`	0	Write 1 to reset channel
[27:0]	Reserved	0	Reserved

21.3.7 CH_STATUS (0x104 + ch*0x40) - Read Only

CH_STATUS Register

Bits	Field	Reset	Description
[31:28]	`state`	0	FSM state

Bits	Field	Reset	Description
			encoding
[27:24]	Reserved	0	Reserved
[23:16]	`desc_count`	0	Descriptors processed
[15:8]	`sram_level`	0	Current SRAM fill level
[7:0]	`flags`	0	Status flags

State Encoding:

CH_STATUS State Encoding

Value	State
0x0	IDLE
0x1	WAIT_DESC
0x2	PARSE_DESC
0x3	XFER_DATA
0x4	CHECK_NEXT
0x5	COMPLETE
0xE	ERROR
0xF	RESET

21.3.8 ERR_STATUS (0x114 + ch*0x40) - Read/Write-1-to-Clear

ERR_STATUS Register

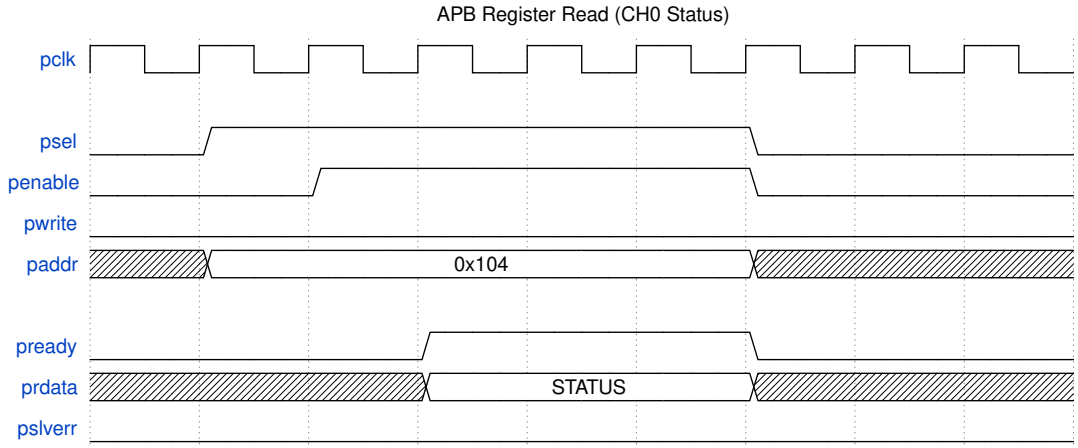
Bits	Field	Reset	Description
[31:28]	`axi_resp`	0	AXI error response code
[27:24]	`error_type`	0	Error type encoding
[23:16]	`error_desc_idx`	0	Descriptor index at error
[15:0]	Reserved	0	Reserved

21.4 Register Access Timing

Register Read

Register Read

Source: register_read.json

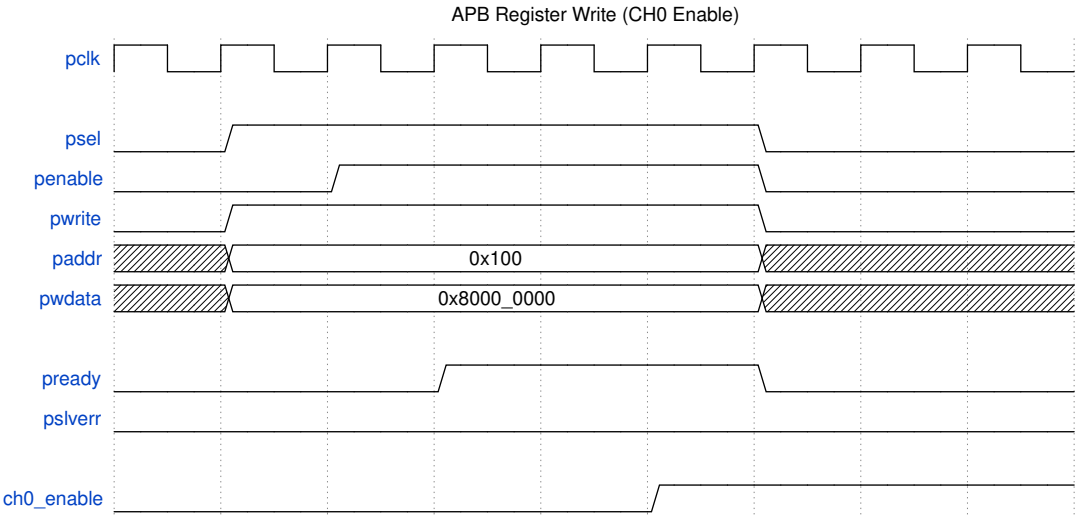


Waveform 31

Register Write

Register Write

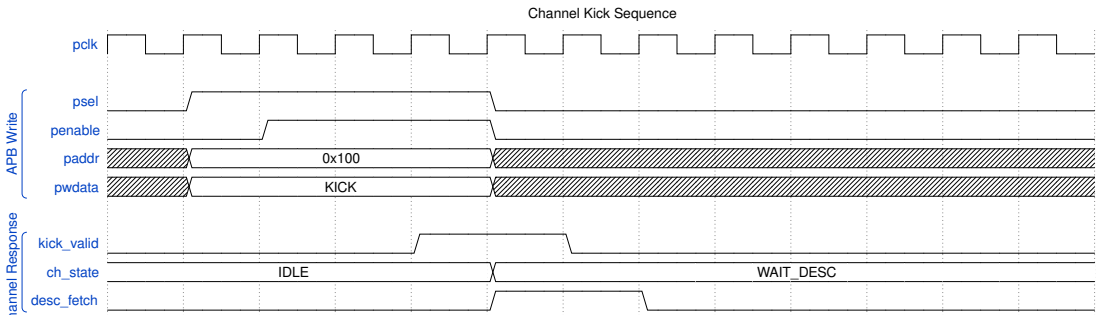
Source: register_write.json



Waveform 32

21.5 Channel Kick Sequence

Writing to the `kick` bit triggers descriptor processing:



Waveform 33

21.6 Address Decoding

21.6.1 Address Space Layout

```
0x000 - 0x0FF: Global registers
0x100 - 0x13F: Channel 0 registers
0x140 - 0x17F: Channel 1 registers
0x180 - 0x1BF: Channel 2 registers
0x1C0 - 0x1FF: Channel 3 registers
0x200 - 0x23F: Channel 4 registers
0x240 - 0x27F: Channel 5 registers
0x280 - 0x2BF: Channel 6 registers
0x2C0 - 0x2FF: Channel 7 registers
0x300 - 0x3FF: Reserved
```

21.6.2 Channel Address Formula

```
channel_base = 0x100 + (channel_number * 0x40)
register_addr = channel_base + register_offset
```

Example: Channel 3 CH_STATUS = 0x100 + (3 * 0x40) + 0x04 = 0x1C4

22 Initialization Sequence

22.1 Overview

This section describes the required steps to initialize RAPIDS Beats and start DMA transfers. The initialization sequence ensures proper hardware state before beginning operations.

22.2 Boot Sequence

Initialization Flow

Initialization Flow

Source: 14_init_flow.mmd

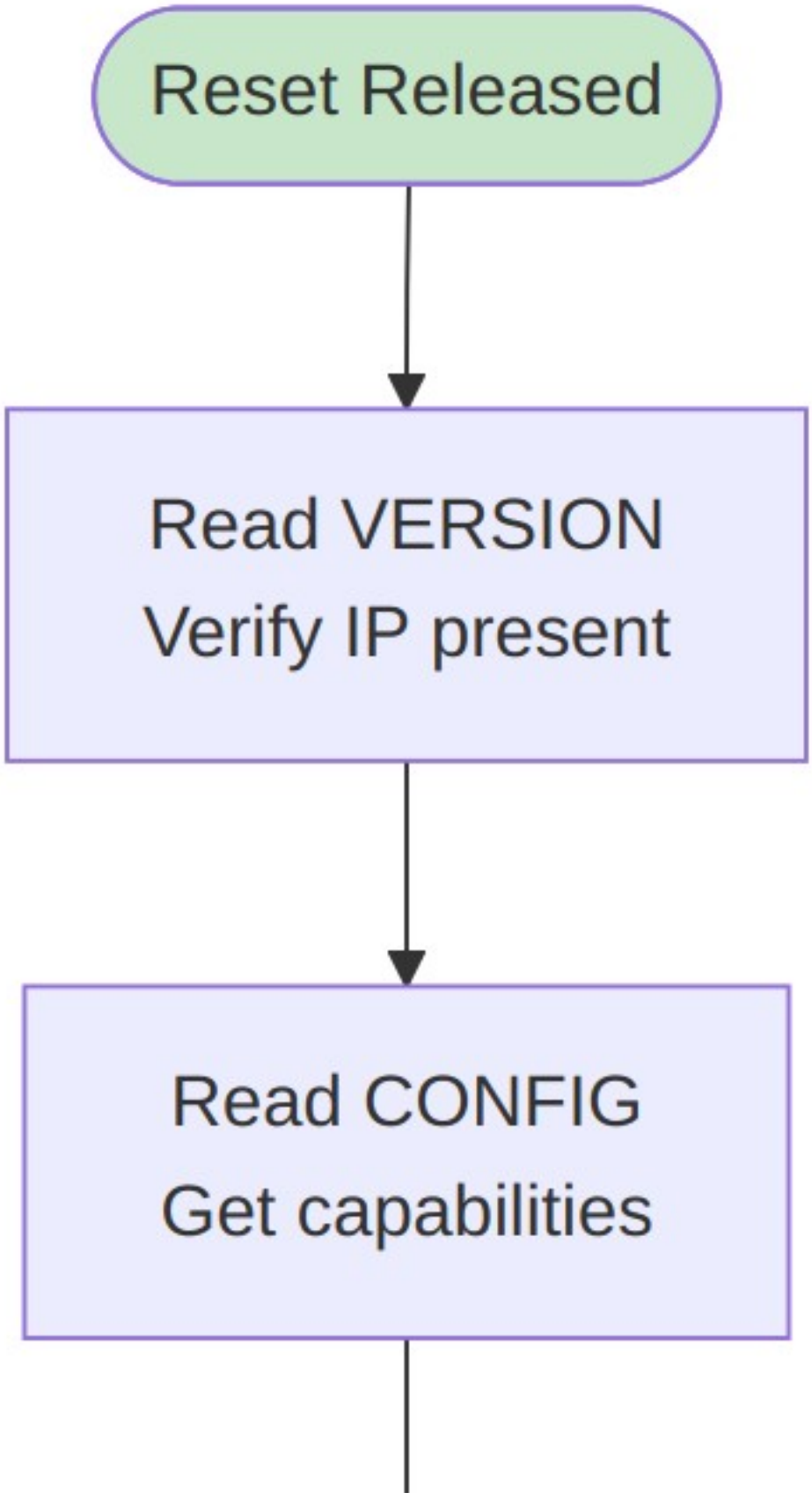

Reset Released
Read VERSION
Verify IP present
Read CONFIG
Get capabilities

Diagram 27

22.3 Detailed Steps

22.3.1 Step 1: Verify Hardware Presence

```
// Read VERSION register
uint32_t version = read_reg(RAPIDS_BASE + VERSION);
uint8_t major = (version >> 24) & 0xFF;
uint8_t minor = (version >> 16) & 0xFF;

if (major < 1) {
    printf("ERROR: RAPIDS not detected or incompatible version\n");
    return -1;
}

printf("RAPIDS Beats v%d.%d detected\n", major, minor);
```

22.3.2 Step 2: Read Configuration

```
// Read CONFIG register
uint32_t config = read_reg(RAPIDS_BASE + CONFIG);
uint8_t num_channels = (config >> 24) & 0xFF;
uint8_t data_width = ((config >> 16) & 0xFF) * 8;  // Convert to bits
uint8_t sram_depth = ((config >> 8) & 0xFF) * 16;

printf("Channels: %d, Data Width: %d bits, SRAM: %d entries\n",
       num_channels, data_width, sram_depth);
```

