



# RTL Design Sherpa

**STREAM Subsystem Hardware Architecture Specification 0.90**

**January 3, 2026**

## Table of Contents

1 Stream Has Index.....	22
2 Document Information.....	22
2.1 STREAM Hardware Architecture Specification.....	22
2.2 Document Purpose.....	22
2.3 References.....	22
2.4 Terminology.....	23
2.5 Revision History.....	23
3 Purpose and Scope.....	24
3.1 Document Purpose.....	24
3.2 Scope.....	24
3.2.1 In Scope.....	24
3.2.2 Out of Scope.....	25
3.3 Design Philosophy.....	25
3.3.1 Tutorial-Focused Simplicity.....	25
3.3.2 Relationship to RAPIDS.....	26
3.4 Intended Audience.....	26
4 Document Conventions.....	26
4.1 Notation Conventions.....	26
4.1.1 Signal Naming.....	26
4.1.2 Interface Prefixes.....	27
4.2 Diagram Conventions.....	27
4.2.1 Block Diagrams.....	27

4.2.2 Timing Diagrams.....	27
4.3 Numeric Conventions.....	27
4.3.1 Units.....	27
4.3.2 Address Notation.....	28
4.3.3 Size Notation.....	28
4.4 Requirement Notation.....	28
4.5 Reference Format.....	28
5 Definitions and Acronyms.....	29
5.1 Acronyms.....	29
5.2 Definitions.....	29
5.2.1 Architectural Terms.....	29
5.2.2 Interface Terms.....	30
5.2.3 Transfer Terms.....	30
5.3 Units and Measurements.....	31
5.4 Document-Specific Terms.....	31
6 Use Cases.....	32
6.1 Primary Use Cases.....	32
6.1.1 Memory-to-Memory Data Movement.....	32
6.1.2 Scatter-Gather Operations.....	32
6.1.3 Multi-Channel Concurrent Transfers.....	32
6.2 Secondary Use Cases.....	33
6.2.1 Buffer Management.....	33
6.2.2 Memory Initialization.....	33
6.3 Target Applications.....	33

7 Key Features.....	33
7.1 Feature Summary.....	33
7.2 Descriptor-Based Operation.....	34
7.2.1 Single-Write Kick-off.....	34
7.2.2 Autonomous Chaining.....	34
7.3 Multi-Channel Architecture.....	34
7.3.1 Channel Independence.....	34
7.3.2 Resource Sharing.....	34
7.3.3 Priority-Based Arbitration.....	35
7.4 Data Path Features.....	35
7.4.1 Parameterizable Data Width.....	35
7.4.2 Aligned Address Requirement.....	35
7.5 Monitoring and Debug.....	36
7.5.1 MonBus Integration.....	36
7.5.2 Status Visibility.....	36
7.6 Error Handling.....	36
7.6.1 Error Detection.....	36
7.6.2 Error Isolation.....	36
8 System Context.....	37
8.1 System Integration Overview.....	37
8.2 External Connections.....	39
8.2.1 Configuration Path.....	39
8.2.2 Data Paths.....	39
8.2.3 Monitoring Path.....	39

8.3 Clock and Reset.....	40
8.3.1 Clock Domain.....	40
8.3.2 Reset Requirements.....	40
8.4 Memory Map Requirements.....	40
8.4.1 Descriptor Memory.....	40
8.4.2 Data Memory.....	40
8.5 Interrupt Integration.....	41
8.5.1 Interrupt Signals.....	41
8.5.2 Interrupt Conditions.....	41
8.6 Power Considerations.....	41
8.6.1 Clock Gating.....	41
8.6.2 Power States.....	41
9 Block Diagram.....	42
9.1 Top-Level Architecture.....	42
9.2 Functional Block Summary.....	43
9.3 Block Interactions.....	43
9.3.1 Control Flow.....	43
9.3.2 Data Flow.....	43
9.3.3 Monitoring Flow.....	43
9.4 Parameterization.....	44
10 Data Flow.....	44
10.1 Transfer Sequence Overview.....	44
10.2 Phase Details.....	46
10.2.1 Phase 1: Kick-off.....	46

10.2.2 Phase 2: Descriptor Fetch.....	46
10.2.3 Phase 3: Descriptor Parse.....	46
10.2.4 Phase 4: Arbitration.....	46
10.2.5 Phase 5: Read Phase.....	47
10.2.6 Phase 6: Write Phase.....	47
10.2.7 Phase 7: Chain Check.....	47
10.3 Concurrent Operation.....	47
10.3.1 Multi-Channel Concurrency.....	47
10.3.2 Pipeline Overlap.....	47
10.4 Data Path Bandwidth.....	48
10.4.1 Theoretical Maximum.....	48
10.4.2 Practical Considerations.....	48
11 Multi-Channel Architecture.....	48
11.1 Channel Organization.....	48
11.2 Per-Channel Resources.....	49
11.3 Shared Resources.....	49
11.4 Arbitration Scheme.....	50
11.4.1 Priority-Based Arbitration.....	50
11.4.2 Fairness Mechanism.....	50
11.4.3 Arbitration Points.....	50
11.5 Channel States.....	50
11.6 SRAM Partitioning.....	52
11.6.1 Static Partitioning (Default).....	52
11.6.2 Dynamic Allocation (Optional).....	53

11.7 Error Isolation.....	53
12 AXI Master Interfaces.....	53
12.1 Overview.....	53
12.2 Descriptor Fetch Master.....	54
12.2.1 Signal Summary.....	54
12.2.2 Transaction Characteristics.....	54
12.2.3 Address Alignment.....	55
12.3 Data Read Master.....	55
12.3.1 Signal Summary.....	55
12.3.2 Transaction Characteristics.....	56
12.3.3 Burst Generation.....	56
12.4 Data Write Master.....	56
12.4.1 Signal Summary.....	56
12.4.2 Transaction Characteristics.....	57
12.5 Common AXI Attributes.....	57
12.5.1 Supported Features.....	57
12.5.2 Error Handling.....	58
13 APB Configuration Interface.....	58
13.1 Overview.....	58
13.2 Signal Summary.....	58
13.3 Register Map Overview.....	59
13.4 Key Registers.....	60
13.4.1 Global Control (CTRL) - 0x000.....	60
13.4.2 Global Status (STATUS) - 0x004.....	60

13.4.3 Channel Control (CHn_CTRL) - 0x040 + n*0x10.....	60
13.4.4 Channel Status (CHn_STATUS) - 0x044 + n*0x10.....	60
13.5 Access Timing.....	61
13.5.1 Write Access.....	61
13.5.2 Read Access.....	62
13.6 Kick-off Blocking.....	62
14 Monitoring Interface.....	62
14.1 Overview.....	62
14.2 Signal Summary.....	63
14.3 Packet Format.....	63
14.4 Event Types.....	63
14.4.1 Transfer Events (EVENT_TYPE = 0x01).....	63
14.4.2 Error Events (EVENT_TYPE = 0x02).....	64
14.4.3 Performance Events (EVENT_TYPE = 0x03).....	64
14.5 Event Generation.....	65
14.5.1 Configurable Events.....	65
14.5.2 Event Priority.....	65
14.6 Backpressure Handling.....	65
14.6.1 Flow Control.....	65
14.6.2 Recommendations.....	65
14.7 Typical MonBus Downstream.....	65
14.8 Example Event Sequence.....	66
15 Throughput Targets.....	66
15.1 Theoretical Maximum.....	66

15.1.1 Per-Interface Bandwidth.....	66
15.1.2 Aggregate Bandwidth.....	66
15.2 Practical Throughput.....	67
15.2.1 Single-Channel Performance.....	67
15.2.2 Multi-Channel Performance.....	67
15.3 Throughput Limiting Factors.....	67
15.3.1 Memory System Factors.....	67
15.3.2 STREAM Internal Factors.....	68
15.4 Performance Targets.....	68
15.4.1 Design Targets.....	68
15.4.2 Verification Criteria.....	68
15.5 Descriptor Chain Efficiency.....	69
15.5.1 Descriptor Overhead.....	69
15.5.2 Chaining Efficiency.....	69
16 Latency Characteristics.....	69
16.1 Latency Components.....	69
16.1.1 End-to-End Latency Breakdown.....	69
16.1.2 Total Latency Formula.....	70
16.2 First-Byte Latency.....	70
16.3 Latency by Transfer Size.....	71
16.3.1 Small Transfers.....	71
16.3.2 Large Transfers.....	71
16.4 Latency Variability.....	71
16.4.1 Sources of Variability.....	71

16.4.2 Worst-Case Latency.....	71
16.5 Latency Optimization.....	72
16.5.1 Software Recommendations.....	72
16.5.2 Hardware Configuration.....	72
16.6 Interrupt Latency.....	72
17 Resource Estimates.....	73
17.1 FPGA Resource Summary.....	73
17.2 Resource Breakdown by Block.....	73
17.2.1 APB Configuration Slave.....	73
17.2.2 Descriptor Engine.....	73
17.2.3 Scheduler (8 channels).....	73
17.2.4 Channel Arbiter.....	74
17.2.5 AXI Read Engine.....	74
17.2.6 SRAM Buffer.....	74
17.2.7 AXI Write Engine.....	74
17.2.8 MonBus Reporter.....	74
17.3 SRAM Sizing.....	74
17.4 Scaling with Parameters.....	75
17.4.1 Channel Count Impact.....	75
17.4.2 Data Width Impact.....	75
17.5 ASIC Estimates.....	75
17.5.1 Power Estimates.....	76
17.6 Timing Estimates.....	76
17.6.1 Target Frequencies.....	76

17.6.2 Critical Paths.....	76
<b>18 System Requirements.....</b>	<b>77</b>
<b>18.1 Interface Requirements.....</b>	<b>77</b>
<b>18.1.1 APB Interface.....</b>	<b>77</b>
<b>18.1.2 AXI Interfaces.....</b>	<b>77</b>
<b>18.1.3 MonBus Interface.....</b>	<b>77</b>
<b>18.2 Memory Requirements.....</b>	<b>77</b>
<b>18.2.1 Descriptor Memory.....</b>	<b>77</b>
<b>18.2.2 Data Memory.....</b>	<b>78</b>
<b>18.3 System Bandwidth Allocation.....</b>	<b>78</b>
<b>18.3.1 AXI Interconnect Requirements.....</b>	<b>78</b>
<b>18.3.2 Interconnect Recommendations.....</b>	<b>78</b>
<b>18.4 Interrupt Requirements.....</b>	<b>78</b>
<b>18.4.1 Interrupt Controller.....</b>	<b>78</b>
<b>18.4.2 Interrupt Handling.....</b>	<b>79</b>
<b>18.5 Address Space Requirements.....</b>	<b>79</b>
<b>18.5.1 STREAM Register Space.....</b>	<b>79</b>
<b>18.5.2 Memory Map Constraints.....</b>	<b>79</b>
<b>18.6 Software Requirements.....</b>	<b>79</b>
<b>18.6.1 Driver Requirements.....</b>	<b>79</b>
<b>18.6.2 Programming Sequence.....</b>	<b>80</b>
<b>19 Clocking and Reset.....</b>	<b>80</b>
<b>19.1 Clock Requirements.....</b>	<b>80</b>
<b>19.1.1 Primary Clock (aclk).....</b>	<b>80</b>

19.1.2 Clock Relationship.....	80
19.2 Reset Requirements.....	81
19.2.1 Reset Signal (aresetn).....	81
19.2.2 Reset Sequence.....	81
19.2.3 Reset Behavior.....	82
19.3 Power Management.....	82
19.3.1 Clock Gating Support.....	82
19.3.2 Low Power Recommendations.....	82
19.4 Timing Constraints.....	82
19.4.1 Clock Period Constraints.....	82
19.4.2 False Path Constraints.....	83
19.5 CDC Considerations.....	83
19.5.1 Single Clock Domain.....	83
19.5.2 CDC Recommendations.....	83
20 Verification Strategy.....	84
20.1 Verification Approach.....	84
20.1.1 Multi-Level Verification.....	84
20.2 Block-Level Verification.....	84
20.2.1 Test Categories.....	84
20.2.2 Coverage Metrics.....	85
20.3 Integration-Level Verification.....	85
20.3.1 Interface Verification.....	85
20.3.2 Multi-Channel Scenarios.....	85
20.4 System-Level Verification.....	86

20.4.1 End-to-End Test Cases.....	86
20.4.2 Performance Verification.....	86
20.5 Testbench Architecture.....	87
20.5.1 CocoTB-Based Verification.....	87
20.5.2 Key Testbench Components.....	87
20.6 Verification Environment.....	88
20.6.1 Simulator Support.....	88
20.6.2 Test Execution.....	88
20.7 Formal Verification.....	88
20.7.1 Properties for Formal.....	88
20.7.2 Formal Tool Usage.....	88
20.8 Sign-Off Criteria.....	89
20.8.1 Block-Level Sign-Off.....	89
20.8.2 Integration Sign-Off.....	89
20.8.3 System Sign-Off.....	89
21 STREAM Specification Index.....	89
21.1 Document Organization.....	89
21.1.1 Front Matter.....	89
21.1.2 Chapter 1: Overview.....	90
21.1.3 Chapter 2: Functional Blocks.....	90
21.1.4 Chapter 3: Interfaces.....	90
21.1.5 Chapter 4: Registers.....	90
21.1.6 Chapter 5: Programming.....	90
21.1.7 Chapter 6: Configuration Reference.....	90

21.2 Quick Reference.....	91
21.2.1 Functional Unit Blocks (FUB).....	91
21.2.2 Integration Blocks (MAC).....	91
21.3 Performance Modes (AXI Engines).....	92
21.3.1 Holistic Overview: V1/V2/V3 Working Together.....	92
21.3.2 V1 - Low Performance (Single Outstanding Transaction).....	92
21.3.3 V2 - Medium Performance (Command Pipelined).....	93
21.3.4 V3 - High Performance (Out-of-Order Completion).....	93
21.3.5 Performance Comparison Summary.....	93
21.4 Clock and Reset Summary.....	94
21.4.1 Clock Domains.....	94
21.4.2 Reset Signals.....	94
21.5 Interface Summary.....	95
21.5.1 External Interfaces.....	95
21.5.2 Internal Buses.....	95
21.6 Area Estimates.....	95
21.6.1 By Performance Mode.....	95
21.6.2 Breakdown (Low Performance).....	96
21.7 Related Documentation.....	96
21.8 Specification Conventions.....	97
21.8.1 Signal Naming.....	97
21.8.2 Parameter Naming.....	97
21.8.3 State Machine Naming.....	97
22 Product Requirements Document (PRD).....	97

22.1 STREAM - Scatter-gather Transfer Rapid Engine for AXI Memory.....	97
22.2 1. Executive Summary.....	97
22.2.1 1.1 Quick Stats.....	98
22.2.2 1.2 Project Goals.....	98
22.3 2. Key Design Principles.....	98
22.3.1 2.1 Simplifications from RAPIDS.....	98
22.3.2 2.2 Tutorial-Friendly Features.....	99
22.4 3. Architecture Overview.....	99
22.4.1 3.1 Top-Level Block Diagram.....	99
22.4.2 3.2 Data Flow.....	99
22.5 4. Interfaces.....	100
22.5.1 4.1 External Interfaces.....	100
22.5.2 4.2 Descriptor Format.....	101
22.6 5. Key Components.....	101
22.6.1 5.1 Descriptor Engine (APB-Only for STREAM).....	101
22.6.2 5.2 Scheduler Group (Integration Wrapper).....	102
22.6.3 5.3 Scheduler (Simplified from RAPIDS).....	104
22.6.4 5.3 AXI Read Engine (Streaming Pipeline - NO FSM).....	104
22.6.5 5.4 AXI Write Engine (Streaming Pipeline - NO FSM).....	105
22.6.6 5.5 Simple SRAM.....	106
22.7 6. Configuration and Control.....	106
22.7.1 6.1 APB Register Map.....	106
22.7.2 6.2 Channel Configuration.....	107
22.8 7. Resource Sharing and Arbitration.....	107

22.8.1 7.1 Shared Resources.....	107
22.8.2 7.2 Arbitration Strategy.....	107
22.9 8. Error Detection and Recovery.....	108
22.9.1 8.1 Error Types.....	108
22.9.2 8.2 Error Recovery.....	108
22.10 9. MonBus Integration.....	108
22.10.1 9.1 Standard MonBus Format.....	108
22.10.2 9.2 STREAM Event Codes.....	109
22.10.3 9.3 Default Configuration.....	109
22.11 10. Design Constraints.....	109
22.11.1 10.1 Tutorial Constraints (Intentional Simplifications).....	109
22.11.2 10.2 Implementation Constraints.....	110
22.12 11. Verification Strategy.....	110
22.12.1 11.1 Test Organization.....	110
22.12.2 11.2 Test Levels.....	110
22.13 12. Performance Characteristics.....	111
22.13.1 12.1 Throughput by Engine Version.....	111
22.13.2 12.2 Latency.....	111
22.13.3 12.3 Resource Utilization by Engine Version.....	111
22.14 13. Development Roadmap.....	112
22.14.1 13.1 Phase 1: Foundation (Current).....	112
22.14.2 13.2 Phase 2: Core Blocks.....	112
22.14.3 13.3 Phase 3: Data Path.....	112
22.14.4 13.4 Phase 4: Integration.....	112

22.14.5 13.5 Phase 5: Multi-Channel.....	113
22.14.6 13.6 Phase 6: Advanced Engines (Future - V2/V3).....	113
22.15 14. Educational Value.....	113
22.15.1 14.1 Learning Objectives.....	113
22.15.2 14.2 Progression Path.....	114
22.16 15. Success Criteria.....	114
22.16.1 15.1 Functional.....	114
22.16.2 15.2 Quality.....	114
22.16.3 15.3 Performance.....	114
22.17 16. Open Questions (For Review).....	115
22.17.1 16.1 Descriptor Engine Adaptation.....	115
22.17.2 16.2 AXI Descriptor Master.....	115
22.17.3 16.3 Channel Arbitration.....	115
22.17.4 16.4 SRAM Partitioning.....	115
22.18 16. Attribution and Contribution Guidelines.....	115
22.18.1 16.1 Git Commit Attribution.....	115
22.19 16.2 PDF Generation Location.....	116
22.20 17. References.....	116
22.20.1 17.1 Internal Documentation.....	116
22.20.2 17.2 External References.....	116
22.21 Navigation.....	117
23 Claude Code Guide: STREAM Subsystem.....	117
23.1 Quick Context.....	117
23.2  Global Requirements Reference.....	117

23.3 Critical Rules for This Subsystem.....	117
23.3.1 Rule #0: Attribution Format for Git Commits.....	117
23.3.2 Rule #0.1: TUTORIAL FOCUS - Intentional Simplifications.....	118
23.3.3 Rule #0.1: Testbench Location and Test Structure (MANDATORY)....	118
23.3.4 Rule #0.2: Three Mandatory TB Methods (MANDATORY).....	119
23.3.5 Rule #1: REUSE from RAPIDS Where Appropriate.....	119
23.3.6 Rule #2: Descriptor Format is DIFFERENT from RAPIDS.....	119
23.3.7 Rule #3: Know the Shared Resources.....	120
23.4 Architecture Quick Reference.....	120
23.4.1 Block Organization.....	120
23.4.2 Module Status.....	120
23.5 Common User Questions and Responses.....	121
23.5.1 Q: “How is STREAM different from RAPIDS?”.....	121
23.5.2 Q: “How do I kick off a transfer?”.....	121
23.5.3 Q: “How many channels can I use?”.....	122
23.5.4 Q: “What’s the descriptor format?”.....	122
23.5.5 Q: “How do I run STREAM tests?”.....	123
23.6 Integration Patterns.....	124
23.6.1 Pattern 1: Basic STREAM Instantiation.....	124
23.6.2 Pattern 2: Descriptor Creation (Software Model).....	125
23.6.3 Pattern 3: MonBus Integration.....	125
23.7 Anti-Patterns to Catch.....	126
23.7.1 ✗ Anti-Pattern 1: Adding Alignment Fixup.....	126
23.7.2 ✗ Anti-Pattern 2: Using Length in Bytes.....	126

23.7.3 ✗ Anti-Pattern 3: Circular Buffer Descriptors.....	126
23.7.4 ✗ Anti-Pattern 4: Assuming Exclusive Channel Access.....	126
23.8 Debugging Workflow.....	127
23.8.1 Issue: Descriptor Not Fetched.....	127
23.8.2 Issue: Data Transfer Stalls.....	127
23.8.3 Issue: Chained Descriptors Not Following.....	127
23.9 Testing Guidance.....	127
23.9.1 Test Organization.....	127
23.9.2 Test Levels.....	128
23.10 Key Documentation Links.....	128
23.10.1 Always Reference These.....	128
23.11 Quick Commands.....	128
23.12 Remember.....	129
23.13 PDF Generation Location.....	129

## List of Figures

*No figures in this document.*

## List of Tables

Table 1: Reference Documents.....	22
Table 2: Terminology and Definitions.....	23
Table 3: Document Revision History.....	23

# 1 Stream Has Index

Generated: 2026-01-03

## 2 Document Information

### 2.1 STREAM Hardware Architecture Specification

**Document Number:** STREAM-HAS-001 **Version:** 0.90 **Status:** Draft **Classification:** Open Source - Apache 2.0 License

---

### 2.2 Document Purpose

This Hardware Architecture Specification (HAS) provides a high-level architectural overview of the STREAM (Scatter-gather Transfer Rapid Engine for AXI Memory) subsystem. It describes the system-level design, external interfaces, performance characteristics, and integration requirements without detailing internal implementation specifics.