22.3.3 Step 3: Global Reset

```
// Perform soft reset to ensure clean state
write_reg(RAPIDS_BASE + GLOBAL_CTRL, GLOBAL_CTRL_SOFT_RESET);

// Wait for reset completion
while (read_reg(RAPIDS_BASE + GLOBAL_STATUS) &
GLOBAL_STATUS_RESET_ACTIVE) {
    // Poll until reset complete
}

// Clear any pending interrupts
write_reg(RAPIDS_BASE + IRQ_STATUS, 0xFFFFFFFF);
```

22.3.4 Step 4: Allocate Resources

```
// Allocate descriptor ring (must be 32-byte aligned)
size_t desc_size = NUM_DESCRIPTORS * sizeof(rapids_descriptor_t);
rapids_descriptor_t *desc_ring = aligned_alloc(32, desc_size);
if (!desc_ring) {
    return -ENOMEM;
}
```

```
// Allocate data buffers (must be 64-byte aligned for 512-bit AXI)
size_t buf_size = BUFFER_SIZE_BYTES;
void *rx_buffer = aligned_alloc(64, buf_size);
void *tx_buffer = aligned_alloc(64, buf_size);
```

22.3.5 Step 5: Initialize Descriptors

```
// Initialize sink descriptor chain
for (int i = 0; i < NUM_RX_DESC - 1; i++) {
    desc_ring[i].dest_addr = (uint64_t)(rx_buffer + i * FRAME_SIZE);
    desc_ring[i].src_addr = 0;
    desc_ring[i].next_ptr = (uint64_t)&desc_ring[i + 1];
    desc_ring[i].direction = DESC_DIR_SINK;
    desc_ring[i].length = FRAME_SIZE / 64;  // Convert to beats
    desc_ring[i].control = DESC_CTRL_IRQ_EN;
}

// Last descriptor
desc_ring[NUM_RX_DESC - 1].dest_addr = (uint64_t)(rx_buffer +
(NUM_RX_DESC-1) * FRAME_SIZE);
desc_ring[NUM_RX_DESC - 1].next_ptr = 0;
desc_ring[NUM_RX_DESC - 1].control = DESC_CTRL_LAST |
DESC_CTRL_IRQ_EN;

// Ensure descriptors are visible to hardware
__sync_synchronize();  // Memory barrier
```

22.3.6 Step 6: Configure Channels

```
// Configure channel 0 for sink (receive)
uint32_t ch0_base = RAPIDS_BASE + 0x100;

// Set descriptor pointer
write_reg(ch0_base + DESC_PTR_LO, (uint32_t)((uint64_t)desc_ring &
0xFFFFFFFF));
write_reg(ch0_base + DESC_PTR_HI, (uint32_t)((uint64_t)desc_ring >>
32));

// Configure channel
write_reg(ch0_base + CH_CONFIG, CH_CONFIG_DEFAULT);
```

22.3.7 Step 7: Enable Interrupts

```
// Enable completion and error interrupts for channel 0
write_reg(RAPIDS_BASE + IRQ_ENABLE, IRQ_CH0_COMPLETE | IRQ_CH0_ERROR);
```

22.3.8 Step 8: Enable Channel

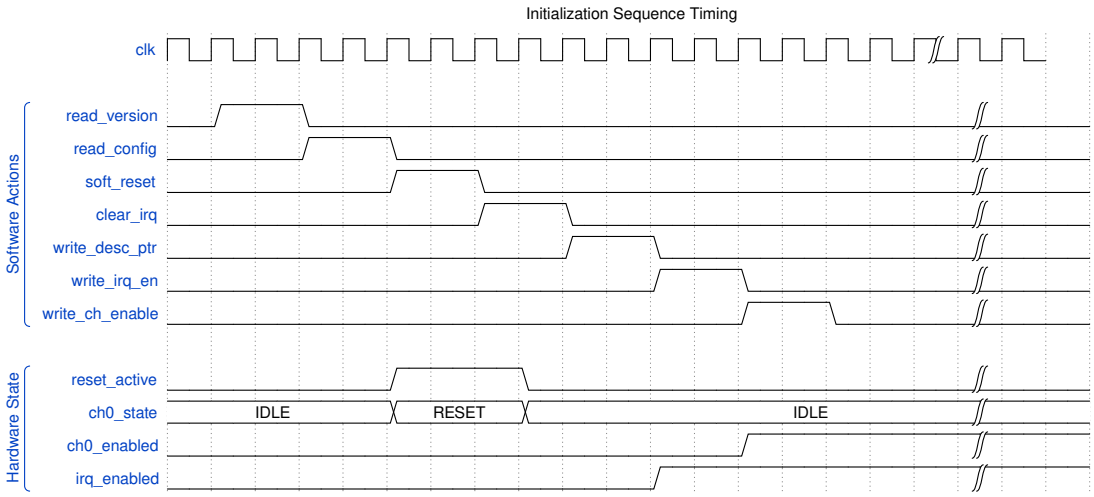
```
// Enable channel (but don't kick yet)
write_reg(ch0_base + CH_CTRL, CH_CTRL_ENABLE);
```

22.4 Initialization Timing

Initialization Timing

Initialization Timing

Source: init_timing.json



Waveform 34

22.5 Starting Transfers

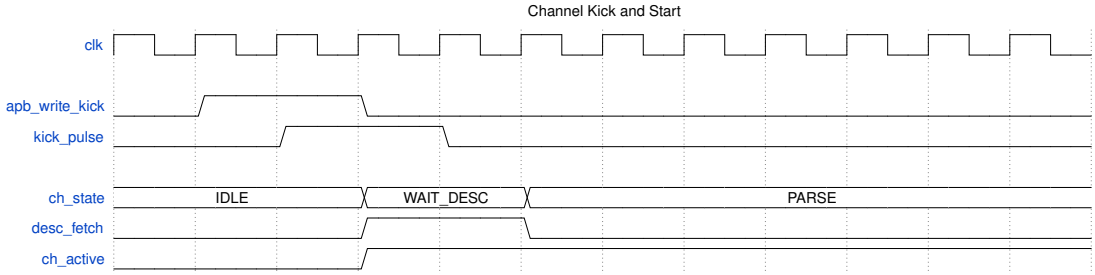
22.5.1 Kick a Channel

After initialization, kick the channel to start processing:

```
// Kick channel 0 to start descriptor processing
write_reg(ch0_base + CH_CTRL, CH_CTRL_ENABLE | CH_CTRL_KICK);
```

Kick Timing

Kick Timing



Waveform 35

22.6 Interrupt Handling

22.6.1 Completion Interrupt

```
void rapids_irq_handler(void) {
    uint32_t irq_status = read_reg(RAPIDS_BASE + IRQ_STATUS);

    // Handle channel 0 completion
    if (irq_status & IRQ_CH0_COMPLETE) {
        // Read channel status
        uint32_t ch_status = read_reg(ch0_base + CH_STATUS);
        uint8_t desc_count = (ch_status >> 16) & 0xFF;

        // Process completed buffers
        process_rx_buffers(desc_count);

        // Clear interrupt
        write_reg(RAPIDS_BASE + IRQ_STATUS, IRQ_CH0_COMPLETE);

        // Re-kick channel for continuous operation (if desired)
        write_reg(ch0_base + CH_CTRL, CH_CTRL_ENABLE | CH_CTRL_KICK);
    }

    // Handle errors
    if (irq_status & IRQ_CH0_ERROR) {
        uint32_t err_status = read_reg(ch0_base + ERR_STATUS);
        handle_error(err_status);

        // Clear error interrupt
        write_reg(RAPIDS_BASE + IRQ_STATUS, IRQ_CH0_ERROR);
    }
}
```

22.7 Reset and Recovery

22.7.1 Per-Channel Reset

```
void reset_channel(int channel) {
    uint32_t ch_base = RAPIDS_BASE + 0x100 + (channel * 0x40);

    // Soft reset the channel
    write_reg(ch_base + CH_CTRL, CH_CTRL_SOFT_RESET);

    // Wait for reset complete
    while ((read_reg(ch_base + CH_STATUS) >> 28) == 0xF) {
        // State = RESET
    }

    // Clear error status
```

```
    write_reg(ch_base + ERR_STATUS, 0xFFFFFFFF);

    // Re-initialize descriptor pointer
    write_reg(ch_base + DESC_PTR_LO, saved_desc_ptr_lo[channel]);
    write_reg(ch_base + DESC_PTR_HI, saved_desc_ptr_hi[channel]);

    // Re-enable
    write_reg(ch_base + CH_CTRL, CH_CTRL_ENABLE);
}
```

22.7.2 Global Reset

```
void reset_rapids(void) {
    // Disable all interrupts
    write_reg(RAPIDS_BASE + IRQ_ENABLE, 0);

    // Global soft reset
    write_reg(RAPIDS_BASE + GLOBAL_CTRL, GLOBAL_CTRL_SOFT_RESET);

    // Wait for all channels idle
    while (read_reg(RAPIDS_BASE + GLOBAL_STATUS) & 0x00FF0000) {
        // Wait for active channels to clear
    }

    // Clear all interrupts
    write_reg(RAPIDS_BASE + IRQ_STATUS, 0xFFFFFFFF);

    // Re-run full initialization
    rapids_init();
}
```

23 Error Handling

23.1 Overview

RAPIDS Beats implements comprehensive error detection and reporting. This section describes error types, detection mechanisms, and recommended recovery procedures.

23.2 Error Categories

Error Hierarchy

Error Hierarchy

Source: 15_error_hierarchy.mmd

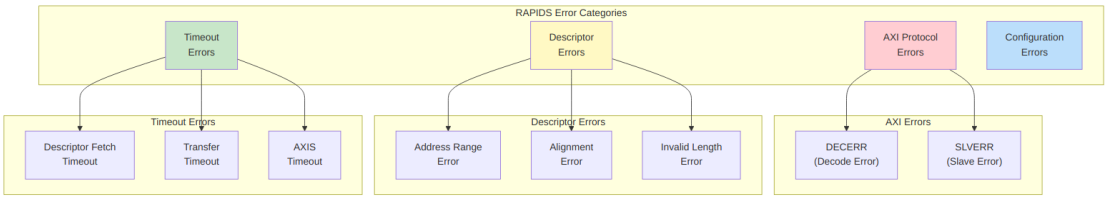


Diagram 28

23.3 Error Detection

23.3.1 AXI Response Errors

RAPIDS monitors all AXI transactions for error responses:

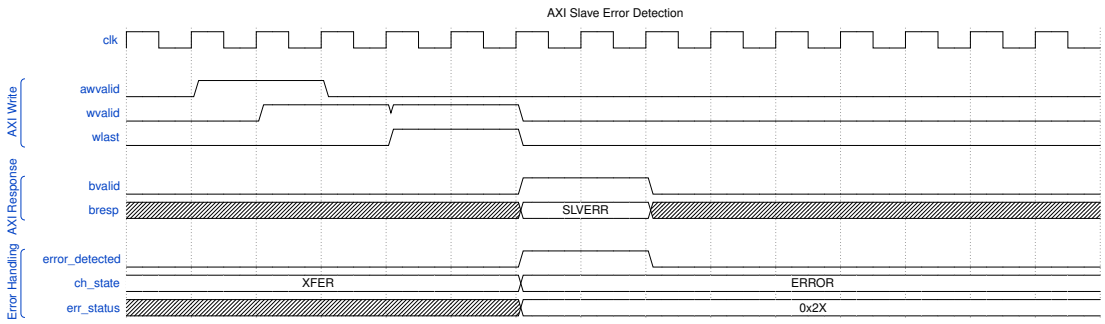
AXI Response Codes

Response	Code	Description	Typical Cause
OKAY	2'b00	Normal success	–
EXOKAY	2'b01	Exclusive OK	–
SLVERR	2'b10	Slave error	Memory protection, invalid access
DECERR	2'b11	Decode error	Address not mapped