**Target Audience:** - System architects evaluating STREAM for integration - Hardware engineers planning system-level integration - Software engineers developing drivers and firmware - Verification engineers planning system-level testing

**Companion Documents:** - STREAM Micro-Architecture Specification (MAS) - Detailed block-level implementation - STREAM Product Requirements Document (PRD) - Requirements and rationale

---

### 2.3 References

#### Reference Documents

ID	Document	Description
[REF-1]	STREAM MAS v0.90	Micro-Architecture Specification
[REF-2]	STREAM PRD	Product Requirements Document
[REF-3]	ARM AMBA AXI4	AXI4 Protocol Specification
[REF-4]	ARM AMBA APB	APB Protocol

ID	Document	Description
		Specification

## 2.4 Terminology

---

### *Terminology and Definitions*

Term	Definition
AXI	Advanced eXtensible Interface - ARM AMBA high-performance bus
APB	Advanced Peripheral Bus - ARM AMBA low-power configuration bus
Beat	Single data transfer on AXI bus (one clock cycle of valid data)
Burst	Sequence of consecutive beats forming a single AXI transaction
Channel	Independent DMA transfer context (STREAM supports 8 channels)
Descriptor	256-bit data structure defining a single DMA transfer operation
DMA	Direct Memory Access - data transfer without CPU involvement
HAS	Hardware Architecture Specification
MAS	Micro-Architecture Specification
MonBus	Monitor Bus - internal debug/trace event streaming interface
Scatter-Gather	DMA mode using linked descriptors for non-contiguous transfers

## 2.5 Revision History

---

### *Document Revision History*

Version	Date	Author	Description
0.90	2026-01-03	seang	Initial HAS

Version	Date	Author	Description
			release

**Last Updated:** 2026-01-03

## 3 Purpose and Scope

### 3.1 Document Purpose

This Hardware Architecture Specification (HAS) defines the high-level architecture of the STREAM (Scatter-gather Transfer Rapid Engine for AXI Memory) subsystem. It serves as the primary reference for:

- **System Integration** - Understanding STREAM's external interfaces and system-level requirements
- **Performance Planning** - Evaluating throughput, latency, and resource characteristics
- **Driver Development** - Programming model and software interface requirements
- **Verification Planning** - System-level test scenarios and coverage requirements

This document complements the STREAM Micro-Architecture Specification (MAS), which provides detailed block-level implementation specifics.

---

### 3.2 Scope

#### 3.2.1 In Scope

This specification covers:

1. **System Architecture**
  - Block-level functional partitioning
  - Data flow through the subsystem
  - Multi-channel operation model
2. **External Interfaces**
  - APB configuration slave interface

- AXI4 master interfaces (descriptor fetch, data read, data write)
- Monitor bus (MonBus) output interface

### 3. Performance Characteristics

- Throughput targets and limiting factors
- Latency breakdown for typical operations
- Resource estimates for FPGA/ASIC targets

### 4. Integration Requirements

- Clock and reset requirements
- System-level constraints
- Verification strategy overview

#### 3.2.2 Out of Scope

This specification does not cover:

- **Detailed Micro-Architecture** - See STREAM MAS for block-level implementation
  - **RTL Implementation Details** - Register-level timing, pipeline stages
  - **Testbench Architecture** - See verification documentation
  - **Software Driver Implementation** - See software programming guide
- 

## 3.3 Design Philosophy

---

### 3.3.1 Tutorial-Focused Simplicity

STREAM is intentionally designed as a beginner-friendly DMA engine tutorial. Key simplifications include:

Aspect	Simplification	Rationale
<b>Address Alignment</b>	Aligned addresses only	Eliminates complex fixup logic
<b>Transfer Length</b>	Specified in beats	Simplifies datapath math
<b>Descriptor Chaining</b>	Linear chains only	No circular buffer management
<b>Flow Control</b>	Simple transaction limits	No credit management complexity

These simplifications make STREAM an ideal learning platform while maintaining production-quality RTL practices.

### 3.3.2 Relationship to RAPIDS

STREAM is derived from the RAPIDS (Rapid AXI Programmable In-band Descriptor System) architecture but removes:

- Network interfaces (source/sink paths)
- Address alignment fixup logic
- Credit-based flow control
- Control read/write engines

This creates a pure memory-to-memory DMA engine suitable for understanding core DMA concepts.

---

## 3.4 Intended Audience

Audience	Primary Use
System Architects	Evaluate STREAM for system integration
Hardware Engineers	Plan physical integration and timing closure
Software Engineers	Develop drivers and firmware
Verification Engineers	Plan system-level test coverage

**Last Updated:** 2026-01-03

## 4 Document Conventions

### 4.1 Notation Conventions

#### 4.1.1 Signal Naming

Prefix/Suffix	Meaning	Example
i_	Input signal	i_clk, i_data
o_	Output signal	o_valid, o_ready
r_	Registered signal	r_state, r_count

Prefix/Suffix	Meaning	Example
w_	Combinational wire	w_next_state
_n	Active-low signal	aresetn, rst_n

#### 4.1.2 Interface Prefixes

Prefix	Protocol	Direction
s_apb_	APB slave	Inbound
m_axi_desc_	AXI4 descriptor master	Outbound
m_axi_rd_	AXI4 read master	Outbound
m_axi_wr_	AXI4 write master	Outbound
monbus_	Monitor bus	Outbound

---

## 4.2 Diagram Conventions

#### 4.2.1 Block Diagrams

- **Rectangles** - Functional blocks
- **Arrows** - Data flow direction
- **Dashed Lines** - Configuration/control paths
- **Double Lines** - High-bandwidth data paths

#### 4.2.2 Timing Diagrams

- **High** - Logic 1
  - **Low** - Logic 0
  - **X** - Don't care / undefined
  - **Hatched** - Valid data window
- 

---

## 4.3 Numeric Conventions

---

#### 4.3.1 Units

Unit	Meaning
Beat	Single data transfer (one clock cycle of valid data)

Unit	Meaning
Burst	Sequence of consecutive beats
Transaction	Complete AXI transaction (address + all data beats)

#### 4.3.2 Address Notation

- All addresses are specified in bytes unless otherwise noted
- Address alignment requirements are specified relative to data width
- Example: 512-bit data width requires 64-byte (512/8) alignment

#### 4.3.3 Size Notation

Notation	Value
KB	1,024 bytes
MB	1,048,576 bytes
Kbit	1,024 bits

## 4.4 Requirement Notation

---

Requirements and constraints are identified using the following format:

Prefix	Category
[REQ-x]	Functional requirement
[PERF-x]	Performance requirement
[INTEG-x]	Integration requirement
[CONST-x]	Design constraint

## 4.5 Reference Format

---

Cross-references to other documents use the format:

- [MAS-x.y] - Reference to MAS section x.y
  - [REF-n] - Reference to document in reference table
- 

**Last Updated:** 2026-01-03

## 5 Definitions and Acronyms

### 5.1 Acronyms

Acronym	Definition
APB	Advanced Peripheral Bus
AXI	Advanced eXtensible Interface
AMBA	Advanced Microcontroller Bus Architecture
BFM	Bus Functional Model
CDC	Clock Domain Crossing
DMA	Direct Memory Access
FIFO	First-In First-Out buffer
FSM	Finite State Machine
HAS	Hardware Architecture Specification
MAS	Micro-Architecture Specification
MonBus	Monitor Bus (internal debug/trace interface)
PRD	Product Requirements Document
RTL	Register Transfer Level
SG	Scatter-Gather
SRAM	Static Random Access Memory
STREAM	Scatter-gather Transfer Rapid Engine for AXI Memory

### 5.2 Definitions

#### 5.2.1 Architectural Terms

Term	Definition
Beat	A single data transfer on an AXI bus, corresponding to one clock cycle where valid data is transferred.

Term	Definition
<b>Burst</b>	For a 512-bit data bus, one beat transfers 64 bytes.
<b>Channel</b>	A sequence of consecutive beats forming a single AXI transaction. STREAM supports burst lengths up to 256 beats (AXI4 maximum).
<b>Descriptor</b>	An independent DMA transfer context. STREAM supports 8 channels, each capable of processing its own descriptor chain.
<b>Descriptor Chain</b>	A 256-bit data structure in memory that defines a single DMA transfer operation, including source address, destination address, length, and chaining information.
<b>Scatter-Gather</b>	A linked list of descriptors where each descriptor points to the next, enabling complex scatter-gather transfer patterns.
<b>Scatter-Gather</b>	A DMA technique where data is transferred to/from non-contiguous memory locations using a list of descriptors.

### 5.2.2 Interface Terms

Term	Definition
<b>APB Slave</b>	STREAM's configuration interface, receiving programming commands from the system processor.
<b>AXI Master</b>	STREAM's memory interfaces for descriptor fetches and data transfers. Three masters: descriptor fetch, data read, data write.
<b>MonBus</b>	Monitor Bus - a 64-bit streaming interface for debug and performance monitoring packets.
<b>Backpressure</b>	Flow control mechanism where a receiver signals inability to accept data, causing the sender to pause.

### 5.2.3 Transfer Terms

Term	Definition
<b>Aligned Address</b>	An address that is a multiple of the data width in

Term	Definition
	bytes. For 512-bit (64-byte) data width, aligned addresses are multiples of 64.
<b>Kick-off</b>	Initiating a DMA transfer by writing the first descriptor address to a channel's control register.
<b>Completion</b>	The successful end of a descriptor chain, optionally generating an interrupt.
<b>Chaining</b>	Following the <code>next_descriptor_ptr</code> to fetch and process subsequent descriptors.

### 5.3 Units and Measurements

Term	Definition
<b>Clock Cycle</b>	One period of the primary system clock (aclk).
<b>Latency</b>	Time from initiating an operation to its completion, typically measured in clock cycles.
<b>Throughput</b>	Data transfer rate, typically measured in bytes per second or beats per cycle.
<b>Utilization</b>	Percentage of available bandwidth actually used for data transfer.

### 5.4 Document-Specific Terms

Term	Definition
<b>External Interface</b>	A signal or bus that crosses the STREAM subsystem boundary.
<b>Internal Interface</b>	A signal or bus between blocks within STREAM, not visible externally.
<b>Configuration Register</b>	An APB-accessible register for controlling STREAM behavior.
<b>Status Register</b>	An APB-readable register reporting STREAM operational state.

Last Updated: 2026-01-03

## 6 Use Cases

### 6.1 Primary Use Cases

#### 6.1.1 Memory-to-Memory Data Movement

**Description:** Transfer data between two memory regions without CPU involvement.

**Operation:** 1. Software prepares descriptor(s) in memory with source/destination addresses 2. Software writes first descriptor address to channel control register 3. STREAM autonomously fetches descriptor and executes transfer 4. Optional interrupt notifies software of completion

**Key Benefits:** - CPU offloading for bulk data transfers - Deterministic transfer timing - Support for large transfer sizes via descriptor chaining

---

#### 6.1.2 Scatter-Gather Operations

**Description:** Collect data from multiple non-contiguous source regions into a contiguous destination, or distribute contiguous source data to multiple destinations.

**Gather Example:**

```
Descriptor 0: src=0x1000, dst=0x8000, len=64 beats
Descriptor 1: src=0x3000, dst=0x8100, len=32 beats (chained)
Descriptor 2: src=0x5000, dst=0x8180, len=16 beats (chained, last)
```

**Scatter Example:**

```
Descriptor 0: src=0x8000, dst=0x1000, len=64 beats
Descriptor 1: src=0x8100, dst=0x3000, len=32 beats (chained)
Descriptor 2: src=0x8180, dst=0x5000, len=16 beats (chained, last)
```

---

#### 6.1.3 Multi-Channel Concurrent Transfers

**Description:** Execute multiple independent transfer operations simultaneously.

**Operation:** - Up to 8 channels operate concurrently - Each channel processes its own descriptor chain - Shared resources (SRAM, AXI masters) arbitrated by priority

**Use Cases:** - Parallel DMA for multi-stream applications - Priority-based bandwidth allocation - Independent channel error isolation

---

## **6.2 Secondary Use Cases**

---

### **6.2.1 Buffer Management**

**Description:** Move data between system memory and local buffers for processing pipelines.

**Pattern:**

System Memory → STREAM → Processing Buffer → STREAM → System Memory

### **6.2.2 Memory Initialization**

**Description:** Fill memory regions with pattern data using chained descriptors.

**Note:** Requires source buffer containing pattern data.

---

## **6.3 Target Applications**

---

Application	Channel Usage	Typical Transfer Size
Video Frame Copy	1-2 channels	1-8 MB per frame
Audio Buffer Management	1 channel	4-64 KB per buffer
Data Collection	4-8 channels	Variable
Memory Test Patterns	1-4 channels	System-dependent

**Last Updated:** 2026-01-03

## **7 Key Features**

### **7.1 Feature Summary**

---

Feature	Specification
<b>Channels</b>	8 independent channels
<b>Descriptor Size</b>	256 bits (32 bytes)
<b>Data Width</b>	Parameterizable (default 512 bits)
<b>Address Width</b>	64 bits

Feature	Specification
<b>Maximum Burst</b>	256 beats (AXI4)
<b>Chaining</b>	Unlimited descriptor chain depth
<b>Arbitration</b>	Priority-based with round-robin

## 7.2 Descriptor-Based Operation

---

### 7.2.1 Single-Write Kick-off

A single APB write to a channel's control register initiates an entire transfer sequence:

1. Write descriptor address to CHn\_CTRL register
2. STREAM automatically fetches and processes descriptor
3. Chained descriptors processed without software intervention
4. Completion indicated via status register and optional interrupt

### 7.2.2 Autonomous Chaining

STREAM follows next\_descriptor\_ptr automatically:

- **Non-zero pointer:** Fetch next descriptor, continue processing
  - **Zero pointer:** Chain complete, transition to idle
  - **Last flag:** Explicit chain termination regardless of pointer
- 

## 7.3 Multi-Channel Architecture

---

### 7.3.1 Channel Independence

Each channel maintains:

- Independent FSM state
- Separate descriptor chain pointer
- Individual error status
- Private completion interrupt

### 7.3.2 Resource Sharing

All channels share:

- Descriptor fetch AXI master
- Data read AXI master
- Data write AXI master
- Internal SRAM buffer
- MonBus reporter

### 7.3.3 Priority-Based Arbitration

- 8-bit priority field in each descriptor
  - Higher priority descriptors serviced first
  - Round-robin within same priority level
  - Anti-starvation timeout mechanism
- 

## 7.4 Data Path Features

---

### 7.4.1 Parameterizable Data Width

Compile-time configurable data width:

Parameter	Description	Typical Values
DATA_WIDTH	AXI data bus width	128, 256, 512 bits
SRAM_DEPTH	Internal buffer depth	1024, 2048, 4096 entries

### 7.4.2 Aligned Address Requirement

[CONST-1] All source and destination addresses must be aligned to data width.

Data Width	Alignment Requirement
128 bits	16-byte aligned
256 bits	32-byte aligned
512 bits	64-byte aligned

This simplification eliminates complex alignment fixup logic for tutorial clarity.

---

## 7.5 Monitoring and Debug

---

### 7.5.1 MonBus Integration

64-bit monitor packets for:

- Transfer initiation events
- Descriptor fetch events
- Transfer completion events
- Error condition reporting
- Performance metrics

### 7.5.2 Status Visibility

APB-readable status registers provide:

- Channel state (idle, active, error)
- Descriptor count per channel
- Error codes and flags
- Performance counters (optional)

---

## 7.6 Error Handling

---

### 7.6.1 Error Detection

Error Type	Detection	Response
AXI read error	RRESP != OKAY	Stop chain, set error flag
AXI write error	BRESP != OKAY	Stop chain, set error flag
Invalid descriptor	Validation failure	Stop chain, set error flag
Address out of range	Limit check	Stop chain, set error flag

### 7.6.2 Error Isolation

Errors in one channel do not affect other channels. Each channel has independent error status and recovery.