AXI Error Detection

AXI Error Detection

Source: axi_error.json



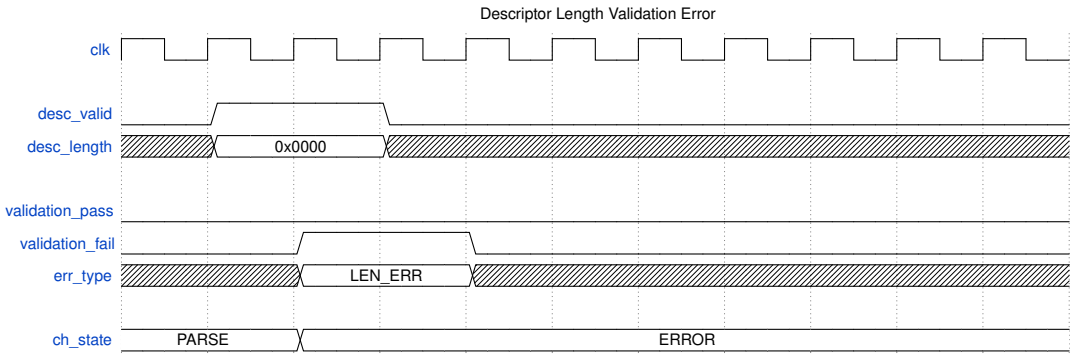
Waveform 36

23.3.2 Descriptor Validation Errors

Descriptors are validated when parsed:

Descriptor Validation Errors

Check	Error Condition	ERR_STATUS Code
Address range	Address outside configured range	0x10
Alignment	Address not properly aligned	0x11
Length	Length = 0 or > max allowed	0x12
Direction	Invalid direction code	0x13



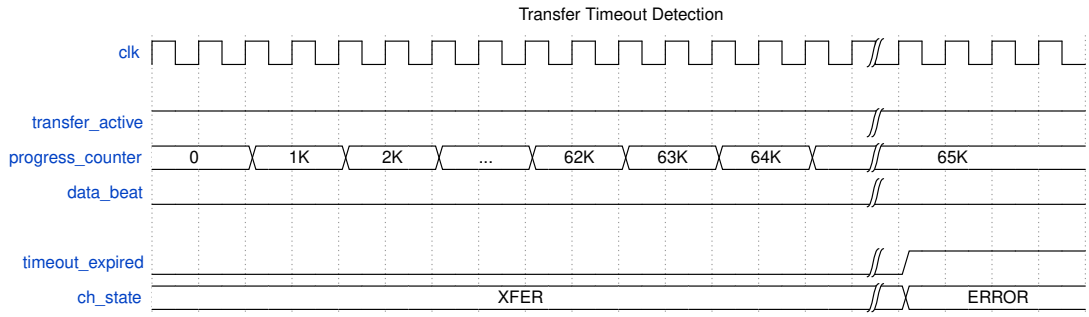
Waveform 37

23.3.3 Timeout Detection

RAPIDS implements watchdog timers for detecting stalled operations:

Timeout Thresholds

Timeout Type	Typical Value	Condition
Descriptor fetch	1024 cycles	No AXI response
Transfer	65536 cycles	No progress
AXIS receive	4096 cycles	TVALID stuck low
AXIS transmit	4096 cycles	TREADY stuck low



Waveform 38

23.4 Error Reporting

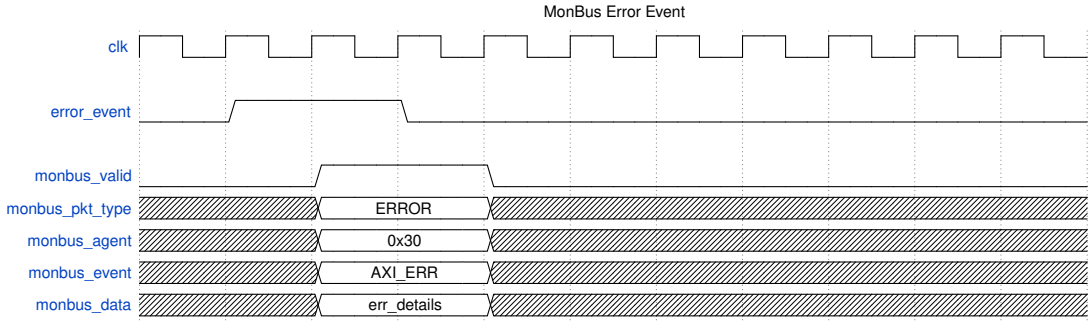
23.4.1 ERR_STATUS Register

When an error occurs, details are captured in the ERR_STATUS register:

```
ERR_STATUS [31:0]:
  [31:28] - AXI response code (BRESP or RRESP)
  [27:24] - Error type:
            0x0 = AXI write error
            0x1 = AXI read error
            0x2 = Descriptor fetch error
            0x3 = Descriptor validation error
            0x4 = Timeout error
  [23:16] - Descriptor index when error occurred
  [15:0]  - Reserved
```

23.4.2 MonBus Error Events

Errors generate MonBus packets for system-wide visibility:



Waveform 39

23.4.3 Interrupt Generation

Error interrupts are generated when enabled:

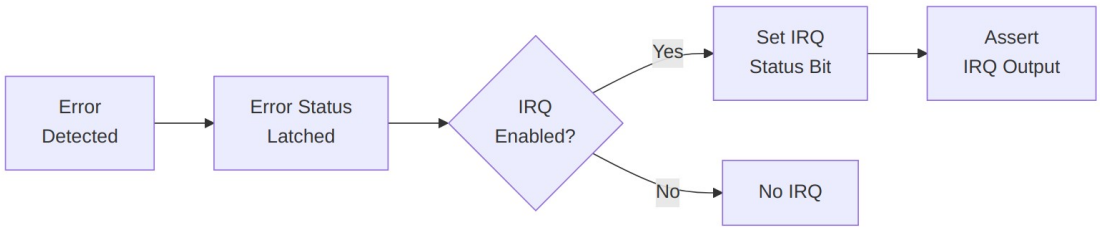


Diagram 29

23.5 Error Recovery

23.5.1 Recovery Flow

Error Recovery

Error Recovery

Source: 16_error_recovery.mmd

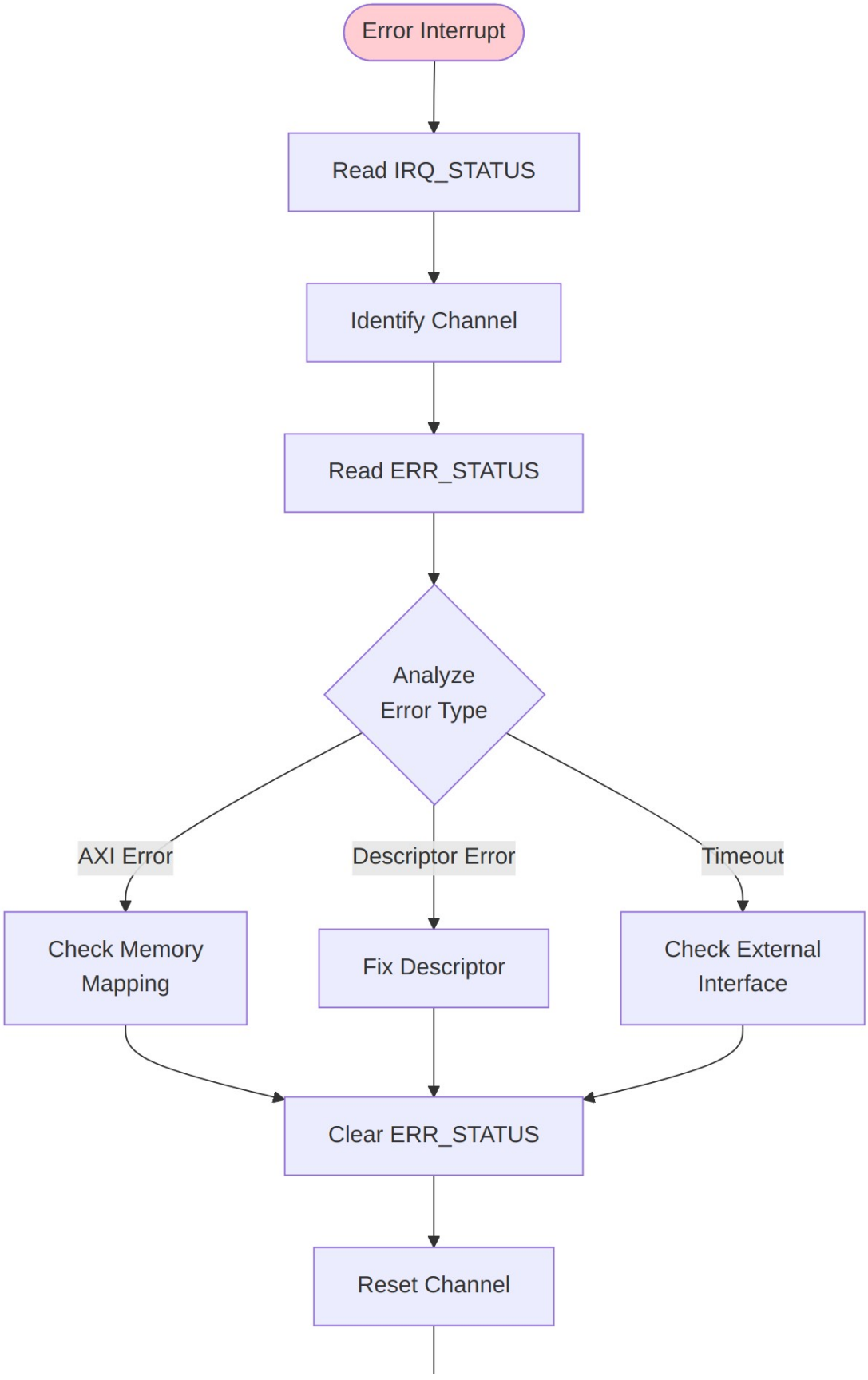

Error Interrupt
Read IRQ_STATUS
Identify Channel
Read ERR_STATUS
Analyze Error Type
AXI Error
Descriptor Error
Timeout
Check Memory Mapping
Fix Descriptor
Check External Interface
Clear ERR_STATUS
Reset Channel

Diagram 30

23.5.2 AXI Error Recovery

```
void handle_axi_error(int channel) {
    uint32_t ch_base = RAPIDS_BASE + 0x100 + (channel * 0x40);
    uint32_t err_status = read_reg(ch_base + ERR_STATUS);

    uint8_t axi_resp = (err_status >> 28) & 0xF;
    uint8_t err_type = (err_status >> 24) & 0xF;
    uint8_t desc_idx = (err_status >> 16) & 0xFF;

    if (axi_resp == 0x3) {  // DECERR
        printf("CH%d: Address decode error at descriptor %d\n",
channel, desc_idx);
        // Check memory mapping, descriptor addresses
    } else if (axi_resp == 0x2) {  // SLVERR
        printf("CH%d: Slave error at descriptor %d\n", channel,
desc_idx);
        // Check memory protection, peripheral status
    }

    // Clear error and reset channel
    write_reg(ch_base + ERR_STATUS, 0xFFFFFFFF);
    reset_channel(channel);
}
```