---

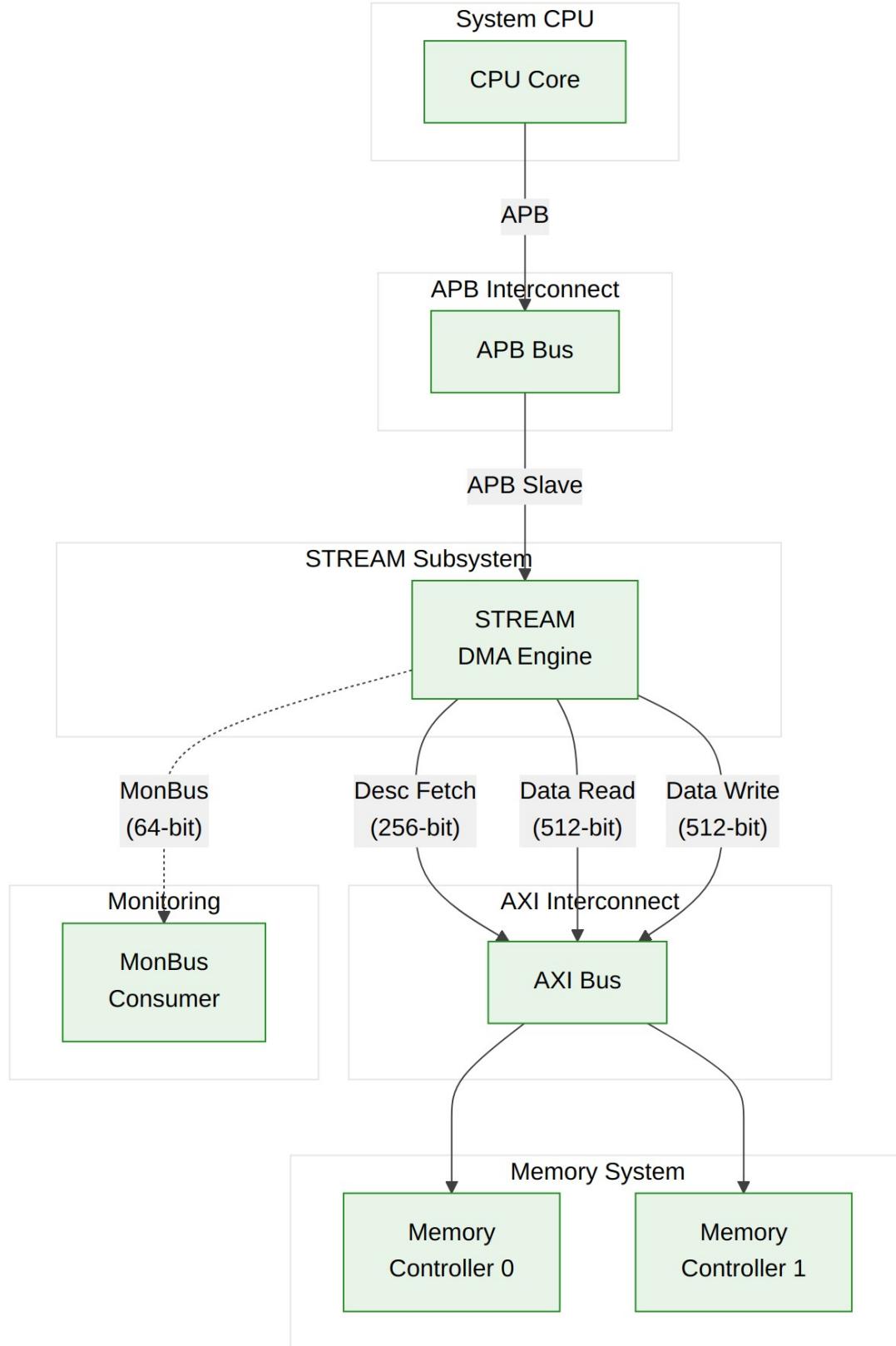
**Last Updated:** 2026-01-03

## 8 System Context

### 8.1 System Integration Overview

---

STREAM integrates into a typical SoC as a peripheral subsystem with the following connections:



## *System Context Diagram*

Source: [01\\_system\\_context.mmd](#)

---

## **8.2 External Connections**

### **8.2.1 Configuration Path**

Connection	Protocol	Description
APB Slave	APB3/APB4	Configuration and control interface

**Functions:** - Channel kick-off (write descriptor address) - Status monitoring - Interrupt control - Error handling

### **8.2.2 Data Paths**

Connection	Protocol	Width	Description
Descriptor Master	AXI4	256-bit	Fetch descriptors from memory
Read Data Master	AXI4	Param	Read source data from memory
Write Data Master	AXI4	Param	Write destination data to memory

### **8.2.3 Monitoring Path**

Connection	Protocol	Width	Description
MonBus Master	MonBus	64-bit	Debug/trace packet output

## 8.3 Clock and Reset

---

### 8.3.1 Clock Domain

STREAM operates in a single clock domain:

Signal	Description
aclk	Primary system clock, all interfaces synchronous
aresetn	Active-low asynchronous reset

**[INTEG-1]** All external interfaces must be synchronous to aclk.

### 8.3.2 Reset Requirements

- Asynchronous assertion, synchronous de-assertion
  - Minimum reset pulse width: 2 clock cycles
  - All state machines return to idle on reset
  - SRAM contents undefined after reset
- 

## 8.4 Memory Map Requirements

---

### 8.4.1 Descriptor Memory

Descriptors must be placed in memory accessible via the descriptor fetch AXI master:

- **Alignment:** 32-byte aligned (256-bit descriptor size)
- **Accessibility:** Read access required
- **Coherency:** Software must ensure descriptors are coherent before kick-off

### 8.4.2 Data Memory

Source and destination regions must be accessible via the appropriate AXI master:

- **Source:** Read access via read data master
  - **Destination:** Write access via write data master
  - **Alignment:** Per data width (see Key Features)
-

## 8.5 Interrupt Integration

---

### 8.5.1 Interrupt Signals

Signal	Description
irq[7:0]	Per-channel completion/error interrupt
irq_combined	OR of all channel interrupts

### 8.5.2 Interrupt Conditions

- Descriptor with interrupt flag completes successfully
  - Channel encounters error condition
  - Software-triggered interrupt (via control register)
- 

## 8.6 Power Considerations

---

### 8.6.1 Clock Gating

STREAM supports clock gating when all channels are idle:

- Monitor `all_channels_idle` status bit
- External clock gate can disable `aclk` when idle
- Re-enable clock before writing kick-off register

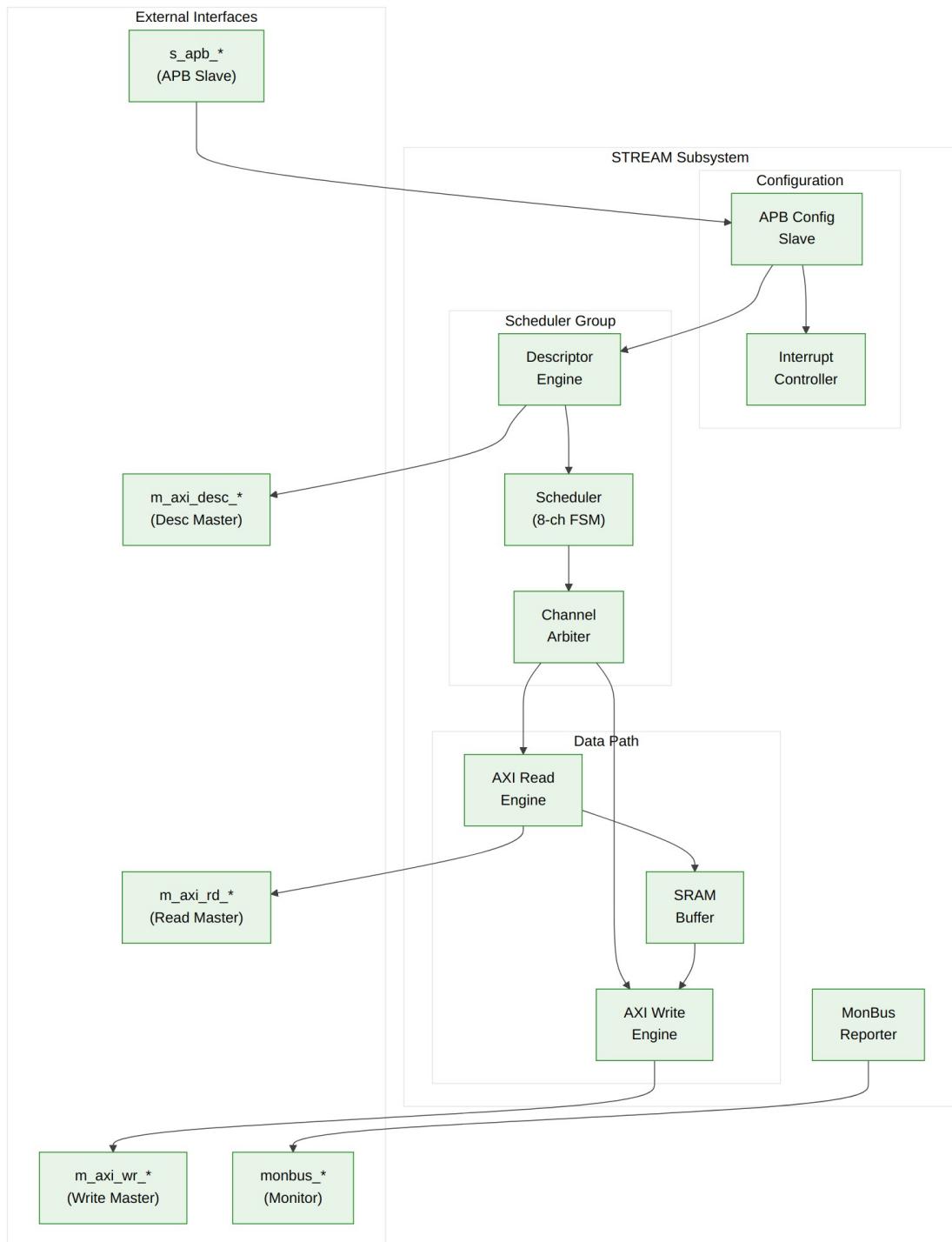
### 8.6.2 Power States

State	Channels	Clock	Power
Active	$\geq 1$ busy	Running	Full
Idle	All idle	Running	Reduced (no switching)
Gated	All idle	Stopped	Minimal (leakage only)

**Last Updated:** 2026-01-03

## 9 Block Diagram

### 9.1 Top-Level Architecture



## *STREAM Block Diagram*

Source: [02\\_block\\_diagram.mmd](#)

---

## **9.2 Functional Block Summary**

Block	Function	Key Interfaces
<b>APB Config Slave</b>	Channel control registers, status	APB slave, kick-off signals
<b>Descriptor Engine</b>	Fetch and parse descriptors	AXI read, descriptor output
<b>Scheduler</b>	Coordinate transfer phases	Control signals to all blocks
<b>Channel Arbiter</b>	Multi-channel priority arbitration	Grant signals
<b>AXI Read Engine</b>	Read source data to SRAM	AXI read master
<b>SRAM Buffer</b>	Temporary data storage	Read/write ports
<b>AXI Write Engine</b>	Write SRAM data to destination	AXI write master
<b>MonBus Reporter</b>	Generate monitoring packets	MonBus output

## **9.3 Block Interactions**

### **9.3.1 Control Flow**

APB Write -> APB Config -> Descriptor Engine -> Scheduler -> Data Path, with AXI Desc Master fetching descriptors.

### **9.3.2 Data Flow**

Memory -> AXI Read Engine -> SRAM Buffer -> AXI Write Engine -> Memory

### **9.3.3 Monitoring Flow**

All Blocks -> Event Signals -> MonBus Reporter -> MonBus Output

---

## 9.4 Parameterization

---

Parameter	Description	Default	Range
NUM_CHANNELS	Number of DMA channels	8	1-8
DATA_WIDTH	AXI data bus width	512	128, 256, 512
ADDR_WIDTH	Address width	64	32, 64
SRAM_DEPTH	Buffer depth in entries	4096	256-8192
DESC_FIFO_DEPTH	Descriptor FIFO depth	4	2-8

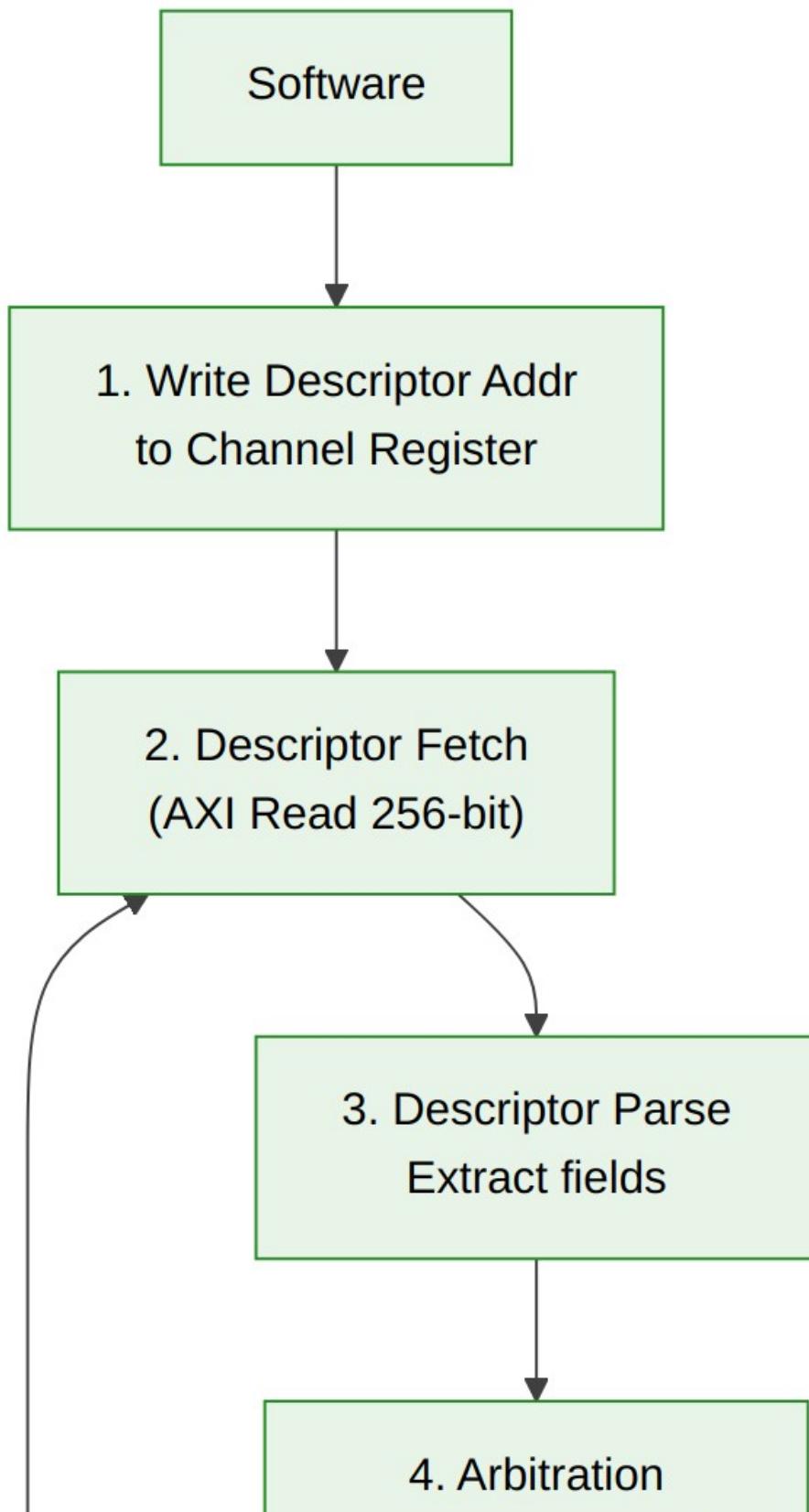
**Last Updated:** 2026-01-03

## 10 Data Flow

### 10.1 Transfer Sequence Overview

---

A complete STREAM transfer follows this sequence:



## *Transfer Sequence Flowchart*

Source: [03\\_data\\_flow.mmd](#)

---

## **10.2 Phase Details**

---

### **10.2.1 Phase 1: Kick-off**

Software writes the first descriptor address to the channel control register:

- APB write triggers channel state transition from IDLE to ACTIVE
- Descriptor address stored in channel's descriptor pointer register
- Kick-off signal sent to Descriptor Engine

### **10.2.2 Phase 2: Descriptor Fetch**

Descriptor Engine issues AXI read transaction:

- Address: From kick-off or next\_descriptor\_ptr
- Size: 256 bits (single beat or 2-beat depending on bus width)
- Response: Descriptor data captured on valid read data

### **10.2.3 Phase 3: Descriptor Parse**

Descriptor fields extracted and validated:

Field	Validation
valid	Must be 1
src_addr	Alignment check, range check
dst_addr	Alignment check, range check
length	Non-zero, within limits

### **10.2.4 Phase 4: Arbitration**

Channel requests access to shared data path:

- Multiple channels may be active simultaneously
- Arbiter grants based on priority and fairness
- Granted channel proceeds to read phase

### **10.2.5 Phase 5: Read Phase**

AXI Read Engine fetches source data:

- Issues AXI read bursts to source address
- Data written to SRAM buffer
- Continues until length beats transferred
- Read/write to SRAM can overlap (ping-pong or streaming)

### **10.2.6 Phase 6: Write Phase**

AXI Write Engine sends data to destination:

- Reads data from SRAM buffer
- Issues AXI write bursts to destination address
- Awaits write responses
- Continues until all data written

### **10.2.7 Phase 7: Chain Check**

Scheduler determines next action:

- If `last == 1` or `next_descriptor_ptr == 0`: Chain complete
  - Otherwise: Fetch next descriptor (return to Phase 2)
- 

## **10.3 Concurrent Operation**

---

### **10.3.1 Multi-Channel Concurrency**

Multiple channels can be at different phases simultaneously. For example: - Ch0: Kick -> Fetch -> Read -> Write -> Complete - Ch1: (offset) Kick -> Fetch -> Arb Wait -> Read -> Write - Ch2: (offset) Kick -> Fetch -> Arb Wait -> Read

### **10.3.2 Pipeline Overlap**

Within a single channel, read and write phases can overlap. As read beats 0-63 complete, write beats 0-63 can begin while read continues with beats 64-127.

The SRAM buffer enables this overlap by storing in-flight data.

---

## **10.4 Data Path Bandwidth**

---

### **10.4.1 Theoretical Maximum**

With 512-bit data width at 200 MHz:

- Per-direction:  $512 \text{ bits} \times 200 \text{ MHz} = 102.4 \text{ Gbps} = 12.8 \text{ GB/s}$
- Bidirectional: 25.6 GB/s (simultaneous read + write)

### **10.4.2 Practical Considerations**

Actual throughput affected by:

- AXI arbitration latency
  - Memory access latency
  - SRAM buffer depth (limits outstanding data)
  - Channel arbitration overhead
- 

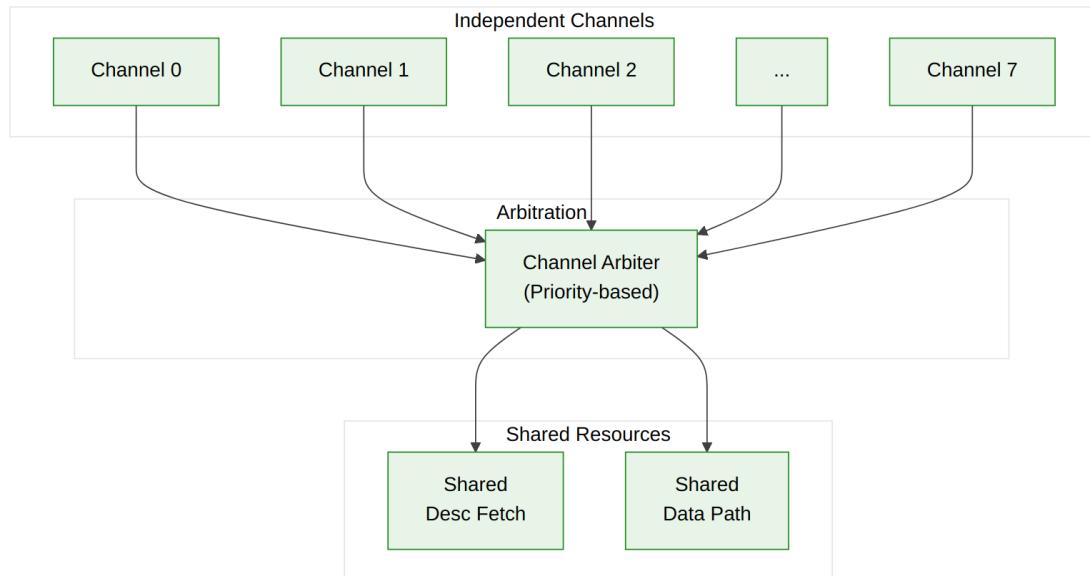
**Last Updated:** 2026-01-03

## **11 Multi-Channel Architecture**

### **11.1 Channel Organization**

---

STREAM supports 8 independent DMA channels, each capable of processing its own descriptor chain:



### *Channel Architecture*

Source: [04\\_channel\\_architecture.mmd](#)

---

## 11.2 Per-Channel Resources

---

Each channel maintains independent state:

Resource	Description	Size
<b>FSM State</b>	Current operation phase	4 bits
<b>Descriptor Pointer</b>	Current/next descriptor address	64 bits
<b>Transfer Counter</b>	Beats remaining in current transfer	32 bits
<b>Error Status</b>	Error flags and codes	8 bits
<b>Priority</b>	Current descriptor priority	8 bits

## 11.3 Shared Resources

---

All channels share the following resources:

Resource	Arbitration	Capacity
<b>Descriptor AXI Master</b>	Round-robin	1 outstanding transaction
<b>Data Read AXI Master</b>	Priority-based	Configurable outstanding
<b>Data Write AXI Master</b>	Priority-based	Configurable outstanding
<b>SRAM Buffer</b>	Partitioned by channel	SRAM_DEPTH entries
<b>MonBus Reporter</b>	FIFO-based	64-entry FIFO

## 11.4 Arbitration Scheme

---

### 11.4.1 Priority-Based Arbitration

Descriptor priority field (8 bits) determines service order:

- **Priority 255:** Highest priority
- **Priority 0:** Lowest priority
- **Same Priority:** Round-robin among equal priority

### 11.4.2 Fairness Mechanism

To prevent starvation:

- Maximum service time per grant (configurable)
- Anti-starvation counter promotes low-priority after timeout
- Emergency preemption for high-priority arrivals (optional)

### 11.4.3 Arbitration Points

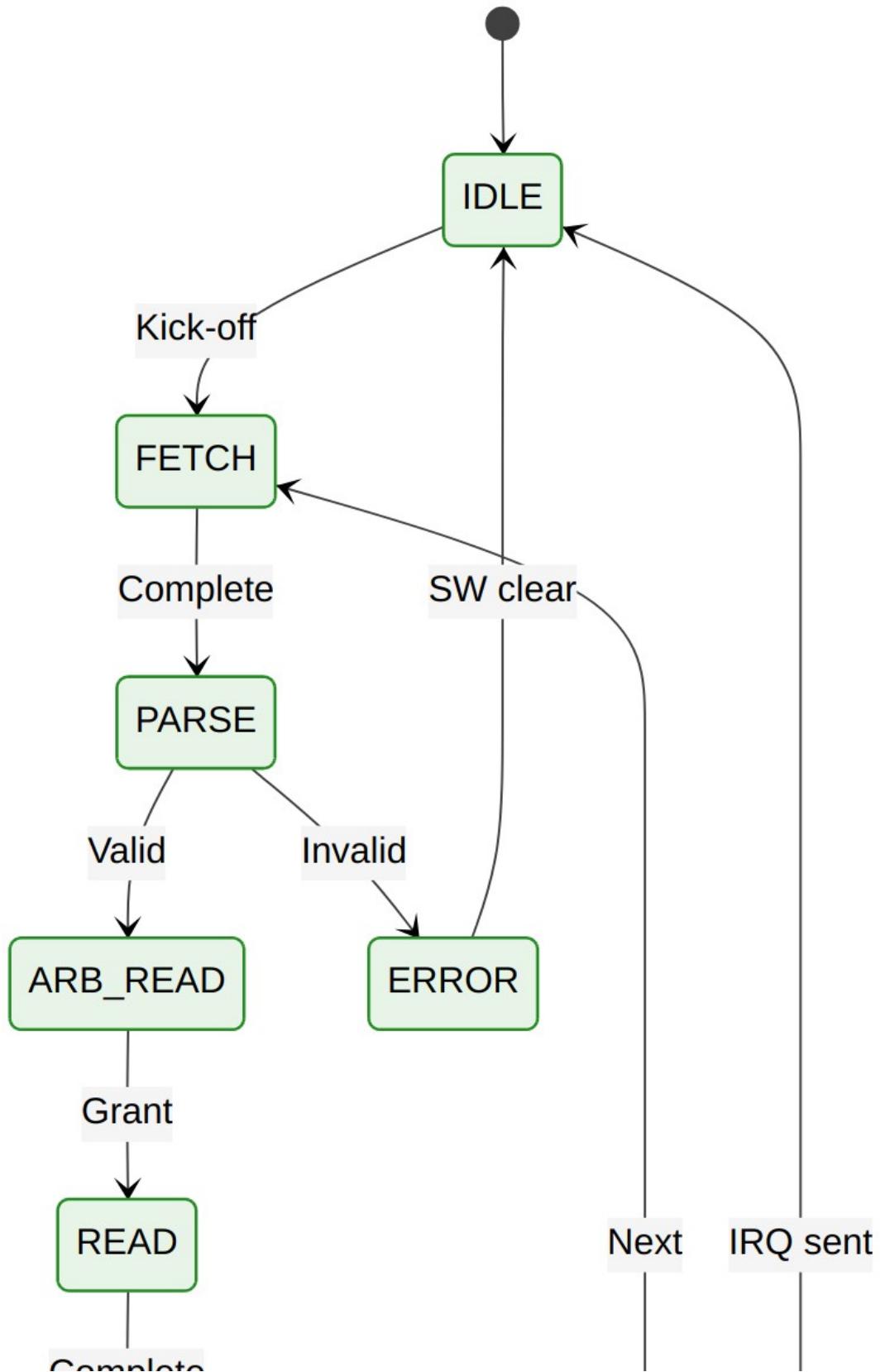
Arbitration occurs at:

1. **Descriptor Fetch:** When channel needs to fetch descriptor
  2. **Read Phase Start:** When channel ready to read source data
  3. **Write Phase Start:** When channel ready to write destination
- 

## 11.5 Channel States

---

Each channel FSM traverses these states:



## *Channel FSM State Diagram*

Source: [05\\_channel\\_fsm.mmd](#)

State	Description	Transitions
<b>IDLE</b>	No active transfer	Kick-off -> FETCH
<b>FETCH</b>	Fetching descriptor	Complete -> PARSE
<b>PARSE</b>	Validating descriptor	Valid -> ARB_READ, Invalid -> ERROR
<b>ARB_READ</b>	Awaiting read grant	Grant -> READ
<b>READ</b>	Reading source data	Complete -> ARB_WRITE
<b>ARB_WRITE</b>	Awaiting write grant	Grant -> WRITE
<b>WRITE</b>	Writing destination	Complete -> CHAIN
<b>CHAIN</b>	Checking for next	Next -> FETCH, Last -> COMPLETE
<b>COMPLETE</b>	Transfer done	IRQ sent -> IDLE
<b>ERROR</b>	Error condition	SW clear -> IDLE

## **11.6 SRAM Partitioning**

The shared SRAM buffer is partitioned among active channels:

### **11.6.1 Static Partitioning (Default)**

SRAM Buffer (4096 entries) is divided equally among 8 channels:

Partition	Entries
Channel 0	512
Channel 1	512
Channel 2	512
Channel 3	512
Channel 4	512
Channel 5	512
Channel 6	512
Channel 7	512

## 11.6.2 Dynamic Allocation (Optional)

When fewer than 8 channels are active, partitions can be expanded:

Active Channels	Entries Per Channel
1	4096
2	2048
4	1024
8	512

## 11.7 Error Isolation

---

Errors are isolated to the affected channel:

- Error in Channel N does not affect Channels 0..(N-1), (N+1)..7
  - Shared resource errors logged but don't block other channels
  - Software must clear error and re-kick to resume
- 

Last Updated: 2026-01-03

# 12 AXI Master Interfaces

## 12.1 Overview

---

STREAM includes three AXI4 master interfaces for memory access:

Interface	Purpose	Data Width	Address Width
<code>m_axi_desc_*</code>	Descriptor fetch	256 bits	64 bits
<code>m_axi_rd_*</code>	Source data read	Parameterizable	64 bits
<code>m_axi_wr_*</code>	Destination write	Parameterizable	64 bits

## 12.2 Descriptor Fetch Master

---

### 12.2.1 Signal Summary

Signal	Direction	Width	Description
m_axi_desc_ar_id	Output	ID_WIDTH	Read transaction ID
m_axi_desc_ar_addr	Output	64	Read address
m_axi_desc_ar_len	Output	8	Burst length (0 = 1 beat)
m_axi_desc_ar_size	Output	3	Beat size (5 = 32 bytes)
m_axi_desc_ar_burst	Output	2	Burst type (INCR)
m_axi_desc_ar_valid	Output	1	Address valid
m_axi_desc_ar_ready	Input	1	Address ready
m_axi_desc_rd_id	Input	ID_WIDTH	Read data ID
m_axi_desc_rd_ata	Input	256	Read data
m_axi_desc_rr_esp	Input	2	Read response
m_axi_desc_rl_ast	Input	1	Last beat
m_axi_desc_rv_alid	Input	1	Read data valid
m_axi_desc_rr_ready	Output	1	Read data ready

### 12.2.2 Transaction Characteristics

Characteristic	Value	Notes
<b>Burst Length</b>	1 beat	Single 256-bit descriptor
<b>Burst Type</b>	INCR	Fixed increment
<b>Outstanding</b>	1-2	Configurable

Characteristic	Value	Notes
<b>Ordering</b>	In-order	Single ID used

### 12.2.3 Address Alignment

Descriptor addresses must be 32-byte aligned (256-bit boundary).

---

## 12.3 Data Read Master

### 12.3.1 Signal Summary

Signal	Direction	Width	Description
m_axi_rd_arid	Output	ID_WIDTH	Read transaction ID
m_axi_rd_arad dr	Output	64	Read address
m_axi_rd_arle n	Output	8	Burst length
m_axi_rd_arsi ze	Output	3	Beat size
m_axi_rd_arbu rst	Output	2	Burst type (INCR)
m_axi_rd_arva lid	Output	1	Address valid
m_axi_rd_arre ady	Input	1	Address ready
m_axi_rd_rid	Input	ID_WIDTH	Read data ID
m_axi_rd_rdat a	Input	DATA_WIDTH	Read data
m_axi_rd_rres p	Input	2	Read response
m_axi_rd_rlas t	Input	1	Last beat
m_axi_rd_rval id	Input	1	Read data valid
m_axi_rd_rrea dy	Output	1	Read data ready

### 12.3.2 Transaction Characteristics

Characteristic	Value	Notes
<b>Burst Length</b>	1-256 beats	AXI4 maximum
<b>Burst Type</b>	INCR	Fixed increment
<b>Outstanding</b>	Configurable	4-16 typical
<b>Ordering</b>	In-order	Channel-based ID tagging

### 12.3.3 Burst Generation

Large transfers are broken into maximum-length bursts:

Transfer Size	Burst Count	Burst Lengths
256 beats	1	256
512 beats	2	256, 256
300 beats	2	256, 44

## 12.4 Data Write Master

---

### 12.4.1 Signal Summary

Signal	Direction	Width	Description
m_axi_wr_awid	Output	ID_WIDTH	Write transaction ID
m_axi_wr_awad <sub>dr</sub>	Output	64	Write address
m_axi_wr_awle <sub>n</sub>	Output	8	Burst length
m_axi_wr_awsi <sub>ze</sub>	Output	3	Beat size
m_axi_wr_awbu <sub>rst</sub>	Output	2	Burst type (INCR)
m_axi_wr_awva <sub>lid</sub>	Output	1	Address valid
m_axi_wr_awre <sub>ady</sub>	Input	1	Address ready
m_axi_wr_wdat <sub>a</sub>	Output	DATA_WIDTH	Write data

Signal	Direction	Width	Description
m_axi_wr_wstrb	Output	DATA_WIDTH/8	Byte strobes
m_axi_wr_wlast	Output	1	Last beat
m_axi_wr_wvalid	Output	1	Write data valid
m_axi_wr_wready	Input	1	Write data ready
m_axi_wr_bid	Input	ID_WIDTH	Response ID
m_axi_wr_bresp	Input	2	Write response
m_axi_wr_bvalid	Input	1	Response valid
m_axi_wr_bready	Output	1	Response ready

#### 12.4.2 Transaction Characteristics

Characteristic	Value	Notes
<b>Burst Length</b>	1-256 beats	Matches read bursts
<b>Burst Type</b>	INCR	Fixed increment
<b>Outstanding</b>	Configurable	4-16 typical
<b>Write Strobes</b>	All-ones	Aligned transfers only

## 12.5 Common AXI Attributes

---

#### 12.5.1 Supported Features

Feature	Descriptor	Read	Write
INCR Burst	Yes	Yes	Yes
WRAP Burst	No	No	No
FIXED Burst	No	No	No
Exclusive	No	No	No
Locked	No	No	No

### 12.5.2 Error Handling

- SLVERR/DECERR on any channel stops the current transfer
  - Error logged in channel status register
  - Subsequent descriptors in chain not processed
  - Software intervention required to clear and resume
- 

**Last Updated:** 2026-01-03

## 13 APB Configuration Interface

### 13.1 Overview

---

STREAM provides an APB slave interface for software configuration and control. The interface supports:

- Channel kick-off (write descriptor address to start transfer)
  - Status monitoring
  - Interrupt control
  - Error handling
- 

### 13.2 Signal Summary

---

Signal	Direction	Width	Description
s_apb_paddr	Input	12	Address bus
s_apb_psel	Input	1	Peripheral select
s_apb_penable	Input	1	Enable phase
s_apb_pwrite	Input	1	Write enable
s_apb_pwdata	Input	32	Write data
s_apb_pstrb	Input	4	Byte strobes
s_apb_pready	Output	1	Ready signal
s_apb_prdata	Output	32	Read data
s_apb_pslverr	Output	1	Slave error

### 13.3 Register Map Overview

Offset	Register	Access	Description
0x000	CTRL	RW	Global control register
0x004	STATUS	RO	Global status register
0x008	IRQ_EN	RW	Interrupt enable mask
0x00C	IRQ_STATUS	RW1C	Interrupt status
0x010	ERR_STATUS	RO	Error status
0x014	ERR_ADDR	RO	Error address capture
0x020-0x03C	Reserved	-	Reserved
0x040	CH0_CTRL	RW	Channel 0 control
0x044	CH0_STATUS	RO	Channel 0 status
0x048	CH0_DESC_PTR	RO	Channel 0 current descriptor
0x04C	Reserved	-	Reserved
0x050	CH1_CTRL	RW	Channel 1 control
...	...	...	...
0x0B0	CH7_CTRL	RW	Channel 7 control
0x0B4	CH7_STATUS	RO	Channel 7 status
0x0B8	CH7_DESC_PTR	RO	Channel 7 current descriptor

## 13.4 Key Registers

---

### 13.4.1 Global Control (CTRL) - 0x000

Bits	Field	Access	Description
[0]	ENABLE	RW	Global enable
[1]	SOFT_RESET	RW	Soft reset (self-clearing)
[31:2]	Reserved	-	Reserved

### 13.4.2 Global Status (STATUS) - 0x004

Bits	Field	Access	Description
[0]	BUSY	RO	Any channel active
[7:1]	Reserved	-	Reserved
[15:8]	CH_ACTIVE	RO	Per-channel active mask
[23:16]	CH_ERROR	RO	Per-channel error mask
[31:24]	Reserved	-	Reserved

### 13.4.3 Channel Control (CHn\_CTRL) - 0x040 + n\*0x10

Bits	Field	Access	Description
[31:0]	DESC_ADDR	RW	Descriptor address (kick-off on write)

Writing to CHn\_CTRL initiates a transfer: 1. Descriptor address stored 2. Channel transitions from IDLE to FETCH 3. PREADY de-asserted until channel accepts kick-off

### 13.4.4 Channel Status (CHn\_STATUS) - 0x044 + n\*0x10

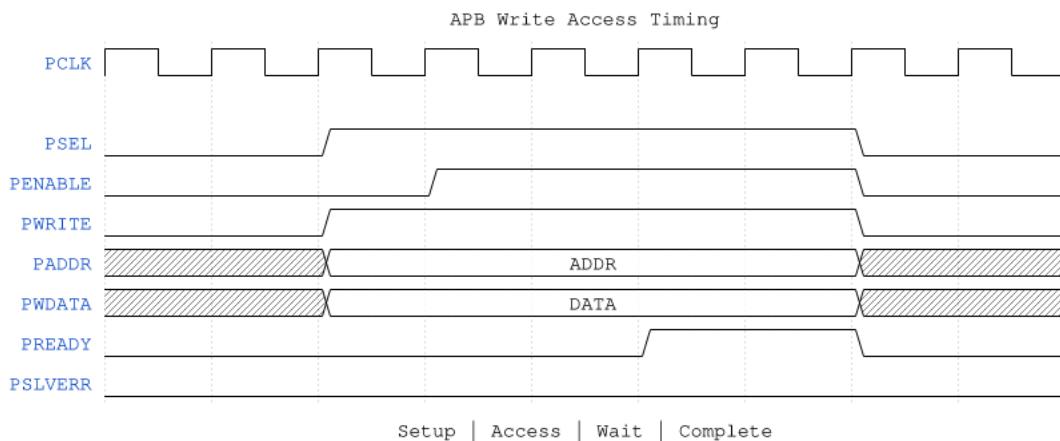
Bits	Field	Access	Description
[3:0]	STATE	RO	Channel FSM state
[4]	IDLE	RO	Channel is idle
[5]	BUSY	RO	Channel is processing

Bits	Field	Access	Description
[6]	ERROR	RO	Error condition
[7]	COMPLETE	RO	Last transfer complete
[15:8]	DESC_COUNT	RO	Descriptors processed
[23:16]	ERR_CODE	RO	Error code if ERROR set
[31:24]	Reserved	-	Reserved

## 13.5 Access Timing

---

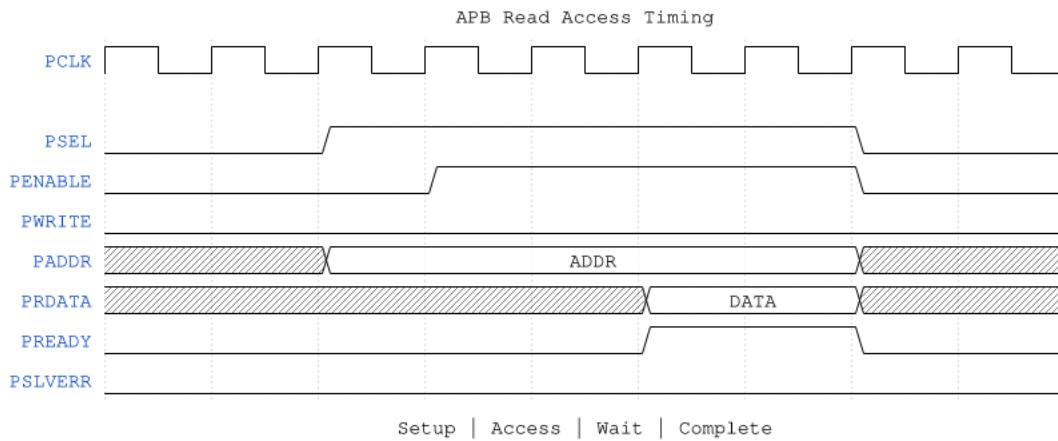
### 13.5.1 Write Access



*APB Write Access Timing*

Source: [apb\\_write\\_access.json](#)

### 13.5.2 Read Access



*APB Read Access Timing*

Source: [apb\\_read\\_access.json](#)

---

## 13.6 Kick-off Blocking

---

When software writes to CHn\_CTRL:

- If channel is IDLE: Immediate accept, PREADY asserted
  - If channel is BUSY: PREADY held low until channel completes
  - Prevents descriptor pointer corruption during active transfer
- 

Last Updated: 2026-01-03

## 14 Monitoring Interface

---

### 14.1 Overview

---

STREAM provides a 64-bit Monitor Bus (MonBus) interface for debug and performance monitoring. This interface streams event packets capturing key operational events.