23.5.3 Descriptor Error Recovery

```
void handle_desc_error(int channel) {
    uint32_t ch_base = RAPIDS_BASE + 0x100 + (channel * 0x40);
    uint32_t err_status = read_reg(ch_base + ERR_STATUS);

    uint8_t err_type = (err_status >> 24) & 0xF;
    uint8_t desc_idx = (err_status >> 16) & 0xFF;

    switch (err_type & 0xF) {
        case 0x0:  // Address range
            printf("CH%d: Descriptor %d has invalid address\n",
channel, desc_idx);
            fix_descriptor_address(channel, desc_idx);
            break;
        case 0x1:  // Alignment
            printf("CH%d: Descriptor %d has alignment issue\n",
channel, desc_idx);
            fix_descriptor_alignment(channel, desc_idx);
            break;
        case 0x2:  // Length
            printf("CH%d: Descriptor %d has invalid length\n",
channel, desc_idx);
```

```
            fix_descriptor_length(channel, desc_idx);
            break;
    }

    // Clear and restart from fixed descriptor
    write_reg(ch_base + ERR_STATUS, 0xFFFFFFFF);
    write_reg(ch_base + DESC_PTR_LO, (uint32_t)&desc_ring[desc_idx]);
    write_reg(ch_base + DESC_PTR_HI, (uint32_t)
((uint64_t)&desc_ring[desc_idx] >> 32));
    reset_channel(channel);
}
```

23.5.4 Timeout Recovery

```
void handle_timeout(int channel) {
    uint32_t ch_base = RAPIDS_BASE + 0x100 + (channel * 0x40);
    uint32_t err_status = read_reg(ch_base + ERR_STATUS);
    uint32_t ch_status = read_reg(ch_base + CH_STATUS);

    uint8_t desc_idx = (err_status >> 16) & 0xFF;
    uint8_t sram_level = (ch_status >> 8) & 0xFF;

    printf("CH%d: Timeout at descriptor %d, SRAM level %d\n",
           channel, desc_idx, sram_level);

    // Check external interfaces
    if (sram_level == 0) {
        printf("  SRAM empty - check network input\n");
    } else if (sram_level > 0) {
        printf("  SRAM has data - check memory interface\n");
    }

    // Abort and reset
    write_reg(ch_base + CH_CTRL, CH_CTRL_ABORT);
    while ((read_reg(ch_base + CH_STATUS) >> 28) != 0x0) {
        // Wait for IDLE
    }

    write_reg(ch_base + ERR_STATUS, 0xFFFFFFFF);
    reset_channel(channel);
}
```

23.6 Error Prevention

23.6.1 Best Practices

Error Prevention Best Practices

Practice	Rationale
Validate descriptors before submission	Catch errors before hardware processing
Use aligned addresses	Avoid alignment errors
Set reasonable timeouts	Detect issues without false positives
Monitor SRAM levels	Identify flow control issues
Enable MonBus	System-wide error visibility

23.6.2 Descriptor Validation Example

```
int validate_descriptor(rapids_descriptor_t *desc) {
    // Check alignment
    if (desc->dest_addr & 0x3F) {  // 64-byte alignment
        return ERR_ALIGNMENT;
    }

    // Check length
    if (desc->length == 0 || desc->length > MAX_BEATS) {
        return ERR_LENGTH;
    }

    // Check direction
    if (desc->direction > DESC_DIR_SOURCE) {
        return ERR_DIRECTION;
    }

    // Check address range
    if (desc->dest_addr < VALID_ADDR_MIN || desc->dest_addr >
VALID_ADDR_MAX) {
        return ERR_ADDRESS;
    }

    return 0;  // Valid
}
```

24 RAPIDS Beats Architecture MAS

Version: 0.25 **Date:** 2025-01-10 **Purpose:** Module Architecture Specification for RAPIDS Phase 1 “Beats” Architecture

---

24.1 Document Organization

Note: All chapters linked below for automated document generation.

24.1.1 Front Matter

- Document Information

24.1.2 Chapter 1: Overview

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

24.1.3 Chapter 2: FUB (Functional Unit Blocks)

Control Path: - Scheduler - Descriptor Engine

AXI Engines: - AXI Read Engine - AXI Write Engine

Flow Control: - Beats Alloc Control - Beats Drain Control - Beats Latency Bridge

24.1.4 Chapter 3: Macro (Integration Blocks)

Scheduler Integration: - Beats Scheduler Group - Beats Scheduler Group Array

Sink Data Path (Network to Memory): - Sink Data Path - Sink Data Path AXIS - Sink SRAM Controller - Sink SRAM Controller Unit

Source Data Path (Memory to Network): - Source Data Path - Source Data Path AXIS - Source SRAM Controller - Source SRAM Controller Unit

Top-Level Integration: - RAPIDS Core Beats

24.1.5 Chapter 4: Interfaces

- Interface Overview
- AXI4 Interface Specification
- AXIS Interface Specification
- MonBus Interface Specification

24.2 Quick Reference

24.2.1 FUB Modules (fub_beats/)

FUB Module Summary

Module	File	Purpose	Status
scheduler	`scheduler.sv`	Transfer coordinator, descriptor processing	Implemented
descriptor_engine	`descriptor_engine.sv`	Descriptor fetch/parse (256-bit)	Implemented
axi_read_engine	`axi_read_engine.sv`	AXI read master, streaming pipeline	Implemented
axi_write_engine	`axi_write_engine.sv`	AXI write master, streaming pipeline	Implemented
beats_alloc_ctrl	`beats_alloc_ctrl.sv`	Space allocation tracking (virtual FIFO)	Implemented
beats_drain_ctrl	`beats_drain_ctrl.sv`	Data availability tracking (virtual FIFO)	Implemented
beats_latency_bridge	`beats_latency_bridge.sv`	Latency compensation, backpressure management	Implemented

24.2.2 Macro Modules (macro_beats/)

Macro Module Summary

Module	File	Purpose	Status
beats_scheduler_group	`beats_scheduler_group.sv`	Scheduler + Descriptor Engine wrapper	Implemented
beats_scheduler_group_array	`beats_scheduler_group_array.sv`	8-channel scheduler array with arbitration	Implemented
sink_data_path	`sink_data_path.sv`	Network-to-memory path integration	Implemented

Module	File	Purpose	Status
sink_data_path_axis	`sink_data_path_axis.sv`	AXIS variant of sink path	Implemented
source_data_path	`source_data_path.sv`	Memory-to-network path integration	Implemented
source_data_path_axis	`source_data_path_axis.sv`	AXIS variant of source path	Implemented
snk_sram_controller	`snk_sram_controller.sv`	Sink SRAM buffer management	Implemented
snk_sram_controller_unit	`snk_sram_controller_unit.sv`	Per-channel sink SRAM unit	Implemented
src_sram_controller	`src_sram_controller.sv`	Source SRAM buffer management	Implemented
src_sram_controller_unit	`src_sram_controller_unit.sv`	Per-channel source SRAM unit	Implemented
rapids_core_beats	`rapids_core_beats.sv`	Top-level RAPIDS integration	Implemented

24.3 Architecture Comparison: RAPIDS vs STREAM

The “beats” architecture is a Phase 1 implementation of RAPIDS that shares concepts with STREAM but targets network-to-memory (and vice versa) transfers rather than memory-to-memory.

RAPIDS Beats vs STREAM Comparison

Feature	STREAM	RAPIDS Beats
Primary Use	Memory-to-memory DMA	Network-to-memory accelerator
Data Paths	Single bidirectional	Separate sink/source paths
Network Interface	None	AXIS master/slave
SRAM Buffering	Shared buffer	Separate sink/source buffers
Descriptor Format	256-bit	256-bit (compatible)
Channel Count	8	8
Flow Control	alloc_ctrl/	beats_alloc_ctrl/beats_drain_ctrl

Feature	STREAM	RAPIDS Beats
	drain_ctrl	

24.4 Clock and Reset Summary

24.4.1 Clock Domains

Clock Domains

Clock	Frequency	Usage
`aclk`	100-500 MHz	Primary - all RAPIDS logic, AXI/AXIS interfaces

24.4.2 Reset Signals

Reset Signals

Reset	Polarity	Type	Usage
`aresetn`	Active-low	Async assert, sync deassert	Primary - all RAPIDS logic

See: Clocks and Reset for complete timing specifications

24.5 Interface Summary

24.5.1 External Interfaces

External Interfaces

Interface	Type	Width	Purpose
AXI4 (Descriptor)	Master	256-bit	Descriptor fetch
AXI4 (Sink Write)	Master	512-bit (param)	Sink data write to memory
AXI4 (Source Read)	Master	512-bit (param)	Source data read from memory
AXIS (Sink)	Slave	512-bit (param)	Network data ingress
AXIS (Source)	Master	512-bit (param)	Network data egress
MonBus	Master	64-bit	Monitor packet

Interface	Type	Width	Purpose
			output

24.5.2 Internal Buses

Internal Buses

Interface	Width	Purpose
MonBus	64-bit	Internal monitoring bus
Descriptor Bus	256-bit	Descriptor distribution
SRAM Data Bus	512-bit	SRAM read/write data

24.6 Related Documentation

- **PRD.md** - Product requirements and overview
- **CLAUDE.md** - AI development guide
- **RAPIDS Spec** - Original RAPIDS specification

24.7 Specification Conventions

24.7.1 Signal Naming

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

24.7.2 Parameter Naming

- **Uppercase:** `NUM_CHANNELS`, `DATA_WIDTH`, `ADDR_WIDTH`
- **Derived:** `CHAN_WIDTH = $clog2(NUM_CHANNELS)`

24.7.3 Figure and Table Numbering

- **Figures:** `Figure X.Y.Z: Title` where X=chapter, Y=section, Z=figure number
- **Tables:** Caption line `: Table Title` after table markdown

Last Updated: 2025-01-10 **Maintained By:** RAPIDS Architecture Team

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

---

25 Product Requirements Document (PRD)

25.1 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`

25.2 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.

25.2.1 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

25.2.2 1.2 Subsystem Goals

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

---

25.3 2. Documentation Structure

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

25.3.1 📚 Complete RAPIDS Specification

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

- **Index** - Complete specification structure

25.3.1.1 Chapter 1: Overview

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

25.3.1.2 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

25.3.1.3 Chapter 3: Interfaces

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

25.3.1.4 Chapter 4 & 5: Programming

- Programming Model
- Register Definitions

25.3.2 🐛 Known Issues

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

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

25.3.3 📖 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

25.4 2.4 Organizational Standards - RAPIDS Code Location

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

25.4.1 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.

25.4.2 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
```

25.4.3 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

25.4.4 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!
```

25.4.5 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

25.4.6 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.

25.5 3. Architecture Overview

25.5.1 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

25.5.2 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

---

25.6 4. Key Features

25.6.1 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		

25.6.2 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

25.6.3 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

25.6.4 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

25.7 5. Interfaces

25.7.1 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

25.7.2 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_*       (...)
);
```

25.8 6. Use Cases

25.8.1 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

25.8.2 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

25.8.3 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

---

25.9 7. Test Coverage

25.9.1 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

25.9.2 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

---

25.10 8. Known Issues Summary

25.10.1 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

25.10.2 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

25.11 9. Integration Guidelines

25.11.1 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)
);
```

25.11.2 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

25.12 10. Development Status

25.12.1 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

25.12.2 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

---

25.13 11. Performance Characteristics

25.13.1 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

25.13.2 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

25.13.3 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

---

25.14 12. Verification Infrastructure

25.14.1 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
```

25.14.2 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)

25.14.3 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

25.14.4 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

---

25.15 13. Quick Reference

25.15.1 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

25.15.2 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
```

25.16 14. Success Criteria

25.16.1 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

25.16.2 14.2 Quality

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

- ✓ All FSMs documented
- ⏳ Integration guide complete

25.16.3 14.3 Documentation

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

25.17 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

---

25.18 15. Attribution and Contribution Guidelines

25.18.1 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.