---

## 14.2 Signal Summary

---

Signal	Direction	Width	Description
monbus_pkt_va_lid	Output	1	Packet valid
monbus_pkt_ready	Input	1	Downstream ready
monbus_pkt_data	Output	64	Packet data

## 14.3 Packet Format

---

Each 64-bit MonBus packet follows this format:

Bits	Field	Description
[63:56]	SUBSYSTEM_ID	STREAM subsystem identifier
[55:48]	EVENT_TYPE	Event category
[47:40]	EVENT_CODE	Specific event within category
[39:32]	CHANNEL_ID	Associated channel (0-7)
[31:0]	PAYLOAD	Event-specific data

## 14.4 Event Types

---

### 14.4.1 Transfer Events (EVENT\_TYPE = 0x01)

EVENT_CODE	Event	Payload
0x01	KICKOFF	Descriptor address [31:0]
0x02	DESC_FETCH	Descriptor address [31:0]
0x03	DESC_VALID	Descriptor fields summary
0x04	READ_START	Source address

<b>EVENT_CODE</b>	<b>Event</b>	<b>Payload</b>
		[31:0]
0x05	READ_DONE	Beat count
0x06	WRITE_START	Destination address
		[31:0]
0x07	WRITE_DONE	Beat count
0x08	CHAIN_NEXT	Next descriptor address
0x09	COMPLETE	Total descriptors processed

#### 14.4.2 Error Events (EVENT\_TYPE = 0x02)

<b>EVENT_CODE</b>	<b>Event</b>	<b>Payload</b>
0x01	AXI_READ_ERR	AXI response code
0x02	AXI_WRITE_ERR	AXI response code
0x03	DESC_INVALID	Validation failure code
0x04	ADDR_RANGE	Offending address [31:0]
0x05	TIMEOUT	Timeout counter value

#### 14.4.3 Performance Events (EVENT\_TYPE = 0x03)

<b>EVENT_CODE</b>	<b>Event</b>	<b>Payload</b>
0x01	ARB_WAIT	Wait cycles
0x02	READ_LATENCY	First beat latency
0x03	WRITE_LATENCY	First response latency
0x04	THROUGHPUT	Bytes transferred

## 14.5 Event Generation

---

### 14.5.1 Configurable Events

Events can be enabled/disabled via APB registers:

Register	Bit	Event Category
MONBUS_CFG	[0]	Transfer events
MONBUS_CFG	[1]	Error events
MONBUS_CFG	[2]	Performance events
MONBUS_CFG	[7:4]	Channel mask

### 14.5.2 Event Priority

When multiple events occur simultaneously:

1. Error events (highest priority)
  2. Transfer completion events
  3. Transfer start events
  4. Performance events (lowest priority)
- 

## 14.6 Backpressure Handling

---

### 14.6.1 Flow Control

- Events generated when `monbus_pkt_ready` is asserted
- If `monbus_pkt_ready` is low, events are queued in internal FIFO
- FIFO depth: 64 entries (configurable)
- Overflow behavior: Oldest events discarded, overflow counter incremented

### 14.6.2 Recommendations

- Connect MonBus to FIFO with sufficient depth
  - Monitor overflow counter in status register
  - Size downstream FIFO based on expected event rate
- 

## 14.7 Typical MonBus Downstream

---

STREAM MonBus → MonBus FIFO → MonBus Arbiter → System Trace  
(64 deep)      (multi-source)      Interface

---

## 14.8 Example Event Sequence

---

A single-descriptor transfer generates these events:

Time	Event	-
T0	KICKOFF	- Software kicks off channel
T1	DESC_FETCH	- Descriptor fetch initiated
T10	DESC_VALID	- Descriptor parsed successfully
T12	ARB_WAIT	- Waiting for data path grant (if applicable)
T15	READ_START	- AXI read burst initiated
T75	READ_DONE	- All source data read
T80	WRITE_START	- AXI write burst initiated
T140	WRITE_DONE	- All data written, responses received
T141	COMPLETE	- Channel complete, back to IDLE

---

Last Updated: 2026-01-03

## 15 Throughput Targets

### 15.1 Theoretical Maximum

---

#### 15.1.1 Per-Interface Bandwidth

Interface	Data Width	Frequency	Bandwidth
Descriptor Fetch	256 bits	200 MHz	6.4 GB/s
Data Read	512 bits	200 MHz	12.8 GB/s
Data Write	512 bits	200 MHz	12.8 GB/s

#### 15.1.2 Aggregate Bandwidth

- **Bidirectional:** 25.6 GB/s (simultaneous read + write)
  - **Descriptor Overhead:** ~0.1% for large transfers
-

## 15.2 Practical Throughput

---

### 15.2.1 Single-Channel Performance

Scenario	Expected Efficiency	Effective Bandwidth
Large sequential (>1MB)	>95%	>12.0 GB/s
Medium sequential (64KB-1MB)	85-95%	10.8-12.0 GB/s
Small sequential (4KB-64KB)	70-85%	9.0-10.8 GB/s
Very small (<4KB)	40-70%	5.0-9.0 GB/s

### 15.2.2 Multi-Channel Performance

Active Channels	Per-Channel BW	Aggregate BW
1	12.8 GB/s	12.8 GB/s
2	6.4 GB/s	12.8 GB/s
4	3.2 GB/s	12.8 GB/s
8	1.6 GB/s	12.8 GB/s

**Note:** Aggregate limited by shared AXI masters. Individual channel throughput scales inversely with active channel count.

---

## 15.3 Throughput Limiting Factors

---

### 15.3.1 Memory System Factors

Factor	Impact	Mitigation
Memory latency	Reduces efficiency for small transfers	Use deeper outstanding
Memory bandwidth	Hard limit	Match STREAM to memory capability
Interconnect contention	Variable impact	Priority configuration

### 15.3.2 STREAM Internal Factors

Factor	Impact	Mitigation
Descriptor fetch overhead	Fixed per descriptor	Use longer transfers
SRAM depth	Limits outstanding	Configure adequate depth
Arbitration overhead	Multi-channel penalty	Reduce active channels
Burst splitting	4KB boundary crossings	Align transfers

## 15.4 Performance Targets

---

### 15.4.1 Design Targets

Metric	Target	Condition
<b>Peak throughput</b>	12.8 GB/s	Single direction, ideal
<b>Sustained throughput</b>	>11.0 GB/s	Large transfers, single channel
<b>Multi-channel aggregate</b>	>10.0 GB/s	8 channels active
<b>Small transfer efficiency</b>	>50%	4KB transfers

### 15.4.2 Verification Criteria

Scenario	Pass Criteria
1MB sequential read	>95% efficiency
1MB sequential write	>95% efficiency
64KB scatter (4x16KB)	>80% efficiency
4KB × 100 descriptors	>60% efficiency

## 15.5 Descriptor Chain Efficiency

---

### 15.5.1 Descriptor Overhead

Transfer Size	Descriptor Fetches	Overhead
64 KB	1	0.05%
256 KB	1	0.01%
1 MB	1	<0.01%
4 KB (100 chained)	100	0.5%

### 15.5.2 Chaining Efficiency

For chained descriptors, next descriptor fetch can overlap with current transfer:

```
Desc 0: [Fetch][-----Transfer 0-----]
Desc 1:           [Fetch][-----Transfer 1-----]
Desc 2:                 [Fetch][-----Transfer 2-----]
```

This pipelining minimizes chaining overhead for continuous transfers.

---

Last Updated: 2026-01-03

## 16 Latency Characteristics

### 16.1 Latency Components

---

#### 16.1.1 End-to-End Latency Breakdown

For a single-descriptor transfer (kick-off to completion):

Phase	Typical Cycles	Notes
APB write processing	2-3	APB transaction
Descriptor fetch	10-50	Memory latency dependent
Descriptor parse	2-4	Fixed pipeline
Arbitration wait	0-100+	Depends on contention
Read phase setup	2-3	AXI AR channel

Phase	Typical Cycles	Notes
Read data transfer	N + 10-50	N beats + memory latency
Write phase setup	2-3	AXI AW channel
Write data transfer	N + 5-20	N beats + response latency
Completion handling	2-4	Status update, IRQ

### 16.1.2 Total Latency Formula

Latency = T\_apb + T\_desc\_fetch + T\_parse + T\_arb + T\_read + T\_write + T\_complete

Where:

T\_apb = 3 cycles (fixed)  
 T\_desc\_fetch = Memory\_latency + 10 cycles  
 T\_parse = 3 cycles (fixed)  
 T\_arb = 0 to N (contention dependent)  
 T\_read = Memory\_latency + (Transfer\_length / Beats\_per\_burst)  
 \* Burst\_overhead + Transfer\_length  
 T\_write = Memory\_latency + Transfer\_length + Response\_latency  
 T\_complete = 3 cycles (fixed)

---

## 16.2 First-Byte Latency

Time from kick-off to first data beat arriving at destination:

Component	Cycles	Cumulative
APB processing	3	3
Descriptor fetch	30	33
Parse + setup	5	38
Read AR accepted	10	48
First read data	20	68
Write AW + first W	10	78
<b>Total</b>	<b>~80</b>	-

At 200 MHz: ~400 ns first-byte latency (typical)

---

## 16.3 Latency by Transfer Size

---

### 16.3.1 Small Transfers

Transfer Size	Total Latency	Efficiency
64 bytes (1 beat)	~100 cycles	1%
256 bytes (4 beats)	~110 cycles	4%
1 KB (16 beats)	~130 cycles	12%
4 KB (64 beats)	~180 cycles	35%

### 16.3.2 Large Transfers

Transfer Size	Total Latency	Efficiency
16 KB (256 beats)	~400 cycles	64%
64 KB (1024 beats)	~1200 cycles	85%
256 KB (4096 beats)	~4400 cycles	93%
1 MB (16384 beats)	~17000 cycles	96%

## 16.4 Latency Variability

---

### 16.4.1 Sources of Variability

Source	Impact	Range
Memory system load	High	10-100+ cycles
AXI interconnect	Medium	5-50 cycles
Channel arbitration	Medium	0-100+ cycles
Descriptor chaining	Low	2-5 cycles

### 16.4.2 Worst-Case Latency

Conditions for worst-case latency:

1. All 8 channels active (maximum arbitration delay)
2. Memory system fully loaded (maximum memory latency)
3. Small transfer size (maximum overhead ratio)
4. Low priority descriptor (maximum wait time)

Under these conditions, first-byte latency could exceed 1000 cycles.

---

## 16.5 Latency Optimization

---

### 16.5.1 Software Recommendations

Recommendation	Benefit
Use larger transfers	Better efficiency
Minimize active channels	Reduce arbitration
Use higher priority for latency-sensitive	Earlier service
Pre-stage descriptors in cache	Faster fetch

### 16.5.2 Hardware Configuration

Setting	Effect
Increase outstanding limit	Hide memory latency
Enable descriptor prefetch	Reduce chaining overhead
Configure larger SRAM	Support longer bursts

---

## 16.6 Interrupt Latency

---

Time from transfer completion to interrupt assertion:

Phase	Cycles
Last write response	0
Completion FSM	2
Status register update	1
Interrupt assertion	1
<b>Total</b>	<b>4 cycles</b>

At 200 MHz: 20 ns from completion to interrupt.

---

**Last Updated:** 2026-01-03

## 17 Resource Estimates

### 17.1 FPGA Resource Summary

Estimates for typical configuration (8 channels, 512-bit data width, 4096-entry SRAM):

Resource	Estimate	Notes
LUTs	15,000 - 25,000	Logic and control
Registers	20,000 - 30,000	Pipeline and state
Block RAM	40 - 60	SRAM buffer
DSP	0	No DSP usage

### 17.2 Resource Breakdown by Block

#### 17.2.1 APB Configuration Slave

Resource	Estimate
LUTs	500 - 800
Registers	400 - 600
BRAM	0

#### 17.2.2 Descriptor Engine

Resource	Estimate
LUTs	1,500 - 2,500
Registers	1,000 - 1,500
BRAM	1 (descriptor FIFO)

#### 17.2.3 Scheduler (8 channels)

Resource	Estimate
LUTs	3,000 - 5,000
Registers	4,000 - 6,000
BRAM	0

#### **17.2.4 Channel Arbiter**

<b>Resource</b>	<b>Estimate</b>
LUTs	500 - 1,000
Registers	200 - 400
BRAM	0

#### **17.2.5 AXI Read Engine**

<b>Resource</b>	<b>Estimate</b>
LUTs	2,000 - 3,500
Registers	3,000 - 4,500
BRAM	0

#### **17.2.6 SRAM Buffer**

<b>Resource</b>	<b>Estimate</b>
LUTs	100 - 200
Registers	100 - 200
BRAM	32 - 64 (depth/width dependent)

#### **17.2.7 AXI Write Engine**

<b>Resource</b>	<b>Estimate</b>
LUTs	2,000 - 3,500
Registers	3,000 - 4,500
BRAM	0

#### **17.2.8 MonBus Reporter**

<b>Resource</b>	<b>Estimate</b>
LUTs	800 - 1,200
Registers	600 - 900
BRAM	1 (event FIFO)

### **17.3 SRAM Sizing**

---

SRAM buffer size depends on configuration:

Configuration	BRAM (18Kb)	BRAM (36Kb)
512b x 1024	16	8
512b x 2048	32	16
512b x 4096	64	32
256b x 4096	32	16
128b x 4096	16	8

## 17.4 Scaling with Parameters

---

### 17.4.1 Channel Count Impact

Channels	LUT Delta	Register Delta
1	Base	Base
2	+500	+800
4	+1,500	+2,400
8	+3,500	+5,600

### 17.4.2 Data Width Impact

Data Width	LUT Delta	Register Delta	BRAM Delta
128 bits	-30%	-40%	-75%
256 bits	-15%	-20%	-50%
512 bits	Base	Base	Base

## 17.5 ASIC Estimates

---

For 28nm technology node (typical):

Metric	Estimate
<b>Gate Count</b>	200K - 350K gates
<b>Area</b>	0.2 - 0.4 mm <sup>2</sup>
<b>SRAM Area</b>	0.1 - 0.2 mm <sup>2</sup> (4096x512)
<b>Total Area</b>	0.3 - 0.6 mm <sup>2</sup>

### 17.5.1 Power Estimates

Condition	Power
Idle (clock gated)	<1 mW
Idle (clock running)	10 - 20 mW
Single channel active	50 - 100 mW
All channels active	150 - 300 mW

## 17.6 Timing Estimates

---

### 17.6.1 Target Frequencies

Technology	Target Fmax
Xilinx Ultrascale+	250 MHz
Intel Agilex	250 MHz
28nm ASIC	400 MHz
16nm ASIC	600 MHz

### 17.6.2 Critical Paths

Path	Concern Level	Mitigation
AXI data path	Low	Fully pipelined
SRAM read/write	Low	Registered outputs
Arbiter decision	Medium	Priority encoder depth
Descriptor parse	Low	Simple combinational

Last Updated: 2026-01-03

## 18 System Requirements

### 18.1 Interface Requirements

#### 18.1.1 APB Interface

Requirement	Specification
<b>Protocol</b>	APB3 or APB4
<b>Data Width</b>	32 bits
<b>Address Width</b>	12 bits minimum
<b>Timing</b>	Synchronous to aclk

#### 18.1.2 AXI Interfaces

Requirement	Descriptor	Read Data	Write Data
<b>Protocol</b>	AXI4	AXI4	AXI4
<b>Data Width</b>	256 bits	Parameterizable	Parameterizable
<b>Address Width</b>	64 bits	64 bits	64 bits
<b>ID Width</b>	4 bits min	4 bits min	4 bits min
<b>Outstanding</b>	2	Configurable	Configurable

#### 18.1.3 MonBus Interface

Requirement	Specification
<b>Data Width</b>	64 bits
<b>Protocol</b>	Valid/Ready streaming
<b>Backpressure</b>	Required (must handle ready deassertion)

## 18.2 Memory Requirements

### 18.2.1 Descriptor Memory

Requirement	Specification
<b>Alignment</b>	32-byte aligned

Requirement	Specification
<b>Access</b>	Read access via descriptor AXI master
<b>Coherency</b>	Software responsibility
<b>Latency</b>	<100 cycles recommended

## 18.2.2 Data Memory

Requirement	Specification
<b>Alignment</b>	Per data width (64-byte for 512-bit)
<b>Access</b>	Read via read master, Write via write master
<b>Bandwidth</b>	Sufficient for target throughput

## 18.3 System Bandwidth Allocation

---

### 18.3.1 AXI Interconnect Requirements

To achieve maximum STREAM throughput:

AXI Master	Required Bandwidth	Priority
Descriptor Fetch	0.1 GB/s	Medium
Data Read	12.8 GB/s	High
Data Write	12.8 GB/s	High

### 18.3.2 Interconnect Recommendations

- Configure STREAM AXI masters with high priority
  - Ensure sufficient outstanding transaction support
  - Consider dedicated paths for high-bandwidth DMA
- 

## 18.4 Interrupt Requirements

---

### 18.4.1 Interrupt Controller

Requirement	Specification
<b>Interrupt Lines</b>	8 (per-channel) or 1 (combined)

Requirement	Specification
Type	Level-sensitive, active-high
Latency	<10 cycles from assertion to controller
<b>18.4.2 Interrupt Handling</b>	
• Software must read IRQ_STATUS to identify source	
• Write-1-to-clear to acknowledge interrupts	
• Channel remains blocked until error cleared (if applicable)	

## 18.5 Address Space Requirements

---

### 18.5.1 STREAM Register Space

Requirement	Specification
Base Address	System-defined
Size	4 KB (0x1000)
Alignment	4 KB aligned

### 18.5.2 Memory Map Constraints

- Descriptor addresses must be accessible via descriptor AXI master
  - Source addresses must be accessible via read AXI master
  - Destination addresses must be accessible via write AXI master
  - All addresses must respect configured address range limits
- 

## 18.6 Software Requirements

---

### 18.6.1 Driver Requirements

Requirement	Description
Descriptor Preparation	Build descriptors in coherent memory
Cache Management	Flush descriptor cache before kick-off
Interrupt Handling	Service interrupts promptly

Requirement	Description
<b>Error Recovery</b>	Clear error status, optionally retry

---

**18.6.2 Programming Sequence**

1. Verify STREAM is enabled (CTRL.ENABLE = 1)
2. Check target channel is idle (CHn\_STATUS.IDLE = 1)
3. Prepare descriptor chain in memory
4. Flush cache (if applicable)
5. Write first descriptor address to CHn\_CTRL
6. Wait for interrupt or poll CHn\_STATUS

---

Last Updated: 2026-01-03

## 19 Clocking and Reset

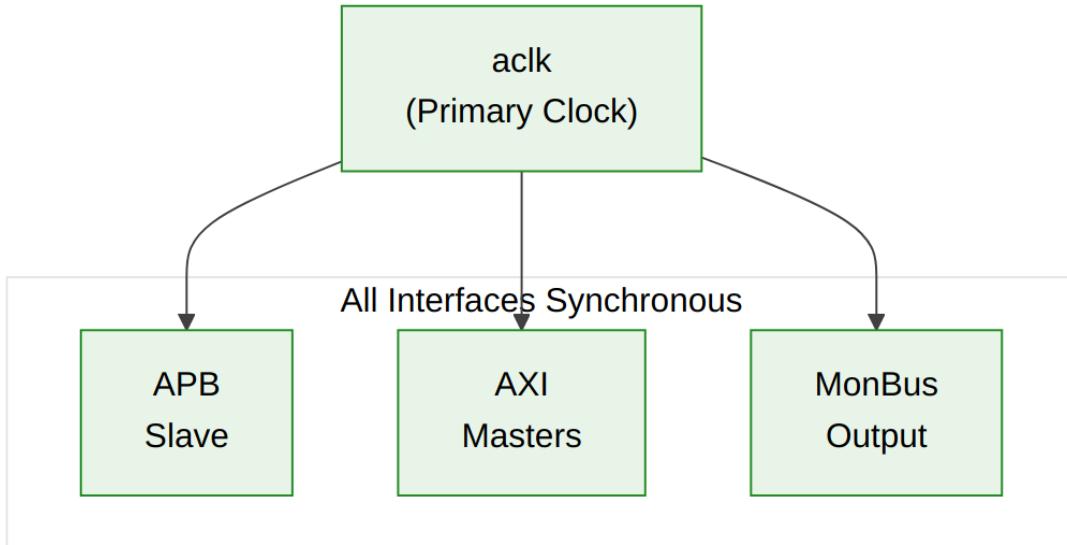
### 19.1 Clock Requirements

#### 19.1.1 Primary Clock (aclk)

Parameter	Requirement
<b>Frequency</b>	100 - 400 MHz (target dependent)
<b>Duty Cycle</b>	40% - 60%
<b>Jitter</b>	<100 ps peak-to-peak
<b>Domain</b>	Single clock domain for all interfaces

#### 19.1.2 Clock Relationship

All STREAM interfaces are synchronous to aclk:



## Clock Distribution

Source: [06\\_clock\\_distribution.mmd](#)

**[INTEG-1]** External interfaces must be synchronous to `aclk`. Clock domain crossing is the system integrator's responsibility.