25.19 16. Documentation Generation

25.19.1 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

25.20 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

25.21 17. References

25.21.1 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`

25.21.2 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/`

25.21.3 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

---

25.22 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/`

26 Address Increment Patterns for DMA and Descriptor-Based Data Movement

Document Version: 1.0 **Last Updated:** 2026-01-13 **Scope:** Comprehensive analysis of address increment patterns for descriptor-based systems **Reference Implementation:** RAPIDS (Rapid AXI Programmable In-band Descriptor System)

26.1 Table of Contents

26.2 1. Executive Summary

Address increment patterns determine how DMA engines and descriptor-based data movement systems calculate memory addresses during transfers. The choice of pattern significantly impacts:

- **Bandwidth efficiency**: Alignment and burst optimization
- **Application fit**: Different workloads require different patterns
- **Hardware complexity**: More patterns = more control logic
- **Software flexibility**: Richer patterns = simpler programming model

26.2.1 Pattern Categories

Category	Description	Use Cases
Linear/Contiguous	Sequential address increment	Block copy, streaming data
2D Stride	Row-major with inter-row gaps	Image processing, video frames
3D Volume	Multi-plane with inter-plane gaps	3D imaging, tensor operations
Scatter-Gather	Non-contiguous fragment list	Protocol buffers, fragmented memory

Category	Description	Use Cases
Circular	Wrap-around at boundary	Audio streaming, network buffers
Tensor	Multi-dimensional with strides	Neural network accelerators

26.3 2. RAPIDS Current Implementation

26.3.1 2.1 Descriptor Format (256-bit)

RAPIDS uses a fixed 256-bit descriptor structure defined in `rapids_pkg.sv`:

```
Bit Field           Width   Description
──────────────────────────────────────────────────────────────
[63:0]              64-bit  src_addr - Source address (byte-aligned)
[127:64]            64-bit  dst_addr - Destination address (byte-
aligned)
[159:128]           32-bit  length - Transfer length in BEATS (not
bytes)
[191:160]           32-bit  next_descriptor_ptr - Next descriptor
address
[207:200]            8-bit  desc_priority - Transfer priority
[199:196]            4-bit  channel_id - Channel ID
[195]                1-bit  error - Error flag
[194]                1-bit  last - Last descriptor in chain
[193]                1-bit  gen_irq - Generate interrupt
[192]                1-bit  valid - Valid descriptor
[255:208]           48-bit  reserved - Future use
```

26.3.2 2.2 Current Address Increment Modes

Mode 1: Beat-Based Linear (Default)

```
Address Calculation:
  next_addr = current_addr + (beat_count × 64 bytes)

  Where: RAPIDS_BYTES_PER_BEAT = 64 bytes (512-bit data width)
```

- All transfers aligned to 64-byte boundaries for optimal throughput
- Descriptor length field specifies count in beats
- Software converts byte count to beats: `beats = (bytes + 63) >> 6`

Mode 2: Chunk-Based Alignment (Sub-Beat Transfers)

For alignment/final phases handling partial beats:

```
Address Calculation for Alignment Phase:
  offset = address[5:0]                    // 6-bit offset (0-63)
  bytes_to_boundary = (offset == 0) ? 0 : 64 - offset

  Transfer Phases:
  Phase 1 (Alignment): Transfer bytes_to_boundary to reach 64B
boundary
  Phase 2 (Streaming): Transfer full 64B beats
  Phase 3 (Final):     Transfer remaining < 64 bytes
```

26.3.3 2.3 Alignment Information Structure

```
typedef struct packed {
    logic [15:0] first_chunk_enables;    // Phase 1 chunk mask
    logic [15:0] streaming_chunk_enables; // Phase 2 (always 0xFFFF)
    logic [15:0] final_chunk_enables;     // Phase 3 chunk mask
    logic [31:0] first_transfer_bytes;    // Phase 1 size
    logic [31:0] streaming_bytes;         // Phase 2 size
    logic [31:0] final_transfer_bytes;    // Phase 3 size
    logic [3:0]  first_start_chunk;       // Starting chunk (0-15)
    logic [3:0]  first_valid_chunks;      // Chunks in phase 1
    logic [3:0]  final_valid_chunks;      // Chunks in phase 3
} alignment_info_t;
```

26.3.4 2.4 AXI Translation

Transfer Phase	AxSIZE	AxBURST	Address Increment
Alignment	3'b010	INCR	4 bytes × (AxLEN+1)
Streaming	3'b110	INCR	64 bytes × (AxLEN+1)
Final	3'b010	INCR	4 bytes × (AxLEN+1)

26.4 3. AXI Protocol Burst Modes

The AXI protocol defines three burst types that constrain address generation:

26.4.1 3.1 FIXED Burst (AxBURST = 2'b00)

```
Address Formula:
  beat[n].addr = AxADDR   (constant for all beats)
```

Use Cases: - FIFO access (single address register) - Memory-mapped I/O ports - Audio codec sample registers

Constraints: - All beats transfer to same address - Cannot cross 4KB boundary (trivially satisfied)

26.4.2 3.2 INCREMENT Burst (AxBURST = 2'b01)

```
Address Formula:
  beat[n].addr = AxADDR + n × (1 << AxSIZE)
```

```
  Example (AxSIZE=6, 64-byte):
    beat[0] = 0x1000
    beat[1] = 0x1040
    beat[2] = 0x1080
    ...
```

Use Cases: - Linear memory access - Block copy operations - Stream data to/from memory

Constraints: - Cannot cross 4KB boundary - Maximum burst length varies by protocol version

26.4.3 3.3 WRAP Burst (AxBURST = 2'b10)

```
Address Formula:
  wrap_boundary = AxADDR & ~((AxLEN+1) × (1<<AxSIZE) - 1)
  wrap_size = (AxLEN+1) × (1 << AxSIZE)

  beat[n].addr = wrap_boundary + ((AxADDR + n×(1<<AxSIZE)) mod
wrap_size)

  Example (AxADDR=0x1020, AxLEN=3, AxSIZE=4):
    Wrap boundary: 0x1000
    Wrap size: 64 bytes (4 × 16)
    beat[0] = 0x1020
    beat[1] = 0x1030
    beat[2] = 0x1000  (wrapped)
    beat[3] = 0x1010
```

Use Cases: - Cache line fills (critical-word-first) - Circular buffer access

Constraints: - AxLEN must be 1, 3, 7, or 15 - Starting address must be aligned to transfer size

---

26.5 4. Linear Address Increment Patterns

26.5.1 4.1 Simple Contiguous

The most basic pattern - sequential addresses:

```
Descriptor Fields:
  src_addr:  Starting source address
  dst_addr:  Starting destination address
  length:    Total transfer size

Address Calculation:
```

```
  for each beat b in [0, length/beat_size):
    src = src_addr + b × beat_size
    dst = dst_addr + b × beat_size
```

RAPIDS Implementation: Native support via beat-based transfers.

26.5.2 4.2 Unaligned Start/End

Handling transfers that don't start or end on natural boundaries:

```
Example: Transfer 200 bytes from 0x1040
  Phase 1 (Align):  Transfer 64 bytes (0x1040-0x107F) to reach 0x1080
  Phase 2 (Stream): Transfer 128 bytes (0x1080-0x10FF) in 2 full beats
  Phase 3 (Final):  Transfer 8 bytes (0x1100-0x1107)
```

RAPIDS Implementation: Native support via alignment_info_t structure.

26.5.3 4.3 Size-Constrained Bursts

Breaking large transfers into AXI-compliant bursts:

```
Constraints:
  - AXI4: Max 256 beats per burst
  - No 4KB boundary crossing

Address Calculation:
  remaining = total_length
  current_addr = start_addr

  while (remaining > 0):
    burst_len = min(remaining, 256×beat_size)
    burst_len = min(burst_len, next_4kb_boundary - current_addr)
    issue_burst(current_addr, burst_len)
    current_addr += burst_len
    remaining -= burst_len
```

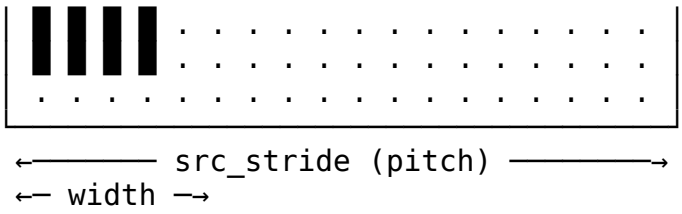
RAPIDS Implementation: Handled by AXI engine burst splitting.

26.6 5. 2D Transfer Patterns (Stride-Based)

26.6.1 5.1 Concept

2D transfers move rectangular regions where source and destination may have different memory layouts (pitches/strides).

```
Memory Layout:

  ┌──────────────────────────────────┐
  ████. . . . . . . . . . . . . . . .  Row 0: Transfer 4 elements
```



26.6.2 5.2 Descriptor Fields (Extended)

```
2D Descriptor Format:
  src_addr:    Starting source address (top-left corner)
  dst_addr:    Starting destination address
  x_length:    Bytes per row (width of transfer region)
  y_length:    Number of rows
  src_stride:  Source bytes between row starts
  dst_stride:  Destination bytes between row starts
```

26.6.3 5.3 Address Calculation

```
for row in [0, y_length):
  row_src = src_addr + row × src_stride
  row_dst = dst_addr + row × dst_stride
  transfer(row_src, row_dst, x_length)
```

26.6.4 5.4 Use Cases

Image Processing:

```
Example: Extract 640×480 ROI from 1920×1080 frame (32-bit pixels)
  src_addr:   frame_base + (y_offset × 1920 + x_offset) × 4
  dst_addr:   roi_buffer
  x_length:   640 × 4 = 2560 bytes
  y_length:   480
  src_stride: 1920 × 4 = 7680 bytes
  dst_stride: 640 × 4 = 2560 bytes (packed destination)
```

Video Frame Tiling:

```
Example: Copy 16×16 macroblock
  x_length:   16 × bytes_per_pixel
  y_length:   16
  src_stride: frame_width × bytes_per_pixel
  dst_stride: 16 × bytes_per_pixel
```

26.6.5 5.5 Analog Devices AXI DMAC Implementation

Reference implementation from ADI's high-speed DMA controller:

```
Registers:
  X_LENGTH  (0x418): Row width - 1
  Y_LENGTH  (0x41C): Row count - 1
  SRC_STRIDE (0x424): Source row spacing
  DEST_STRIDE (0x420): Destination row spacing
```

```
Address Formulas:
  ROW_SRC_ADDRESS = SRC_ADDRESS + SRC_STRIDE × N
  ROW_DEST_ADDRESS = DEST_ADDRESS + DEST_STRIDE × N
```

26.7 6. 3D Transfer Patterns

26.7.1 6.1 Concept

3D transfers extend 2D with an additional depth/plane dimension:

```
Memory Layout (Z planes stacked):
  Plane 0:    Plane 1:    Plane 2:
```

26.7.2 6.2 Descriptor Fields (Extended)

```
3D Descriptor Format:
  src_addr:     Starting address (origin corner)
  dst_addr:     Destination address
  x_length:     Width in bytes
  y_length:     Height in rows
  z_length:     Depth in planes
  src_stride_x: Source row stride (unused, implicit x_length)
  src_stride_y: Source row-to-row stride
  src_stride_z: Source plane-to-plane stride
  dst_stride_y: Destination row stride
  dst_stride_z: Destination plane stride
```

26.7.3 6.3 Address Calculation

```
for plane in [0, z_length):
  for row in [0, y_length):
    src = src_addr + plane×src_stride_z + row×src_stride_y
    dst = dst_addr + plane×dst_stride_z + row×dst_stride_y
    transfer(src, dst, x_length)
```

26.7.4 6.4 Use Cases

3D Medical Imaging:

```
Example: Extract 64×64×64 VOI from 512×512×256 CT volume
  src_addr:     volume_base + z_off×512×512 + y_off×512 + x_off
  x_length:     64 bytes
  y_length:     64
  z_length:     64
```

```
  src_stride_y: 512
  src_stride_z: 512 × 512
```