---

## 19.2 Reset Requirements

### 19.2.1 Reset Signal (`aresetn`)

Parameter	Requirement
<b>Polarity</b>	Active-low
<b>Type</b>	Asynchronous assert, synchronous deassert
<b>Minimum Pulse</b>	2 clock cycles
<b>Glitch Filter</b>	Not required (external synchronizer assumed)

### 19.2.2 Reset Sequence

Reset is asserted asynchronously and de-asserted synchronously:

1. `aresetn` goes low (asynchronous assertion)
2. Reset remains active for minimum 2 clock cycles

3. aresetn goes high on rising edge of aclk (synchronous release)
4. All blocks begin normal operation

### 19.2.3 Reset Behavior

Block	Reset State
APB Config	All registers to default
Scheduler	All channels IDLE
Descriptor Engine	FSM to IDLE, FIFOs empty
AXI Engines	No outstanding transactions
SRAM	Contents undefined
MonBus	FIFO empty, no output

## 19.3 Power Management

---

### 19.3.1 Clock Gating Support

STREAM supports external clock gating when idle:

Signal	Purpose
all_channels_idle	Indicates safe to gate clock

**Clock Gating Sequence:** 1. Check STATUS.BUSY = 0 2. Disable clock via external gate 3. Re-enable clock before any APB access

### 19.3.2 Low Power Recommendations

Technique	Implementation
Clock gating	Gate aclk when all_channels_idle
Power gating	Not recommended (state loss)
Voltage scaling	Supported if timing met

## 19.4 Timing Constraints

---

### 19.4.1 Clock Period Constraints

```
# Example Vivado XDC constraints
create_clock -period 5.000 -name aclk [get_ports aclk]
```

```

# APB interface (same clock)
set_input_delay -clock aclk 1.0 [get_ports s_apb_*]
set_output_delay -clock aclk 1.0 [get_ports s_apb_pready]
set_output_delay -clock aclk 1.0 [get_ports s_apb_prdata*]
set_output_delay -clock aclk 1.0 [get_ports s_apb_pslverr]

# AXI interfaces (same clock)
set_input_delay -clock aclk 1.0 [get_ports m_axi_*ready]
set_input_delay -clock aclk 1.0 [get_ports m_axi_*rdata*]
set_input_delay -clock aclk 1.0 [get_ports m_axi_*rresp*]
set_input_delay -clock aclk 1.0 [get_ports m_axi_*bresp*]
set_output_delay -clock aclk 1.0 [get_ports m_axi_*valid]
set_output_delay -clock aclk 1.0 [get_ports m_axi_*addr*]
set_output_delay -clock aclk 1.0 [get_ports m_axi_*data*]

```

#### 19.4.2 False Path Constraints

```

# Reset is asynchronous
set_false_path -from [get_ports aresetn]

# Configuration registers to datapath (multi-cycle)
set_multicycle_path 2 -setup -from [get_cells *cfg_*]
set_multicycle_path 1 -hold -from [get_cells *cfg_*]

```

---

## 19.5 CDC Considerations

### 19.5.1 Single Clock Domain

STREAM operates in a single clock domain. All clock domain crossing must be handled externally:

Interface	CDC Responsibility
APB Slave	External if APB on different clock
AXI Masters	External if AXI interconnect on different clock
MonBus	External if consumer on different clock

### 19.5.2 CDC Recommendations

For systems requiring CDC:

1. Use async FIFO for MonBus output
  2. Use AXI clock converter IP for AXI interfaces
  3. Use APB bridge with clock crossing for APB
- 

**Last Updated:** 2026-01-03

## 20 Verification Strategy

### 20.1 Verification Approach

#### 20.1.1 Multi-Level Verification

STREAM verification follows a hierarchical approach:

Level	Focus	Coverage Target
<b>Block</b>	Individual module function	100% line/toggle
<b>Integration</b>	Block-to-block interfaces	95% functional
<b>System</b>	End-to-end operation	90% use cases

### 20.2 Block-Level Verification

#### 20.2.1 Test Categories

Block	Key Test Scenarios
<b>APB Config</b>	Register access, kick-off timing, error response
<b>Descriptor Engine</b>	Fetch, parse, chain, error handling
<b>Scheduler</b>	State transitions, multi-channel coordination
<b>Channel Arbiter</b>	Priority, fairness, starvation prevention
<b>AXI Read Engine</b>	Burst generation, error

Block	Key Test Scenarios
<b>AXI Write Engine</b>	handling, backpressure
<b>SRAM Buffer</b>	Burst generation, response handling
<b>MonBus Reporter</b>	Read/write concurrent, full/empty
	Event generation, backpressure, overflow

### 20.2.2 Coverage Metrics

Metric	Target
Line coverage	100%
Toggle coverage	95%
FSM state coverage	100%
FSM transition coverage	100%
Parameter coverage	All supported values

## 20.3 Integration-Level Verification

---

### 20.3.1 Interface Verification

Interface	Test Focus
APB to Descriptor Engine	Kick-off handshake, blocking behavior
Descriptor Engine to Scheduler	Descriptor passing, backpressure
Scheduler to AXI Engines	Grant/request, data flow
AXI Engines to SRAM	Concurrent access, pointer management
All blocks to MonBus	Event generation, arbitration

### 20.3.2 Multi-Channel Scenarios

Scenario	Description
Sequential	Channels operate one at a time

Scenario	Description
Concurrent	All 8 channels active simultaneously
Priority	High priority preempts low priority
Fairness	Equal priority channels share fairly
Error isolation	Error in one channel, others continue

## 20.4 System-Level Verification

---

### 20.4.1 End-to-End Test Cases

Test Case	Description
<b>Basic Transfer</b>	Single descriptor, single channel
<b>Chained Transfer</b>	Multiple descriptors, single channel
<b>Multi-Channel</b>	Multiple channels, independent transfers
<b>Scatter</b>	Single source to multiple destinations
<b>Gather</b>	Multiple sources to single destination
<b>Maximum Throughput</b>	Sustained bandwidth test
<b>Error Injection</b>	AXI errors, invalid descriptors
<b>Stress</b>	Random traffic, long duration

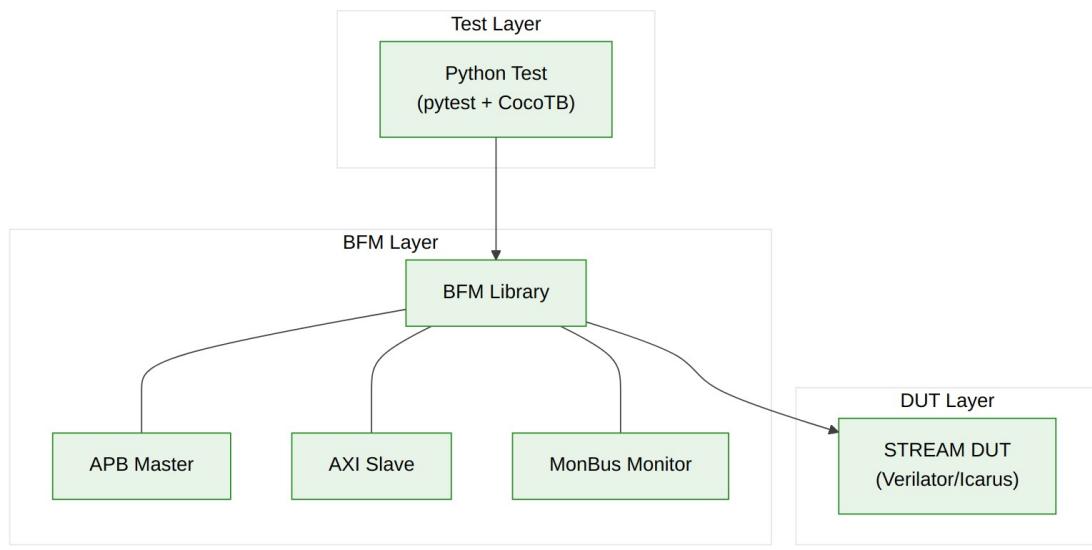
### 20.4.2 Performance Verification

Metric	Verification Method
Throughput	Measure bytes/cycle over large transfer
Latency	Measure kick-off to completion cycles
Efficiency	Compare actual vs theoretical

Metric	Verification Method
	maximum

## 20.5 Testbench Architecture

### 20.5.1 CocoTB-Based Verification



*Testbench Architecture*

Source: [07\\_testbench\\_stack.mmd](#)

### 20.5.2 Key Testbench Components

Component	Purpose
<b>APB Master BFM</b>	Drive configuration interface
<b>AXI Slave BFM</b>	Model memory system
<b>MonBus Monitor</b>	Capture and verify events
<b>Scoreboard</b>	Verify data integrity
<b>Coverage Collector</b>	Track functional coverage

## 20.6 Verification Environment

---

### 20.6.1 Simulator Support

Simulator	Status
Verilator	Primary (open-source)
Icarus Verilog	Secondary
Commercial (VCS, Questa)	Supported

### 20.6.2 Test Execution

# Run all STREAM tests

```
pytest projects/components/stream/dv/tests/ -v
```

# Run specific test category

```
pytest projects/components/stream/dv/tests/fub_tests/ -v
```

```
pytest projects/components/stream/dv/tests/integration_tests/ -v
```

# Generate coverage report

```
pytest projects/components/stream/dv/tests/ --cov-report=html
```

---

## 20.7 Formal Verification

---

### 20.7.1 Properties for Formal

Category	Properties
Protocol	AXI handshake rules, APB timing
Live ness	No deadlock in arbitration
Safety	No data corruption, proper error handling
Functional	Descriptor chaining correctness

### 20.7.2 Formal Tool Usage

Recommended for:

- AXI protocol compliance
- Arbiter deadlock freedom
- FSM reachability analysis

---

## **20.8 Sign-Off Criteria**

---

### **20.8.1 Block-Level Sign-Off**

- 100% line coverage
- 95% toggle coverage
- All FSM states exercised
- All FSM transitions exercised
- No simulation errors
- No lint warnings

### **20.8.2 Integration Sign-Off**

- All interface scenarios passed
- Multi-channel concurrency verified
- Error handling verified
- Performance targets met

### **20.8.3 System Sign-Off**

- All end-to-end test cases passed
  - Throughput targets achieved
  - Latency targets achieved
  - 72-hour stress test passed
- 

**Last Updated:** 2026-01-03

## **21 STREAM Specification Index**

**Version:** 0.90 **Date:** 2025-11-22 **Purpose:** Complete technical specification for STREAM subsystem

---

## **21.1 Document Organization**

---

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

### **21.1.1 Front Matter**

- [Document Information](#)

### **21.1.2 Chapter 1: Overview**

- [Architecture](#)
- [Top-Level Port List](#)
- [Clocks and Reset](#)

### **21.1.3 Chapter 2: Functional Blocks**

**Macro/Integration Level:** - Stream Core - Scheduler Group Array - Scheduler Group

**Control Path:** - Scheduler - Descriptor Engine

**Read Datapath:** - AXI Read Engine - Stream Alloc Control

**SRAM Buffering:** - SRAM Controller - SRAM Controller Unit - Stream Latency Bridge

**Write Datapath:** - Stream Drain Control - AXI Write Engine

**Configuration & Monitoring:** - APB to Descriptor Router - APB Config - Performance Profiler - MonBus AXIL Group

### **21.1.4 Chapter 3: Interfaces**

- [Interface Overview](#)
- [AXI4 Interface Specification](#)
- [AXIL4 Interface Specification](#)
- [APB Programming Interface](#)
- [MonBus Interface Specification](#)

### **21.1.5 Chapter 4: Registers**

- [Register Map](#)

### **21.1.6 Chapter 5: Programming**

- [Programming Guide](#)

### **21.1.7 Chapter 6: Configuration Reference**

- [Complete Configuration Guide](#)
-

## 21.2 Quick Reference

---

### 21.2.1 Functional Unit Blocks (FUB)

Module	File	Purpose	Lines	Status
descriptor_engine	stream_fub/ descriptor_engine.sv	Descriptor fetch/parse (256-bit)	~300	[Done]
scheduler	stream_fub/ scheduler.sv	Transfer coordinator	~400	[Done] Created (corrected)
axi_read_engine	stream_fub/ axi_read_engine.sv	AXI read master	~350	[Done] Created (no FSM - streaming pipeline)
axi_write_engine	stream_fub/ axi_write_engine.sv	AXI write master	~400	[Done] Created (no FSM - streaming pipeline)
sram_controller	stream_fub/ sram_controller.sv	Per-channel buffer management	~350	[Done] Created (no FSM - pointer arithmetic + pre-allocation)
simple_sram	stream_fub/ simple_sram.sv	Dual-port SRAM primitive	~150	[Done]
perf_profiler	stream_fub/ perf_profiler.sv	Channel performance profiling	~400	[Done] Dual-mode timestamp/elapsed tracking

### 21.2.2 Integration Blocks (MAC)

Module	File	Purpose	Lines	Status
channel_arbiter	stream_macro/	Priority arbitration	~200	[Pending] To be created

Module	File	Purpose	Lines	Status
	channel_a rbiter.sv			
apb_config	regs/ stream_re gs.rdl + wrapper	Config registers	~350	[Future] PeakRDL- based
monbus_axil_g roup	stream_ma cro/ monbus_ax il_group. sv	MonBus + AXIL	~800	[Done]
stream_top	stream_ma cro/ stream_to p.sv	Top-level	~500	[Pending] To be created

## 21.3 Performance Modes (AXI Engines)

---

STREAM AXI engines support three performance modes via compile-time parameters, offering area/performance trade-offs from tutorial simplicity to datacenter throughput.

### 21.3.1 Holistic Overview: V1/V2/V3 Working Together

**Design Philosophy:** - **V1 (Low):** Tutorial-focused, simple architecture, minimal area - **V2 (Medium):** Command pipelining hides memory latency, 6.7x throughput improvement - **V3 (High):** Out-of-order completion maximizes memory controller efficiency, 7.0x throughput

**Key Insight:** The benefit scales with memory latency. For low-latency SRAM (2-3 cycles), V1 achieves 40% throughput. For high-latency DDR4 (70-100 cycles), V1 drops to 14% but V2/V3 maintain 94-98% throughput.

**Parameterization Strategy:** - `ENABLE_CMD_PIPELINE = 0`: V1 (default, tutorial mode) - `ENABLE_CMD_PIPELINE = 1`: V2 (command pipelined, best area efficiency) - `ENABLE_CMD_PIPELINE = 1, ENABLE_000_DRAIN = 1` (write) or `ENABLE_000_READ = 1` (read): V3 (out-of-order, maximum throughput)

### 21.3.2 V1 - Low Performance (Single Outstanding Transaction)

**Architecture:** Single-burst-per-request, blocking on completion - **Area:** ~1,250 LUTs per engine - **Throughput:** 0.14 beats/cycle (DDR4), 0.40 beats/cycle (SRAM) - **Outstanding Txns:** 1 - **Use Case:** Tutorial examples, embedded systems, low-latency SRAM - **Key Feature:** Zero-bubble streaming pipeline (NO FSM!)

**Blocking Behavior:** - Write: Blocks on B response before accepting next request - Read: Blocks on R last before accepting next request - Simple control: 3 flags (r\_ar\_inflight, r\_ar\_valid, completion tracking)

### 21.3.3 V2 - Medium Performance (Command Pipelined)

**Architecture:** Command queue decouples command acceptance from data completion - **Area:** ~2,000 LUTs per engine (1.6x increase) - **Throughput:** 0.94 beats/cycle (DDR4), 0.85 beats/cycle (SRAM) - **Outstanding Txns:** 4-8 (command queue depth) - **Use Case:** General-purpose FPGA, DDR3/DDR4, balanced area/performance - **Key Feature:** Hides memory latency via pipelining (6.7x improvement)

**Command Pipelining:** - Write: AW queue + W drain FSM + B scoreboard (async response tracking) - Read: AR queue + R in-order reception (simpler than write, no B channel) - Per-command SRAM pointers enable multiple bursts from same channel

**Best Area Efficiency:** 4.19x throughput per unit area (6.7x throughput / 1.6x area)

### 21.3.4 V3 - High Performance (Out-of-Order Completion)

**Architecture:** OOO command selection maximizes memory controller efficiency - **Area:** ~3,500-4,000 LUTs per engine (2.8-3.2x increase) - **Throughput:** 0.98 beats/cycle (DDR4), 0.92 beats/cycle (SRAM) - **Outstanding Txns:** 8-16 (larger command queue) - **Use Case:** Datacenter, ASIC, HBM2, high-performance memory controllers - **Key Feature:** Flexible drain order based on SRAM data availability (7.0x improvement)

**Out-of-Order Mechanisms:** - Write: OOO W drain (select command with SRAM data ready), AXI ID matching for B responses - Read: OOO R reception (match m\_axi\_rid to queue entry), independent SRAM write per command - Transaction ID structure: {counter, channel\_id} preserves channel routing for MonBus

**Why OOO is Naturally Supported:** 1. Per-channel SRAM partitioning (no cross-channel hazards) 2. Per-command pointer tracking (no pointer collision) 3. Transaction ID matching (channel ID in lower bits for routing)

### 21.3.5 Performance Comparison Summary

Configuration	Area a	DDR4 Throughput	SRAM Throughput	Area Efficiency	Use Case
V1	1.0x	0.14 beats/cycle (1.0x)	0.40 beats/cycle (1.0x)	1.00	Tutorial, embedd ed
V2	1.6x	0.94 beats/cycle (6.7x)	0.85 beats/cycle (2.1x)	4.19	General FPGA

Configuration	Area a	DDR4 Throughput	SRAM Throughput	Area Efficiency	Use Case
V3	2.8x	0.98 beats/cycle (7.0x)	0.92 beats/cycle (2.3x)	2.50	Datacenter, ASIC

**Key Takeaway:** V2 offers best area efficiency (4.19x) for typical FPGA use cases. V3 provides marginal throughput improvement (7.0x vs 6.7x) at higher area cost, justified only for high-performance memory controllers that support out-of-order responses.