Neural Network Tensors:

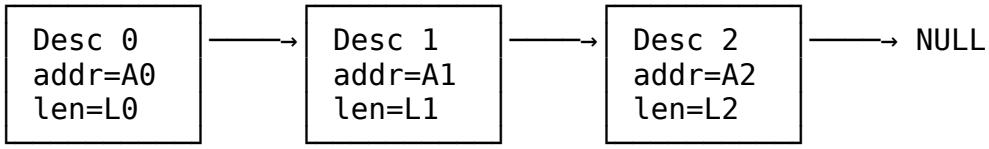
```
Example: Move 4D tensor slice [batch, channel, H, W]
  Requires nested 3D operations or generalized tensor addressing
```

26.8 7. Scatter-Gather Patterns

26.8.1 7.1 Concept

Scatter-Gather (SG) handles non-contiguous memory regions using linked lists of descriptors:

```
Descriptor Chain:
```



26.8.2 7.2 SG Descriptor Fields

```
Scatter-Gather Descriptor:
  buffer_addr:      Address of this fragment
  buffer_length:    Size of this fragment
  next_desc_ptr:    Address of next descriptor (0 = last)
  control_flags:    Completion interrupt, error handling, etc.
  status:           Completion status (written by DMA)
```

26.8.3 7.3 Operation Modes

Gather (Multiple Source -> Single Destination):

```
Source fragments: [A0:L0], [A1:L1], [A2:L2]
Destination: Contiguous buffer B

Result: B = concat(mem[A0:A0+L0], mem[A1:A1+L1], mem[A2:A2+L2])
```

Scatter (Single Source -> Multiple Destinations):

```
Source: Contiguous buffer B of length L0+L1+L2
Destination fragments: [A0:L0], [A1:L1], [A2:L2]

Result: mem[A0:A0+L0] = B[0:L0]
        mem[A1:A1+L1] = B[L0:L0+L1]
        mem[A2:A2+L2] = B[L0+L1:end]
```

26.8.4 7.4 AMD/Xilinx AXI DMA SG Descriptor Format

```
Standard Descriptor (32-bit addressing):
  Offset  Field            Description
  ──────────────────────────────────────────────────
  0x00    NXTDESC          Next descriptor pointer
  0x04    Reserved
  0x08    BUFFER_ADDRESS   Data buffer pointer
  0x0C    Reserved
  0x10    Reserved
  0x14    Reserved
  0x18    CONTROL          Transfer length, SOF, EOF
  0x1C    STATUS           Completion status
```

26.8.5 7.5 Use Cases

Network Packet Assembly:

```
Example: Assemble TCP packet from scattered headers and payload
  Desc 0: Ethernet header (14 bytes at addr_eth)
  Desc 1: IP header (20 bytes at addr_ip)
  Desc 2: TCP header (20 bytes at addr_tcp)
  Desc 3: Payload (1460 bytes at addr_payload)
```

RAPIDS Implementation: Native support via next_descriptor_ptr chaining.

26.9 8. Circular and Ring Buffer Patterns

26.9.1 8.1 Concept

Circular buffers wrap addresses at a boundary, enabling continuous streaming without descriptor reload:

```
Memory Layout:
   ┌─────────────────────────────────────────┐
   │ 7 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | (wraps)
   └─────────────────────────────────────────┘
     ↑                   ↑
   write               read
```

26.9.2 8.2 Descriptor Fields (Extended)

```
Circular Buffer Descriptor:
  base_addr:    Buffer base address
  buffer_size:  Total buffer size (must be power of 2 for efficient
wrap)
  current_ptr:  Current read/write position
  wrap_enable:  Enable address wrapping
```

26.9.3 8.3 Address Calculation

```
Efficient wrap (power-of-2 size):
  next_addr = base_addr + ((current_ptr + increment) & (buffer_size -
1))

General wrap:
  offset = (current_ptr + increment) % buffer_size
  next_addr = base_addr + offset
```

26.9.4 8.4 Ping-Pong Double Buffering

Common implementation using two descriptors pointing to each other:

```
Descriptor A:                Descriptor B:
  buffer_addr = BUF_A          buffer_addr = BUF_B
  next_desc = &Desc_B          next_desc = &Desc_A

Operation:
  While DMA processes BUF_A, CPU fills BUF_B
  On completion, DMA auto-loads Desc_B, starts BUF_B
  CPU processes BUF_A while DMA runs BUF_B
  (repeat)
```

26.9.5 8.5 Use Cases

Audio Streaming:

```
Example: 48kHz stereo audio, 20ms buffers
  buffer_size: 48000 × 2 channels × 2 bytes × 0.02s = 3840 bytes
  Ring with 4 buffers: 15360 bytes total
  Wrap at: base + 15360
```

Network RX Ring:

```
Example: Ethernet receive ring
  descriptor_count: 256 (power of 2)
  Each descriptor points to packet buffer
  Last descriptor.next points to first descriptor
  Hardware auto-advances through ring
```

26.10 9. Tensor Accelerator Memory Access Patterns (PRIORITY 0)

Tensor accelerators for neural network inference represent the most demanding and sophisticated address generation requirements. This section provides comprehensive coverage of industry-standard approaches.

26.10.1 9.1 Systolic Array Dataflow Fundamentals

Systolic arrays (used in Google TPU, NVIDIA Tensor Cores, etc.) require coordinated data movement across three matrices: inputs (activations), weights, and outputs (partial sums).

Three Primary Dataflows:

Dataflow	Stationary Data	Moving Data	Best For
Weight Stationary (WS)	Weights preloaded in PEs	Inputs + partial sums flow	Weight reuse (many inputs)
Input Stationary (IS)	Inputs held in PEs	Weights + partial sums flow	Input reuse (many kernels)
Output Stationary (OS)	Partial sums accumulate in PEs	Inputs + weights flow	Output accumulation

Address Pattern Implications:

```
Weight Stationary:
  - Weights: Load once per layer, stream through array
  - Inputs: Stream row-by-row, reuse across weight tiles
  - Address pattern: 2D tiled with weight-tile-aligned boundaries

Output Stationary:
  - Outputs: Accumulate in place
  - Inputs/Weights: Double-buffered streaming
  - Address pattern: Output-tile-centric addressing
```

26.10.2 9.2 Data Cube Model (NVDLA Reference)

NVIDIA's Deep Learning Accelerator (NVDLA) defines tensor addressing using a "data cube" model with three key stride parameters:

Data Cube Structure:

```
Dimensions: Width (W) × Height (H) × Channels (C)

Memory Layout:
  - Data divided into 1×1×32byte "atom cubes"
  - Scanning order: C'(32B) → W → H → C (surfaces)
  - C' changes fastest (innermost loop)
```

```
Address Calculation:
  element_addr = base_addr
                 + surface_index × surface_stride
                 + line_index × line_stride
                 + element_offset_within_line
```

Stride Parameters:

Parameter	Description	Alignment
`line_stride`	Bytes from line N to line N+1	32 bytes
`surface_stride`	Bytes from surface N to surface N+1	32 bytes
`base_addr`	Starting address	32 bytes

NVDLA Register Configuration:

```
D_DATAIN_FORMAT     : Data format selection
D_DATAIN_SIZE_0     : Width × Height
D_DATAIN_SIZE_1     : Channels
D_LINE_STRIDE       : Line-to-line spacing (0x5040)
D_SURF_STRIDE       : Surface-to-surface spacing

Example:
  Width=26, Height=32, Channels=3
  line_stride = 0x1A0 (aligned width)
  surface_stride = 0x1520 (line_stride × height)
```

26.10.3 9.3 Tensor Memory Accelerator (TMA) - NVIDIA Hopper

NVIDIA's Hopper architecture introduces hardware TMA for efficient multi-dimensional tensor transfers:

TMA Descriptor Format:

```
CUtensorMap structure (64-bit packed descriptor):
  - Global memory base address
  - Tensor rank (1D to 5D supported)
  - Dimension sizes and strides
  - Swizzle pattern for bank conflict avoidance
  - Box dimensions (tile size for transfer)
```

Key Constraints:

```
Alignment Requirements:
  - Contiguous dimension must have stride=1
  - All other strides must be multiples of 16 bytes
  - For float tensors [M,N] with stride [N,1]: N % 4 == 0
```

```
Tile Quantization:
  - TPU/GPU systolic arrays process 128×128 or 128×8 tiles
  - Tensors must align to tile boundaries to avoid padding waste
  - Performance penalty for partial tiles (idle MACs)
```

Address Generation (Hardware):

```
TMA eliminates per-thread address calculation:
  Traditional: Each thread computes addr = base + thread_id × stride
  TMA: Single descriptor, hardware generates all addresses

Coordinate-based access:
  ArithTuple(row, col) → Hardware translates to physical address
  Automatic out-of-bounds predication (no manual boundary checks)
```

26.10.4 9.4 Convolution Address Patterns

26.10.4.1 9.4.1 Direct Convolution

```
Input Feature Map:  [N, C_in, H_in, W_in]
Weight Kernel:      [C_out, C_in, K_h, K_w]
Output Feature Map: [N, C_out, H_out, W_out]

Output position (n, c_out, h_out, w_out):
  For each (c_in, k_h, k_w):
    input_addr = base_in + n×(C_in×H_in×W_in)
                         + c_in×(H_in×W_in)
                         + (h_out×stride_h + k_h)×W_in
                         + (w_out×stride_w + k_w)

    weight_addr = base_w + c_out×(C_in×K_h×K_w)
                         + c_in×(K_h×K_w)
                         + k_h×K_w
                         + k_w
```

26.10.4.2 9.4.2 Im2Col Transformation

Converts convolution to GEMM for systolic array efficiency:

```
Original Convolution:
  Output[h,w] = sum(Input[h+kh, w+kw] × Weight[kh, kw])

Im2Col Transformation:
  - Unroll each receptive field into a column
  - Weight kernels become rows
  - Convolution becomes matrix multiplication

Address Pattern for Im2Col:
  For output position (h_out, w_out):
```

```
    col_index = h_out × W_out + w_out
    For kernel position (c_in, k_h, k_w):
      row_index = c_in × K_h × K_w + k_h × K_w + k_w
      src_addr = base + c_in×(H×W) + (h_out×s_h+k_h)×W +
(w_out×s_w+k_w)
      dst_addr = im2col_base + row_index × num_output_positions +
col_index

Data Expansion Factor:
  Naive: K_h × K_w × data replication
  Hardware Im2Col: Generate addresses on-the-fly, no physical
expansion
```

26.10.4.3 9.4.3 Strided and Dilated Convolution

```
Strided Convolution (stride > 1):
  input_h = output_h × stride + kernel_h
  input_w = output_w × stride + kernel_w
  Reduces output spatial dimensions

Dilated/Atrous Convolution:
  input_h = output_h + kernel_h × dilation
  input_w = output_w + kernel_w × dilation
  Expands receptive field without increasing parameters

Address Modification:
  base_addr + (h × stride + kh × dilation) × line_stride
            + (w × stride + kw × dilation) × element_size
```