---

## 21.4 Clock and Reset Summary

### 21.4.1 Clock Domains

Clock	Frequency	Usage
aclk	100-500 MHz	Primary - all STREAM logic, AXI/AXIL interfaces
pclk	50-200 MHz	APB configuration interface (may be async to aclk)

### 21.4.2 Reset Signals

Reset	Polarity	Type	Usage
aresetn	Active-low	Async assert, sync deassert	Primary - all STREAM logic
presetn	Active-low	Async assert, sync deassert	APB configuration interface

See: [Clocks and Reset](#) for complete timing specifications

---

## 21.5 Interface Summary

---

### 21.5.1 External Interfaces

Interface	Type	Width	Purpose
APB	Slave	32-bit	Configuration registers
AXI (Descriptor)	Master	256-bit	Descriptor fetch
AXI (Read)	Master	512-bit (param)	Source data read
AXI (Write)	Master	512-bit (param)	Destination data write
AXIL (Slave)	Slave	32-bit	Error/interrupt FIFO access
AXIL (Master)	Master	32-bit	MonBus packet logging to memory
IRQ	Output	1-bit	Error interrupt

### 21.5.2 Internal Buses

Interface	Width	Purpose
MonBus	64-bit	Internal monitoring bus (channels -> monbus_axil_group)

## 21.6 Area Estimates

---

### 21.6.1 By Performance Mode

Configuration	Total LUTs	SRAM	Use Case
Low (Tutorial)	~9,500	64 KB	Educational, area-constrained
Medium (Typical)	~11,200	64 KB	Balanced FPGA implementation

Configuration	Total LUTs	SRAM	Use Case
High (Performance)	~13,700	64 KB	ns
			High-throughput ASIC/FPGA

### 21.6.2 Breakdown (Low Performance)

Component	Instances	Area/Instance	Total
Descriptor Engine	8	~300 LUTs	~2,400 LUTs
Scheduler	8	~400 LUTs	~3,200 LUTs
AXI Read Engine (Low)	1	~250 LUTs	~250 LUTs
AXI Write Engine (Low)	1	~250 LUTs	~250 LUTs
SRAM Controller	1	~1,600 LUTs	~1,600 LUTs
Simple SRAM (internal)	1-8	1024x64B total	64 KB
Channel Arbiter	3	~150 LUTs	~450 LUTs
APB Config	1	~350 LUTs	~350 LUTs
MonBus AXIL Group	1	~1,000 LUTs	~1,000 LUTs
<b>Total</b>	-	-	<b>~9,500 LUTs + 64KB</b>

## 21.7 Related Documentation

---

- [PRD.md](#) - Product requirements and overview
  - [ARCHITECTURAL\\_NOTES.md](#) - Critical design decisions
  - [CLAUDE.md](#) - AI development guide
  - [Register Generation](#) - PeakRDL workflow
-

## 21.8 Specification Conventions

---

### 21.8.1 Signal Naming

- **Clock:** aclk, pclk
- **Reset:** aresetn, presetn (active-low)
- **Valid/Ready:** Standard AXI/custom handshake
- **Registers:** r\_ prefix (e.g., r\_state, r\_counter)
- **Wires:** w\_ prefix (e.g., w\_next\_state, w\_grant)

### 21.8.2 Parameter Naming

- **Uppercase:** NUM\_CHANNELS, DATA\_WIDTH, ADDR\_WIDTH
- **String parameters:** PERFORMANCE (“LOW”, “MEDIUM”, “HIGH”)

### 21.8.3 State Machine Naming

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

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

---

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

## 22 Product Requirements Document (PRD)

### 22.1 STREAM - Scatter-gather Transfer Rapid Engine for AXI Memory

---

Version: 1.0 Date: 2025-10-17 Status: Nearly Complete - Final Integration Pending Owner: RTL Design Sherpa Project Parent Document: /PRD.md

---

### 22.2 1. Executive Summary

---

The **STREAM** (Scatter-gather Transfer Rapid Engine for AXI Memory) is a simplified DMA-style accelerator designed for efficient memory-to-memory data movement with descriptor-based

scatter-gather support. STREAM serves as a beginner-friendly tutorial demonstrating descriptor-based DMA engine design patterns while maintaining production-quality RTL practices.

### 22.2.1 1.1 Quick Stats

- **Modules:** ~8-10 SystemVerilog files (estimated)
- **Channels:** Maximum 8 independent channels
- **Interfaces:** APB (config), AXI4 (descriptor fetch + data read/write), MonBus (monitoring)
- **Architecture:** Simplified from RAPIDS - pure memory-to-memory
- **Tutorial Focus:** Aligned addresses only, straightforward data flow
- **Status:** Nearly complete - config interface and top-level wrapper pending

### 22.2.2 1.2 Project Goals

- **Primary:** Educational DMA engine demonstrating scatter-gather descriptor chains
  - **Secondary:** Production-quality RTL suitable for FPGA/ASIC implementation
  - **Tertiary:** Foundation for understanding more complex DMA architectures (e.g., RAPIDS)
- 

## 22.3 2. Key Design Principles

---

### 22.3.1 2.1 Simplifications from RAPIDS

STREAM is intentionally simplified for tutorial purposes:

Feature	RAPIDS	STREAM
<b>Network Interfaces</b>	Yes (Network master/slave)	✗ No (pure memory-to-memory)
<b>Address Alignment</b>	Complex fixup logic	✓ <b>Aligned addresses only</b>
<b>Credit Management</b>	Exponential encoding	✓ Simple transaction limits
<b>Control Engines</b>	Control read/write engines	✗ No (direct APB config)
<b>Descriptor Length</b>	Chunks (4-byte)	✓ <b>Beats</b> (data width)
<b>Program Engine</b>	Complex alignment	✓ Simplified

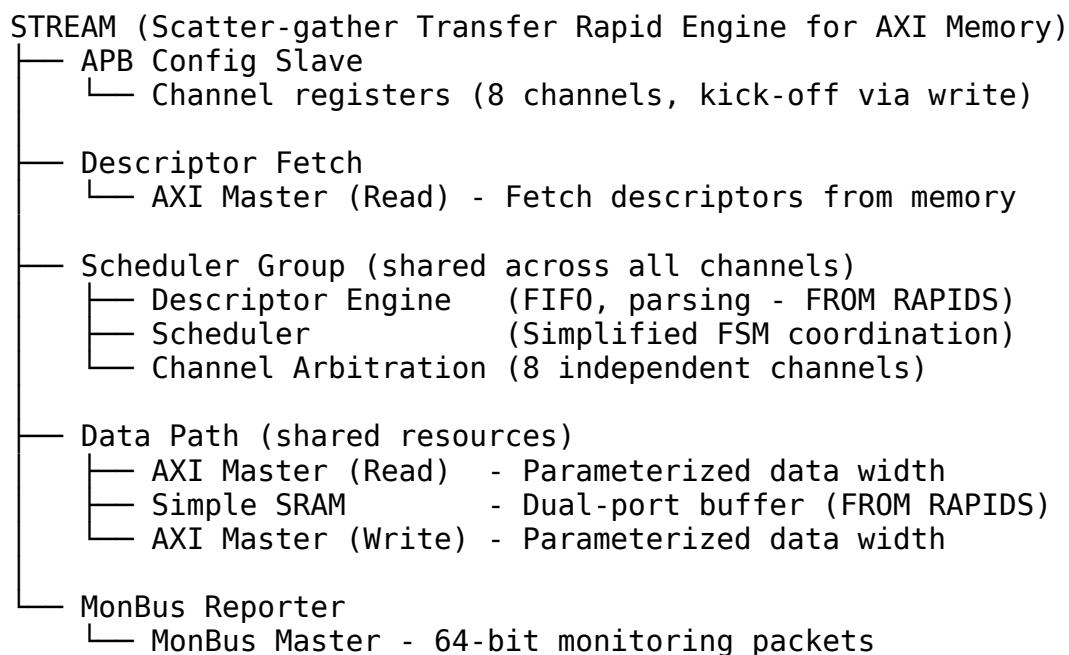
Feature	RAPIDS	STREAM
	FSM	coordination

### 22.3.22.2 Tutorial-Friendly Features

- **Aligned Addresses:** Source and destination addresses must be aligned to data width
- **Length in Beats:** Descriptor length specified in data beats (not bytes/chunks)
- **Single APB Write:** One APB register write kicks off entire descriptor chain
- **No Circular Buffers:** Chained descriptors with explicit termination
- **Parameterized Engines:** Multiple AXI engine versions (compile-time selection)

## 22.4 3. Architecture Overview

### 22.4.1 3.1 Top-Level Block Diagram



### 22.4.2 3.2 Data Flow

#### Descriptor-Based Transfer Sequence:

1. Software writes to APB channel register  
↓

2. Descriptor fetch via AXI descriptor master
  - ↓
3. Descriptor Engine parses descriptor fields
  - ↓
4. Scheduler coordinates data transfer:
  - a. AXI Read Engine fetches source data → SRAM buffer
  - b. AXI Write Engine writes SRAM → destination
  - ↓
5. Check for chained descriptor (`next_descriptor_ptr != 0`)
  - ↓ (if chained)
6. Fetch next descriptor, repeat from step 3
  - ↓ (if last)
7. Generate MonBus completion packet

**Channel Independence:** - 8 channels operate independently - All channels share: SRAM, AXI data masters, descriptor fetch master - Arbitration required for shared resources

---

## 22.5 4. Interfaces

### 22.5.1 4.1 External Interfaces

Interface	Type	Width	Purpose	Notes
<b>APB Slave</b>	Slave	32-bit	Configuration, channel kick-off	Write to channel register starts transfer
<b>AXI Master (Descriptor)</b>	Master	256-bit	Fetch descriptors from memory	Dedicated descriptor fetch path
<b>AXI Master (Data Read)</b>	Master	Parameterizable	Read source data	Multiple engine versions (compile-time)
<b>AXI Master (Data Write)</b>	Master	Parameterizable	Write destination data	Multiple engine versions (compile-time)

Interface	Type	Width	Purpose	Notes
<b>MonBus Master</b>	Master	64-bit	Monitor packet output	Standard AMBA format

### 22.5.24.2 Descriptor Format

256-bit Descriptor Structure:

Bits	Field	Description
[63:0]	src_addr	Source address (64-bit, must be aligned to data width)
[127:64]	dst_addr	Destination address (64-bit, must be aligned to data width)
[159:128]	length	Transfer length in <b>BEATS</b> (not bytes!)
[191:160]	next_descriptor_ptr	Address of next descriptor (0 = last in chain)
[192]	valid	Descriptor is valid
[193]	interrupt	Generate interrupt on completion
[194]	last	Last descriptor in chain (explicit flag)
[195]	error	Error status (used for reporting)
[199:196]	channel_id	Channel ID (0-7)
[207:200]	priority	Transfer priority (for arbitration)
[255:208]	reserved	Reserved for future use

**Key Descriptor Features:** - ✓ **Chained descriptors:** next\_descriptor\_ptr links to next descriptor - ✗ **No circular buffers:** Explicit termination (last flag or ptr=0) - ✓ **Length in beats:** Simplified for tutorial (no byte/chunk conversion) - ✓ **Aligned addresses:** Tutorial constraint (performance hidden for now)

---

## 22.6 5. Key Components

---

### 22.6.15.1 Descriptor Engine (APB-Only for STREAM)

**Source:** Adapted from RAPIDS descriptor\_engine.sv

**Purpose:** - Autonomous descriptor fetch and chaining - APB interface for initial descriptor address - AXI read interface for descriptor memory fetches - Descriptor FIFO storage and distribution

**Key Features:** - ✓ **Autonomous chaining:** Automatically fetches next descriptor if `next_descriptor_ptr != 0` AND `last == 0` - ✓ **Address validation:** Validates next descriptor addresses against `cfg_addr0/1_base/limit` - ✓ **APB blocking:** APB blocked until `channel_idle == 1` (channel fully idle) - ✓ **Error handling:** AXI errors stop chaining, set `descriptor_error`, block `descriptor_valid`

**Adaptations from RAPIDS:** - ✗ **RDA removed:** STREAM is memory-to-memory only (no network interfaces) - ✓ **APB-only:** Single APB write kicks off entire descriptor chain - ✓ **Descriptor Read Address FIFO:** 2-deep FIFO stores addresses for AXI fetch (APB + chaining) - ✓ **Chaining logic:** Descriptor engine autonomously manages `next_descriptor_ptr` chaining

**Idle Signal:** - `descriptor_engine_idle` asserted when: - FSM in RD\_IDLE state - No pending descriptor fetches (address FIFO empty) - No active AXI transactions

## 22.6.25.2 Scheduler Group (Integration Wrapper)

**Purpose:** Wraps descriptor engine and scheduler into a single channel processing unit

**Architecture:**

```
scheduler_group (
    // APB interface (from APB config slave)
    .apb_valid          (apb_valid),
    .apb_ready          (apb_ready),    // Blocked when channel not idle
    .apb_addr           (descriptor_addr),

    // Channel idle signal composition (CRITICAL!)
    .channel_idle       (channel_idle),

    // Descriptor → Scheduler flow
    .descriptor_valid   (desc_valid),
    .descriptor_ready   (desc_ready),
    .descriptor_packet  (desc_packet),

    // Data engine interfaces
    .datard_*           (datard_*),      // Read engine
    .datawr_*           (datawr_*),      // Write engine

    // Status
    .scheduler_idle     (sched_idle),
    .descriptor_idle    (desc_idle)
);
```

**Channel Idle Signal Composition:**

```
// Channel is idle ONLY when BOTH sub-blocks are idle
assign channel_idle = scheduler_idle && descriptor_engine_idle;
```

#### Why Both Signals Matter:

Signal	Indicates	Used For
scheduler_idle	No active data transfers, all descriptors processed	Prevents new APB request during active transfer
descriptor_engine_idle	No pending descriptor fetches (FIFO empty)	Prevents new APB request during chaining
channel_idle (AND of both)	Channel fully quiescent	<b>Gates APB interface</b>

#### APB Blocking Logic:

```
// Descriptor engine blocks APB when channel not idle
assign apb_ready = apb_skid_ready_in &&
                  !r_channel_reset_active &&
                  w_desc_addr_fifo_empty &&      // No pending fetches
                  channel_idle;                   // Scheduler +
descriptor_idle
```

#### Example Scenario:

1. Software writes APB → descriptor\_addr = 0x1000
  - channel\_idle = 1 (both idle)
  - APB accepted
2. Descriptor engine fetches descriptor @ 0x1000
  - descriptor\_engine\_idle = 0 (fetch in progress)
  - channel\_idle = 0
  - APB BLOCKED
3. Descriptor pushed to scheduler
  - descriptor\_engine\_idle = 1 (fetch complete)
  - scheduler\_idle = 0 (transfer starting)
  - channel\_idle = 0
  - APB BLOCKED
4. Scheduler completes data transfer
  - Descriptor has next\_descriptor\_ptr = 0x1100 (chained!)
  - Descriptor engine autonomously fetches @ 0x1100
  - descriptor\_engine\_idle = 0 (autonomous fetch)
  - channel\_idle = 0
  - APB BLOCKED

5. Final descriptor completes (last = 1 OR next\_ptr = 0)
  - scheduler\_idle = 1 (transfer done)
  - descriptor\_engine\_idle = 1 (no more fetches)
  - channel\_idle = 1
  - APB UNBLOCKED (ready for next transfer!)

**Key Insight:** The AND gate ensures software cannot interrupt a descriptor chain in progress!

### 22.6.35.3 Scheduler (Simplified from RAPIDS)

**Purpose:** - Coordinate descriptor-to-data-transfer flow - Manage 8 independent channels - Arbitrate shared resources (SRAM, AXI masters)

**FSM States:**

```
typedef enum logic [7:0] {
    SCHED_IDLE          = 8'b00000001, // Idle, waiting for
    channel activation
    SCHED_FETCH_DESCRIPTOR = 8'b00000010, // Fetch descriptor via
    AXI master
    SCHED_PARSE_DESCRIPTOR = 8'b00000100, // Parse descriptor fields
    SCHED_READ_PHASE     = 8'b00001000, // Coordinate read engine
    SCHED_WRITE_PHASE    = 8'b00010000, // Coordinate write engine
    SCHED_CHAIN_CHECK    = 8'b00100000, // Check for next
    descriptor
    SCHED_COMPLETE       = 8'b01000000, // Transfer complete,
    report status
    SCHED_ERROR          = 8'b10000000 // Error state
} scheduler_state_t;
```

**Key Differences from RAPIDS:** - ✗ No credit management (just simple transaction limits) - ✗ No program engine coordination (no alignment fixup) - ✓ Simplified FSM (no control read/write phases)

### 22.6.45.3 AXI Read Engine (Streaming Pipeline - NO FSM)

**Purpose:** High-performance streaming reads from memory to SRAM buffer

**Architecture:** Pipelined streaming design (NO FSM for performance)

**Key Insight:** FSMs are horrible for performance! Instead, use: - **Arbiter** selects which channel's datard\_\* interface gets access - **Streaming pipeline** continuously moves data when granted - **Data interface** (datard\_valid, datard\_ready, datard\_beats\_remaining) controls flow

**Data Read Interface (per channel):**

```
// Channel requests read access
 logic datard_valid;           // Channel has read request
 output logic datard_ready; // Engine ready for request
 input logic [63:0] datard_addr; // Source address (aligned)
```

```



```

**Multiple Versions (Compile-Time Selection):** 1. **Version 1 - Basic:** Single outstanding read, fixed burst length 2. **Version 2 - Pipelined:** Multiple outstanding reads, configurable bursts 3. **Version 3 - Adaptive:** Dynamic burst sizing based on remaining beats

#### Operation:

1. Arbiter grants channel access based on priority
2. Engine accepts datard\_\* request (continuous streaming)
3. AXI AR channel issues read burst
4. AXI R channel streams data → SRAM (no FSM stalls!)
5. Engine updates beats\_remaining, accepts next request
6. Different channels can have different burst lengths (e.g., CH0: 8 beats, CH1: 16 beats)

#### Example: Different Burst Lengths per Channel

```

// Channel 0 prefers 8-beat bursts
datard_burst_len[0] = 8'd8;

// Channel 1 prefers 16-beat bursts
datard_burst_len[1] = 8'd16;

// Engine adapts to requested burst length (within MAX_BURST_LEN)

```

### 22.6.5.4 AXI Write Engine (Streaming Pipeline - NO FSM)

**Purpose:** High-performance streaming writes from SRAM buffer to memory

**Architecture:** Pipelined streaming design (NO FSM for performance)

**Key Insight:** Same as read engine - no FSMs! Use streaming pipeline with arbiter.

#### Data Write Interface (per channel):

```

// Channel requests write access


```

**Multiple Versions (Compile-Time Selection):** 1. **Version 1 - Basic:** Single outstanding write, fixed burst length 2. **Version 2 - Pipelined:** Multiple outstanding writes, configurable bursts 3. **Version 3 - Adaptive:** Dynamic burst sizing based on remaining beats

#### Operation:

1. Arbiter grants channel access based on priority
2. Engine accepts datawr\_\* request (continuous streaming)
3. Engine reads data from SRAM
4. AXI AW channel issues write address
5. AXI W channel streams data (no FSM stalls!)
6. AXI B channel receives response, updates beats\_remaining
7. Different channels can have different burst lengths (e.g., CH0: 8 beats, CH2: 32 beats)

#### Read/Write Asymmetry Example:

```
// Channel can use different burst lengths for read vs write
// Example: Read in small bursts, write in large bursts
datard_burst_len[0] = 8'd8; // Read: 8 beats
datawr_burst_len[0] = 8'd16; // Write: 16 beats

// Engine handles asymmetry via SRAM buffering
```

#### 22.6.6.5 Simple SRAM

**Source:** Direct copy from RAPIDS simple\_sram.sv

**Purpose:** - Dual-port SRAM buffer - Decouples read and write engines - Shared across all channels (arbitration required)

**Why Reuse:** - Standard dual-port SRAM design - Proven in RAPIDS integration tests - Parameterizable depth and width

---

## 22.7 6. Configuration and Control

---

### 22.7.16.1 APB Register Map

Offset	Register	Access	Description
0x0000	GLOBAL_CTRL	RW	Global enable, reset
0x0004	GLOBAL_STATUS	RO	Global status, error flags
0x0100	CH0_CTRL	WO	<b>Channel 0 kick-off</b> (write descriptor address)

Offset	Register	Access	Description
0x0104	CH0_STATUS	RO	Channel 0 status
0x0108	CH0_DESC_ADDR	RO	Channel 0 current descriptor address
0x010C	CH0_BYTES_XFER	RO	Channel 0 bytes transferred
...	...	...	... (repeat for channels 1-7)
0x0200	CH1_CTRL	WO	Channel 1 kick-off
...			
0x0700	CH7_CTRL	WO	Channel 7 kick-off

**Kick-Off Sequence:** 1. Software writes descriptor address to CHx\_CTRL register 2. STREAM fetches descriptor from memory 3. Transfer begins automatically 4. If chained, STREAM follows next\_descriptor\_ptr automatically 5. Completion reported via MonBus packet

### 22.7.26.2 Channel Configuration

Each channel independently configurable: - **Descriptor start address:** Written to CHx\_CTRL - **Priority:** Encoded in descriptor (arbitration) - **Interrupt enable:** Per-descriptor flag - **Status monitoring:** Read CHx\_STATUS

---

## 22.8 7. Resource Sharing and Arbitration

---

### 22.8.17.1 Shared Resources

**All channels share:** 1. **Descriptor Fetch AXI Master** - Fetches descriptors for all channels 2. **Data Read AXI Master** - Reads source data for all channels 3. **Data Write AXI Master** - Writes destination data for all channels 4. **SRAM Buffer** - Shared buffer (dual-port, but still arbitrated) 5. **MonBus Reporter** - Single monitor output

### 22.8.27.2 Arbitration Strategy

**Priority-Based Round-Robin:** - Channels have priority field in descriptor - Higher priority = serviced first - Within same priority: round-robin - Prevents starvation with timeout

#### Example Arbitration:

Channel 0: Priority 7 (highest)  
Channel 1: Priority 5

Channel 2: Priority 5  
Channel 3: Priority 3

Service order: CH0 → CH1 → CH2 (round-robin) → CH0 → CH1 → CH2 → CH3 ...