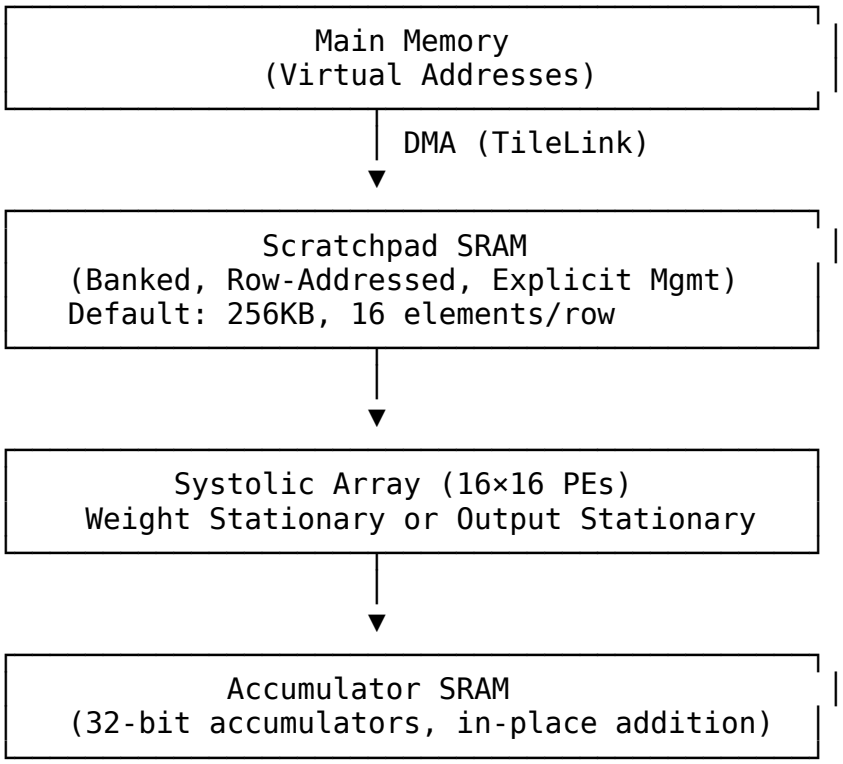
26.10.4.4 9.4.4 Depthwise Separable Convolution

```
Standard Convolution: C_in × C_out × K × K multiplications
Depthwise Separable: C_in × K × K + C_in × C_out multiplications

Phase 1 - Depthwise (per-channel):
  For each channel c:
    output[c, h, w] = conv2d(input[c], kernel[c])

  Address Pattern:
    Each channel processes independently
    No cross-channel data movement
    input_addr = base + c × H × W + h × W + w

Phase 2 - Pointwise (1×1 convolution):
  Standard GEMM across channels
  input_addr = base + c × (H × W) + position
```

26.10.5 9.5 Gemmini Accelerator Architecture (UC Berkeley Reference)

Open-source systolic array generator with well-documented DMA and addressing:

Memory Hierarchy:



Scratchpad Address Format (32-bit):

```
Bit 31:    0=Scratchpad, 1=Accumulator
Bit 30:    Accumulator mode (0=overwrite, 1=accumulate)
Bit 29:    Read format (0=scaled inputType, 1=raw accType)
Bits 28:0: Row address within selected memory

Row Width:
  Scratchpad: DIM × inputType bits (default: 16 × 8 = 128 bits)
  Accumulator: DIM × accType bits (default: 16 × 32 = 512 bits)
```

DMA Operations:

```
// mvin: Move data from DRAM to scratchpad
// Configuration registers set stride
mvin(
    dram_addr,          // Virtual address (64-bit)
    sp_addr,            // Scratchpad row address
    rows,               // Number of rows to transfer
```

```
    cols                // Elements per row (≤ DIM)
);

// mvout: Move data from accumulator to DRAM
// Supports integrated max-pooling
mvout(
    dram_addr,          // Destination address
    acc_addr,           // Accumulator row address
    rows, cols,
    pool_size,          // Optional pooling window
    pool_stride
);
```

Stride Configuration:

```
config_mvin:
  stride[0]: Source memory stride for mvin_A
  stride[1]: Source memory stride for mvin_B
  stride[2]: Source memory stride for mvin_D

config_mvout:
  stride: Destination memory stride
  pool_params: Pooling window and stride
```

26.10.6 9.6 Tiling for Large Matrices

When matrices exceed on-chip memory, tiling is essential:

Hierarchical Tiling (CUTLASS Model):

```
Level 1 - Thread Block Tile:
  Load tile from global memory to shared memory
  Tile size: M_tile × K_tile (A), K_tile × N_tile (B)

Level 2 - Warp Tile:
  Load from shared memory to registers
  Tile size: Warp_M × Warp_K (A), Warp_K × Warp_N (B)

Level 3 - Thread Tile:
  Individual thread's computation
  Register-to-register operations

Address Generation per Level:
  Global: base + block_id × block_tile_size
  Shared: smem_base + warp_id × warp_tile_size
  Register: Direct indexing within tile
```

Double-Buffering for Latency Hiding:

```
Buffer Structure:
  tile_buffer[2][TILE_ROWS][TILE_COLS]

Pipeline:
  Cycle N:   Compute on buffer[0], Load to buffer[1]
  Cycle N+1: Compute on buffer[1], Load to buffer[0]

Address Pattern:
  compute_addr = base + (cycle % 2) × tile_size + element_offset
  load_addr    = base + ((cycle + 1) % 2) × tile_size + element_offset
```

26.10.7 9.7 Sparse Tensor Patterns

Sparse accelerators require index-based indirect addressing:

Compressed Sparse Row (CSR) Format:

```
Dense: [a 0 b]    CSR: values = [a, b, c, d]
       [0 c 0]         col_idx = [0, 2, 1, 0]
       [d 0 0]         row_ptr = [0, 2, 3, 4]

Address Calculation:
  For row r, iterate col_idx[row_ptr[r]] to col_idx[row_ptr[r+1]-1]
  value_addr = values_base + idx
  col_addr = col_idx_base + idx

Indirect Access:
  actual_col = memory[col_addr]
  dense_addr = base + r × stride + actual_col × element_size
```

Hardware Implications:

```
Dense Accelerator (TPU): Initiation interval = 1 (fully pipelined)
Sparse Accelerator (EIE): Initiation interval = 17 (metadata overhead)

Trade-off:
  - Sparse saves memory bandwidth (skip zeros)
  - But adds index indirection latency
  - Effective for >90% sparsity
```

26.10.8 9.8 Transformer/Attention Patterns

Modern LLM accelerators require specialized patterns:

Q, K, V Matrix Generation:

```
Input: X [seq_len, d_model]
Weights: W_Q, W_K, W_V [d_model, d_k]
```

```
Q = X × W_Q  →  [seq_len, d_k]
K = X × W_K  →  [seq_len, d_k]
V = X × W_V  →  [seq_len, d_k]

Address Pattern: Standard GEMM with different weight matrices
  q_addr = q_base + seq_idx × d_k + head_idx
  k_addr = k_base + seq_idx × d_k + head_idx
  v_addr = v_base + seq_idx × d_k + head_idx
```

Attention Score Computation:

```
Attention = softmax(Q × K^T / sqrt(d_k)) × V

Memory Challenge:
  Q × K^T produces [seq_len × seq_len] matrix
  O(n^2) memory for sequence length n

FlashAttention Tiling:
  Tile Q, K, V into blocks that fit in SRAM
  Compute attention incrementally without materializing full matrix

  Tile Address Pattern:
    q_tile_addr = q_base + tile_row × tile_size × d_k
    k_tile_addr = k_base + tile_col × tile_size × d_k

  Online Softmax:
    Track running max and sum across tiles
    No need to store intermediate attention matrix
```

26.10.9 9.9 Programmable Address Generation Unit (PAGU)

Academic reference for flexible tensor addressing:

PAGU Instruction Set:

```
Operations Supported:
  - Standard Convolution
  - Padded Convolution
  - Strided Convolution
  - Dilated (Atrous) Convolution
  - Pooling (Max, Average)
  - Upsampled Convolution
  - Transposed Convolution

Performance: 1.7 cycles per address for convolution
Area Overhead: ~4.6× vs fixed datapath (justified by flexibility)
```

Configuration Registers:

```
tensor_config_t:
  base_addr[63:0]           // Tensor base address
  dim_sizes[4][31:0]        // Size of each dimension
  dim_strides[4][31:0]      // Stride of each dimension
  kernel_size[15:0]         // Convolution kernel size
  conv_stride[7:0]          // Convolution stride
  dilation[7:0]             // Dilation factor
  padding[7:0]              // Padding amount
  operation_mode[3:0]       // Select operation type
```

Address Generation FSM:

```
always_ff @(posedge clk) begin
    case (state)
        IDLE: begin
            if (start) begin
                dim_counters <= '0;
                state <= GENERATE;
            end
        end

        GENERATE: begin
            // Multi-dimensional address calculation
            addr <= base_addr;
            for (int d = 0; d < num_dims; d++) begin
                case (operation_mode)
                    CONV_STANDARD:
                        addr <= addr + dim_counters[d] *
dim_strides[d];
                    CONV_STRIDED:
                        addr <= addr + (dim_counters[d] * conv_stride)
* dim_strides[d];
                    CONV_DILATED:
                        addr <= addr + (dim_counters[d] * dilation) *
dim_strides[d];
                endcase
            end

            // Increment counters (innermost first)
            increment_counters();

            if (done) state <= IDLE;
        end
    endcase
end
```

26.11 10. Implementation Comparison Matrix

26.11.1 10.1 Pattern Complexity vs. Capability

Pattern	Descriptor Fields	Address Logic	Hardware Cost	SW Flexibility
Linear	3 (src, dst, len)	Simple add	Low	Limited
2D Stride	6 (+x_len, y_len, strides)	Nested loop	Medium	Good
3D Volume	9 (+z_len, z_strides)	Triple nested	Medium-High	Very Good
Scatter-Gather	4 per fragment	List traversal	Medium	Excellent
Circular	4 (+size, wrap)	Modulo/mask	Low-Medium	Good
Tensor	N (variable stride)	Sum-of-products	High	Excellent

26.11.2 10.2 Use Case Mapping

Application Domain	Primary Pattern	Secondary Pattern
Block memory copy	Linear	-
Network packets	Scatter-Gather	Linear
Audio streaming	Circular	Linear
Image processing	2D Stride	Linear
Video encode/decode	2D Stride	Scatter-Gather
3D graphics	3D Volume	2D Stride
Neural networks	Tensor	2D Stride
Database operations	Scatter-Gather	Linear

26.11.3 10.3 RAPIDS Current vs. Extended Capability

Capability	Current RAPIDS	Industry Standard
Linear contiguous	YES	YES
Unaligned transfers	YES (3-phase)	Varies
Descriptor chaining	YES	YES

Capability	Current RAPIDS	Industry Standard
2D stride	NO	Common
3D volume	NO	Rare
Scatter-Gather	Partial (chain)	Full SG
Circular wrap	NO	Common
Tensor addressing	NO	Emerging

26.12 11. RAPIDS Extension Recommendations

26.12.1 11.0 PRIORITY 0: Tensor Accelerator Addressing (CRITICAL)

Rationale: Essential for AI/ML workloads. Tensor accelerators represent the highest-growth market for DMA engines. Without tensor addressing, RAPIDS cannot serve neural network inference applications.

Implementation Approach: Data Cube Model (NVDLA-style)

This approach provides the best balance of flexibility and hardware complexity, supporting convolution, pooling, and GEMM operations.

Extended Descriptor Format:

```
typedef struct packed {
    // === Base Fields (existing, 256 bits) ===
    logic [63:0]  src_addr;           // Base source address
    logic [63:0]  dst_addr;           // Base destination address
    logic [31:0]  length;             // For linear mode: beats
    logic [31:0]  next_descriptor_ptr;
    logic [7:0]   desc_priority;
    logic [3:0]   channel_id;
    logic         error;
    logic         last;
    logic         gen_irq;
    logic         valid;

    // === Tensor Extension (repurpose reserved bits + add 2nd
descriptor word) ===
    // Mode selection
    logic [3:0]   transfer_mode;      // 0=linear, 1=2D, 2=3D/tensor,
3=conv, 4=pool

    // Data cube dimensions
    logic [15:0]  width;              // W dimension (elements)
    logic [15:0]  height;             // H dimension (lines)
```

```
    logic [15:0]  channels;           // C dimension (surfaces)

    // Stride parameters (32-byte aligned)
    logic [31:0]  line_stride;        // Bytes between lines
    logic [31:0]  surface_stride;     // Bytes between surfaces

    // Convolution parameters (when transfer_mode == CONV)
    logic [7:0]   kernel_width;       // Kw
    logic [7:0]   kernel_height;      // Kh
    logic [7:0]   conv_stride_x;      // Horizontal stride
    logic [7:0]   conv_stride_y;      // Vertical stride
    logic [7:0]   dilation_x;         // Horizontal dilation
    logic [7:0]   dilation_y;         // Vertical dilation
    logic [7:0]   pad_left;           // Left padding
    logic [7:0]   pad_right;          // Right padding
    logic [7:0]   pad_top;            // Top padding
    logic [7:0]   pad_bottom;         // Bottom padding

    // Pooling parameters (when transfer_mode == POOL)
    logic [7:0]   pool_width;
    logic [7:0]   pool_height;
    logic [7:0]   pool_stride_x;
    logic [7:0]   pool_stride_y;
    logic [1:0]   pool_type;          // 0=max, 1=avg, 2=min

    // Data type and precision
    logic [3:0]   data_type;          // 0=int8, 1=int16, 2=fp16,
3=bf16, 4=fp32
    logic [3:0]   element_size;       // Bytes per element (1, 2, 4)
} descriptor_tensor_t;
```

Address Generation Unit (AGU) Architecture:

```
module tensor_address_generator #(
    parameter int PIPE_STAGES = 3     // Pipeline for multiply-
accumulate
) (
    input  logic        clk,
    input  logic        rst_n,

    // Configuration from descriptor
    input  descriptor_tensor_t desc,
    input  logic        start,

    // Address output stream
    output logic [63:0] addr,
    output logic        addr_valid,
    input  logic        addr_ready,
```

```
    // Status
    output logic        done
);

    // Dimension counters
    logic [15:0] w_cnt, h_cnt, c_cnt;
    logic [7:0]  kw_cnt, kh_cnt;       // Kernel position (for conv)

    // Pipeline registers for address calculation
    logic [63:0] addr_stage[PIPE_STAGES];

    // FSM states
    typedef enum logic [2:0] {
        IDLE,
        CALC_BASE,
        CALC_LINE,
        CALC_SURFACE,
        OUTPUT
    } state_t;
    state_t state;

    // Address calculation based on mode
    always_ff @(posedge clk or negedge rst_n) begin
        if (!rst_n) begin
            state <= IDLE;
            w_cnt <= '0;
            h_cnt <= '0;
            c_cnt <= '0;
        end else begin
            case (state)
                IDLE: begin
                    if (start) begin
                        w_cnt <= '0;
                        h_cnt <= '0;
                        c_cnt <= '0;
                        state <= CALC_BASE;
                    end
                end

                CALC_BASE: begin
                    // Stage 1: Base address
                    addr_stage[0] <= desc.src_addr;
                    state <= CALC_LINE;
                end

                CALC_LINE: begin
                    // Stage 2: Add line offset
```

```
                    case (desc.transfer_mode)
                        MODE_LINEAR:
                            addr_stage[1] <= addr_stage[0] + (w_cnt *
desc.element_size);

                        MODE_2D, MODE_3D:
                            addr_stage[1] <= addr_stage[0]
                                           + (h_cnt *
desc.line_stride)
                                           + (w_cnt *
desc.element_size);

                        MODE_CONV:
                            // Convolution: account for stride and
kernel position
                            addr_stage[1] <= addr_stage[0]
                                           + ((h_cnt *
desc.conv_stride_y + kh_cnt * desc.dilation_y) * desc.line_stride)
                                           + ((w_cnt *
desc.conv_stride_x + kw_cnt * desc.dilation_x) * desc.element_size);

                        MODE_POOL:
                            addr_stage[1] <= addr_stage[0]
                                           + ((h_cnt *
desc.pool_stride_y) * desc.line_stride)
                                           + ((w_cnt *
desc.pool_stride_x) * desc.element_size);
                    endcase
                    state <= CALC_SURFACE;
                end

                CALC_SURFACE: begin
                    // Stage 3: Add surface offset (for 3D/tensor)
                    if (desc.transfer_mode inside {MODE_3D,
MODE_CONV}) begin
                        addr_stage[2] <= addr_stage[1] + (c_cnt *
desc.surface_stride);
                    end else begin
                        addr_stage[2] <= addr_stage[1];
                    end
                    state <= OUTPUT;
                end

                OUTPUT: begin
                    if (addr_ready) begin
                        // Increment counters (innermost first)
                        if (desc.transfer_mode == MODE_CONV) begin
                            // Convolution: iterate kernel, then
spatial, then channels
```

```
                            if (kw_cnt < desc.kernel_width - 1) begin
                                kw_cnt <= kw_cnt + 1;
                            end else begin
                                kw_cnt <= '0;
                                if (kh_cnt < desc.kernel_height - 1)
begin
                                    kh_cnt <= kh_cnt + 1;
                                end else begin
                                    kh_cnt <= '0;
                                    // Continue to spatial/channel
iteration
                                    increment_spatial_counters();
                                end
                            end
                        end else begin
                            increment_spatial_counters();
                        end

                        if (transfer_complete())
                            state <= IDLE;
                        else
                            state <= CALC_BASE;
                    end
                end
            endcase
        end
    end

    assign addr = addr_stage[PIPE_STAGES-1];
    assign addr_valid = (state == OUTPUT);
    assign done = (state == IDLE) && !start;

endmodule
```

Scratchpad Integration (Gemmini-style):

```
// Memory address format for tensor scratchpad
typedef struct packed {
    logic        mem_select;       // 0=scratchpad, 1=accumulator
    logic        acc_mode;         // 0=overwrite, 1=accumulate
    logic        read_format;      // 0=scaled, 1=raw
    logic [28:0] row_addr;         // Row address within memory
} scratchpad_addr_t;

// DMA configuration for tensor loads
typedef struct packed {
    logic [63:0] dram_addr;        // Virtual address in main memory
    scratchpad_addr_t sp_addr;     // Scratchpad destination
    logic [15:0] rows;             // Number of rows to transfer
```

```
    logic [15:0] cols;             // Elements per row
    logic [31:0] dram_stride;      // Source memory stride
    logic [15:0] sp_stride;        // Scratchpad row stride (usually
1)
} tensor_dma_config_t;
```

Alignment Requirements:

```
All tensor addresses must satisfy:
  - base_addr aligned to 32 bytes (NVDLA requirement)
  - line_stride aligned to 32 bytes
  - surface_stride aligned to 32 bytes
  - For optimal performance: align to 64 bytes (RAPIDS beat size)

Verification:
  assert((desc.src_addr & 32'h1F) == 0);
  assert((desc.line_stride & 32'h1F) == 0);
  assert((desc.surface_stride & 32'h1F) == 0);
```

Performance Considerations:

```
Address Generation Rate:
  - Target: 1 address per cycle (pipelined)
  - Convolution: 1.7 cycles/address typical (due to kernel iteration)

Memory Bandwidth:
  - TPU-class: 600+ GB/s (HBM)
  - FPGA-class: 20-80 GB/s (DDR4/5)
  - RAPIDS target: Match AXI interface bandwidth

Tiling Support:
  - Hardware computes tile boundaries
  - Double-buffering via ping-pong descriptors
  - Software provides tile dimensions, hardware handles addressing
```

26.12.2 11.1 Priority 1: 2D Stride Support

Rationale: High value for image/video processing with moderate complexity. Also serves as foundation for tensor addressing.

Descriptor Extension:

```
typedef struct packed {
    // Existing fields (256 bits)...

    // 2D extension (96 bits in reserved space)
    logic [31:0] x_length;         // Bytes per row
    logic [15:0] y_length;         // Number of rows
    logic [31:0] src_stride;       // Source row stride
```

```
    logic [31:0] dst_stride;      // Destination row stride
    logic        mode_2d;         // Enable 2D mode
} descriptor_2d_t;
```

Address Generation:

```
always_comb begin
    if (mode_2d) begin
        row_src_addr = src_addr + current_row * src_stride;
        row_dst_addr = dst_addr + current_row * dst_stride;
        row_length   = x_length;
    end else begin
        row_src_addr = src_addr;
        row_dst_addr = dst_addr;
        row_length   = length * BYTES_PER_BEAT;
    end
end
```

26.12.3 11.2 Priority 2: Circular Buffer Mode

Rationale: Essential for streaming applications, low complexity addition.

Descriptor Extension:

```
typedef struct packed {
    // Existing fields...

    // Circular buffer extension
    logic [31:0] buffer_size;     // Total buffer size
    logic [31:0] buffer_mask;     // Size-1 for power-of-2
    logic        circular_mode;   // Enable wrap-around
    logic        src_circular;    // Apply to source
    logic        dst_circular;    // Apply to destination
} descriptor_circular_t;
```

Address Generation:

```
function automatic logic [63:0] wrap_address(
    logic [63:0] base,
    logic [63:0] offset,
    logic [31:0] mask,
    logic        enable
);
    if (enable)
        return base + (offset & {32'b0, mask});
    else
        return base + offset;
endfunction
```

26.12.4 11.3 Priority 3: Enhanced Scatter-Gather

Rationale: Full SG capability for network and fragmented memory use cases.

Current Limitation: Chain-only SG requires one descriptor per fragment.

Enhancement: Add fragment list support within single descriptor:

```
typedef struct packed {
    logic [63:0] fragment_list_ptr;  // Pointer to fragment array
    logic [15:0] fragment_count;     // Number of fragments
    logic        sg_mode;            // Enable SG mode

    // Fragment entry format (in memory):
    // [63:0]  address
    // [31:0]  length
    // [31:0]  flags
} descriptor_sg_t;
```

26.13 12. References

26.13.1 Industry Documentation

1. AMD AXI DMA Product Guide (PG021) - Scatter-Gather descriptor format
2. Analog Devices AXI DMAC - 2D transfer implementation
3. Understanding AXI Addressing (ZipCPU) - Burst mode analysis
4. WRAP Address Calculation (Verification Guide) - Wrap burst formulas

26.13.2 GPU and Accelerator Patterns

5. NVIDIA CUDA Memory Coalescing - Coalesced access patterns
6. CUDA Pitch Linear Memory - 2D array allocation
7. NVIDIA TMA Tutorial (CUTLASS) - Tensor Memory Accelerator
8. Efficient GEMM in CUDA (CUTLASS) - Hierarchical tiling

26.13.3 DMA Architecture

9. Intel DMA Descriptors - PCIe DMA linked lists
10. DMA330 Microcode - Linked-list implementation

26.13.4 Neural Network Accelerators

11. NVDLA Hardware Architecture - Data cube model, CDMA
12. NVDLA In-Memory Data Formats - Line stride, surface stride
13. NVDLA Unit Description - Convolution DMA details
14. Gemmini Accelerator (UC Berkeley) - Open-source systolic array

15. Gemmini Documentation (Chipyard) - DMA and scratchpad
16. Google TPU Architecture - Systolic array overview
17. TPU System Architecture - Memory hierarchy
18. Programmable AGU for DNNs - Flexible address generation

26.13.5 Convolution and Tensor Operations

19. Im2Col Characterization - Convolution memory patterns
20. ST Neural-ART Programming Model - NPU descriptor concepts
21. Systolic Array Dataflows Survey - Weight/input/output stationary
22. FlashAttention Analysis - Attention tiling patterns
23. Depthwise Separable Convolutions - Channel-wise addressing

26.13.6 Sparse Tensor Formats

24. FLAASH Sparse Tensor Accelerator - CSR/CSF formats
25. SparTen CNN Accelerator - Sparse tensor addressing
26. Indirection Stream Architecture - Hardware indirect addressing

---

26.14 Document History

Version	Date	Author	Changes
1.0	2026-01-13	RTL Design Sherpa	Initial comprehensive analysis

This document is part of the RAPIDS (Rapid AXI Programmable In-band Descriptor System) design documentation.