---

## 22.9 8. Error Detection and Recovery

---

### 22.9.1 8.1 Error Types

Error Type	Detection	Response
<b>Invalid descriptor</b>	Valid bit = 0	Skip, move to next
<b>Alignment error</b>	Address not aligned	Set error flag, halt channel
<b>AXI SLVERR</b>	AXI response	Set error flag, halt channel
<b>AXI DECERR</b>	AXI response	Set error flag, halt channel
<b>Timeout</b>	Transaction timeout	Set error flag, halt channel
<b>SRAM overflow</b>	Buffer full	Backpressure, wait

### 22.9.2 8.2 Error Recovery

**Per-Channel Error Handling:** - Error sets channel to CH\_ERROR state - Channel halts, does not affect other channels - Software must: 1. Read CHx\_STATUS to identify error 2. Clear error condition 3. Re-kick channel with new descriptor

**No Automatic Retry:** - Tutorial design keeps error handling simple - Software responsible for retry logic

---

## 22.10 9. MonBus Integration

---

### 22.10.1 9.1 Standard MonBus Format

Uses standard 64-bit MonBus packet format: - [63:60] Packet Type (0=ERROR, 1=COMPL, etc.) - [59:57] Protocol (custom STREAM protocol) - [56:53] Event Code (STREAM-specific events) - [52:47] Channel ID (0-7) - [46:43] Unit ID (unused for STREAM) - [42:35] Agent ID (unused for STREAM) - [34:0] Event Data (address, byte count, etc.)

## 22.10.2 9.2 STREAM Event Codes

Code	Event	Description
0x0	DESC_START	Descriptor started
0x1	DESC_COMPLETE	Descriptor completed
0x2	READ_START	Read phase started
0x3	READ_COMPLETE	Read phase completed
0x4	WRITE_START	Write phase started
0x5	WRITE_COMPLETE	Write phase completed
0x6	CHAIN_FETCH	Chained descriptor fetch
0xF	ERROR	Error occurred

## 22.10.3 9.3 Default Configuration

**Tutorial-Friendly Defaults:** - Errors only: `cfg_error_enable = 1`, all others = 0 - **Interrupts:** Descriptor flag controls per-transfer interrupt - **Reduces MonBus traffic** for beginner understanding

---

## 22.11 10. Design Constraints

### 22.11.1 10.1 Tutorial Constraints (Intentional Simplifications)

Constraint	Rationale
<b>Aligned addresses only</b>	Simplify data path, hide alignment complexity
<b>Length in beats</b>	Avoid byte/chunk conversion math
<b>No circular buffers</b>	Explicit termination easier to understand
<b>Single APB kick-off</b>	Simple software model
<b>No credit management</b>	Avoid exponential encoding complexity

## 22.11.2 10.2 Implementation Constraints

Parameter	Value	Notes
Max channels	8	Compile-time parameter
Max burst length	256	AXI4 spec limit
Descriptor size	256 bits	4 × 64-bit words
SRAM depth	Parameterizable	Typical: 1024-4096 entries
Data width	Parameterizable	Typical: 512-bit

## 22.12 11. Verification Strategy

---

### 22.12.1 11.1 Test Organization

```
projects/components/stream/dv/tests/
└── fub_tests/                               # Functional Unit Block tests
    ├── descriptor_engine/                  # Copy from RAPIDS (adapt imports)
    ├── scheduler/                         # Simplified scheduler tests
    ├── axi_engines/                      # Read/write engine tests
    └── sram/                             # SRAM tests

└── integration_tests/                     # Multi-block scenarios
    ├── single_channel/                  # Single channel transfers
    ├── multi_channel/                  # 8-channel concurrent
    ├── chained_descriptors/            # Descriptor chain tests
    └── error_handling/                # Error recovery tests
```

### 22.12.2 11.2 Test Levels

**Basic (Quick Smoke Tests):** - Single descriptor, single channel - Aligned addresses, simple transfers - ~30 seconds runtime

**Medium (Typical Scenarios):** - Multiple descriptors, 2-4 channels - Chained descriptors (2-3 deep) - ~90 seconds runtime

**Full (Comprehensive Validation):** - All 8 channels concurrent - Long descriptor chains (10+ deep) - Error injection, stress testing - ~180 seconds runtime

---

## 22.13 12. Performance Characteristics

---

### 22.13.1 12.1 Throughput by Engine Version

**V1 (Low Performance - Tutorial Mode):** - **Throughput:** 0.14 beats/cycle (DDR4), 0.40 beats/cycle (SRAM) - **Architecture:** Single outstanding transaction, blocks on completion - **Use Case:** Tutorial examples, embedded systems, low-latency SRAM

**V2 (Medium Performance - Command Pipelined):** - **Throughput:** 0.94 beats/cycle (DDR4), 0.85 beats/cycle (SRAM) - **Improvement:** 6.7x over V1 (DDR4), 2.1x over V1 (SRAM) - **Architecture:** Command queue (4-8 deep), hides memory latency - **Use Case:** General-purpose FPGA, DDR3/DDR4, best area efficiency

**V3 (High Performance - Out-of-Order):** - **Throughput:** 0.98 beats/cycle (DDR4), 0.92 beats/cycle (SRAM) - **Improvement:** 7.0x over V1 (DDR4), 2.3x over V1 (SRAM) - **Architecture:** OOO command selection, maximizes memory controller efficiency - **Use Case:** Datacenter, ASIC, HBM2, high-performance memory

**Key Insight:** Benefit scales with memory latency. V1 throughput degrades from 40% (SRAM) to 14% (DDR4), while V2/V3 maintain 94-98% throughput regardless of latency.

**Configuration Parameters:** - `ENABLE_CMD_PIPELINE = 0`: V1 (default, tutorial mode) - `ENABLE_CMD_PIPELINE = 1`: V2 (command pipelined) - `ENABLE_CMD_PIPELINE = 1`, `ENABLE_000_DRAIN = 1` (write) or `ENABLE_000_READ = 1` (read): V3

**Factors Affecting Throughput:** - Memory latency (V2/V3 hide latency via pipelining) - SRAM buffer size (limits burst pipelining) - Channel arbitration overhead - Descriptor fetch latency - Engine version (V1/V2/V3 configuration)

### 22.13.2 12.2 Latency

**Transfer Latency Breakdown:** - Descriptor fetch: ~10-50 cycles (depends on memory) - Read phase:  $(\text{length} / \text{burst\_len}) \times \text{burst\_latency}$  - Write phase:  $(\text{length} / \text{burst\_len}) \times \text{burst\_latency}$  - End-to-end: Typically <200 cycles for small transfers (V1), <100 cycles (V2/V3 pipelined)

**V2/V3 Latency Hiding:** - Command pipelining overlaps descriptor fetch, read, write operations - Multiple outstanding transactions hide memory latency - OOO completion (V3) reduces head-of-line blocking

### 22.13.3 12.3 Resource Utilization by Engine Version

**V1 Configuration (Tutorial - Minimum Area):** - Total: ~9,500 LUTs + 64 KB SRAM - Descriptor Engine (8×): ~2,400 LUTs - Scheduler (8×): ~3,200 LUTs - AXI Read Engine: ~1,250 LUTs - AXI Write Engine: ~1,250 LUTs - SRAM Controller: ~1,600 LUTs - APB Config: ~350 LUTs - MonBus AXIL Group: ~1,000 LUTs

**V2 Configuration (Balanced - Best Area Efficiency):** - Total: ~11,000 LUTs + 64 KB SRAM (1.16x area) - AXI Read Engine: ~2,000 LUTs (1.6x increase, 6.7x throughput) - AXI Write Engine: ~2,500 LUTs (2.0x increase, 6.7x throughput) - Other blocks: Same as V1

**V3 Configuration (Maximum Performance):** - Total: ~14,000 LUTs + 64 KB SRAM (1.47x area) - AXI Read Engine: ~3,500 LUTs (2.8x increase, 7.0x throughput) - AXI Write Engine: ~4,000 LUTs (3.2x increase, 7.0x throughput) - Other blocks: Same as V1

**Area Efficiency Comparison:** - V1: 1.00 throughput / 1.00 area = 1.00 - V2: 6.70 throughput / 1.16 area = 5.78 (best efficiency) - V3: 7.00 throughput / 1.47 area = 4.76

**Recommendation:** V2 provides best area efficiency for most use cases. V3 justified only for high-performance memory controllers that support OOO responses.

---

## 22.14 13. Development Roadmap

---

### 22.14.1 13.1 Phase 1: Foundation (Current)

- ✓ Directory structure
- ✓ Package definitions (`stream_pkg.sv`)
- ✓ Imports header (`stream_imports.svh`)
- ⏳ Documentation (this PRD)

### 22.14.2 13.2 Phase 2: Core Blocks

- Adapt descriptor engine from RAPIDS
- Design simplified scheduler
- Create APB config interface
- Copy simple SRAM from RAPIDS

### 22.14.3 13.3 Phase 3: Data Path

- AXI read engine (version 1 - basic)
- AXI write engine (version 1 - basic)
- SRAM integration
- Channel arbitration

### 22.14.4 13.4 Phase 4: Integration

- Top-level module
- MonBus reporter
- Integration testbench

- Single-channel validation

#### **22.14.5 13.5 Phase 5: Multi-Channel**

- 8-channel support
- Arbiter implementation
- Multi-channel tests
- Performance tuning

#### **22.14.6 13.6 Phase 6: Advanced Engines (Future - V2/V3)**

**Goal:** Add parameterized high-performance engine variants

**V2 - Command Pipelined (Medium Performance):** - Command queue implementation (4-8 deep) - W drain FSM for write engine - B response scoreboard (write) or in-order R reception (read) - Per-command SRAM pointer tracking - Parameter: ENABLE\_CMD\_PIPELINE = 1 - Expected: 6.7x throughput improvement over V1

**V3 - Out-of-Order Completion (High Performance):** - OOO command selection logic - Transaction ID matching (AXI ID to queue entry) - SRAM data availability checking (write engine) - R beat matching to queue entry (read engine) - Parameters: ENABLE\_CMD\_PIPELINE = 1, ENABLE\_000\_DRAIN = 1 (write) or ENABLE\_000\_READ = 1 (read) - Expected: 7.0x throughput improvement over V1

**Deliverables:** - Updated RTL with parameterization - Performance comparison tests (V1 vs V2 vs V3) - Tutorial documentation explaining trade-offs - Area/throughput measurements on target FPGA

---

### **22.15 14. Educational Value**

#### **22.15.1 14.1 Learning Objectives**

**STREAM teaches:** 1. **Descriptor-based DMA design patterns** - Descriptor structure and parsing - Chained descriptors (scatter-gather) - Descriptor fetch via AXI

##### **2. AXI4 Memory Interface Usage**

- Burst transactions
- Address/data/response channels
- Outstanding transactions

##### **3. Resource Sharing and Arbitration**

- Multiple channels sharing AXI masters
- Priority-based arbitration

- Conflict resolution
- 4. FSM Design and Coordination**
- Multiple interconnected FSMs
  - State machine composition
  - Error handling
- 5. Buffer Management**
- SRAM-based buffering
  - Rate matching
  - Flow control

## 22.15.2 14.2 Progression Path

**Learning Sequence:** 1. **STREAM (this project):** Memory-to-memory DMA, aligned addresses 2. **STREAM Extended:** Add alignment fixup, more complex scenarios 3. **RAPIDS:** Add network interfaces, credit management, full complexity

---

## 22.16 15. Success Criteria

### 22.16.1 15.1 Functional

- ✓ Single descriptor transfer working
- ✓ Chained descriptors (2-3 deep)
- ✓ Multi-channel operation (8 channels concurrent)
- ✓ Error detection and reporting
- ✓ MonBus packet generation
- ✓ >90% functional test coverage

### 22.16.2 15.2 Quality

- ✓ Verilator compiles with 0 warnings
- ✓ All tests passing (100% success rate)
- ✓ Comprehensive documentation
- ✓ Tutorial-quality code comments

### 22.16.3 15.3 Performance

- ✓ Achieves >80% of theoretical AXI bandwidth
- ✓ <5K LUT utilization (excluding SRAM)
- ✓ <200 cycle end-to-end latency for small transfers

---

## **22.17 16. Open Questions (For Review)**

---

### **22.17.1 16.1 Descriptor Engine Adaptation**

- **Q:** Should descriptor engine use APB-only, RDA-only, or mixed mode?
- **A (pending):** TBD - depends on software use case preference

### **22.17.2 16.2 AXI Descriptor Master**

- **Q:** Fixed 256-bit width, or parameterizable?
- **A (pending):** Propose fixed 256-bit for simplicity

### **22.17.3 16.3 Channel Arbitration**

- **Q:** Fixed priority, or dynamic priority based on age/fairness?
- **A (pending):** Propose fixed priority with round-robin for tutorial simplicity

### **22.17.4 16.4 SRAM Partitioning**

- **Q:** Single shared SRAM, or per-channel SRAMs?
  - **A (pending):** Propose single shared SRAM with arbitration (matches RAPIDS pattern)
- 

## **22.18 16. Attribution and Contribution Guidelines**

---

### **22.18.1 16.1 Git Commit Attribution**

When creating git commits for STREAM documentation or implementation:

**Use:**

Documentation and implementation support by Claude.

**Do NOT use:**

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

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

---

## 22.19 16.2 PDF Generation Location

---

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

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

**Quick Command:** Use the provided shell script:

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

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

 See: bin/md\_to\_docx.py for complete implementation details

---

## 22.20 17. References

---

### 22.20.1 17.1 Internal Documentation

- **RAPIDS PRD:** projects/components/rapids/PRD.md - Parent architecture
- **RAPIDS Descriptor Engine:**  
projects/components/rapids/rtl/rapids\_fub/descriptor\_engine.sv
- **RAPIDS Simple SRAM:**  
projects/components/rapids/rtl/rapids\_fub/simple\_sram.sv
- **AMBA PRD:** rtl/amba/PRD.md - MonBus integration
- **Repository Guide:** /CLAUDE.md - Design patterns and conventions

### 22.20.2 17.2 External References

- **AXI4 Specification:** ARM IHI0022E
  - **APB Specification:** ARM IHI0024C
  - CocoTB Documentation: <https://docs.cocotb.org/>
  - Verilator Manual: <https://verilator.org/guide/latest/>
- 

**Document Version:** 1.0 **Last Updated:** 2025-10-17 **Review Cycle:** Weekly during initial design

**Next Review:** TBD (after user feedback) **Owner:** RTL Design Sherpa Project

---

## 22.21 Navigation

---

- ← **Back to Root:** /PRD.md
- **Parent Architecture:** projects/components/rapids/PRD.md
- **AI Guidance:** CLAUDE.md (to be created)
- **Quick Start:** README.md (to be created)

## 23 Claude Code Guide: STREAM Subsystem

**Version:** 1.0 **Last Updated:** 2025-10-17 **Purpose:** AI-specific guidance for working with STREAM subsystem

---

### 23.1 Quick Context

---

**What:** STREAM - Scatter-gather Transfer Rapid Engine for AXI Memory **Status:** 🟡 Initial design - tutorial-focused DMA engine **Your Role:** Help users build a beginner-friendly descriptor-based DMA engine

📖 **Complete Specification:** projects/components/stream/PRD.md ← Always reference this for technical details

---

### 23.2 📖 Global Requirements Reference

---

**IMPORTANT:** Review /GLOBAL\_REQUIREMENTS.md for all mandatory requirements

All mandatory requirements are consolidated in the global requirements document: - **See:** /GLOBAL\_REQUIREMENTS.md - Repository-wide mandatory requirements - **STREAM-Specific:** Attribution format, tutorial focus (intentional simplifications) - **Universal:** TB location, three methods, TBBBase inheritance, 100% success

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

---

### 23.3 Critical Rules for This Subsystem

---

#### 23.3.1 Rule #0: Attribution Format for Git Commits

**IMPORTANT:** When creating git commit messages for STREAM documentation or code:

#### Use:

Documentation and implementation support by Claude.

#### Do NOT use:

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

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

### 23.3.2 Rule #0.1: TUTORIAL FOCUS - Intentional Simplifications

⚠ STREAM is INTENTIONALLY SIMPLIFIED for educational purposes ⚠

**Key Simplifications (DO NOT “fix” these):** 1. ✓ **Aligned addresses only** - No alignment fixup logic (kept for RAPIDS) 2. ✓ **Length in beats** - Not bytes or chunks (simplifies math) 3. ✓ **No circular buffers** - Explicit chain termination only 4. ✓ **No credit management** - Simple transaction limits 5. ✓ **Pure memory-to-memory** - No network interfaces

**When users ask “Can we add alignment fixup?”:** - ✓ **Correct answer:** “STREAM intentionally keeps addresses aligned for tutorial simplicity. For complex alignment, see RAPIDS.” - ✗ **Wrong answer:** “Sure, let me add alignment logic...” (defeats tutorial purpose!)

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

📖 See: /GLOBAL\_REQUIREMENTS.md Section 2.1 for complete requirement

#### STREAM-Specific Directory Structure:

```
projects/components/stream/dv/
  └── tbclasses/                                # ★ STREAM TB classes (project
    └── area!
      └── scheduler_tb.py                      # Scheduler testbench
      └── descriptor_engine_tb.py            # Descriptor engine testbench
      └── axi_engine_tb.py                  # AXI engine testbenches
    └── tests/fub_tests/                     # Test runners import from
      └── tbclasses/
```

#### STREAM Import Pattern:

```
# Import STREAM TB from project area
from projects.components.stream.dv.tbclasses.scheduler_tb import
StreamSchedulerTB
```

```
# Shared utilities from framework
from CocoTBFramework.tbclasses.shared.tbbase import TBBBase
```

📖 **Complete Pattern:** projects/components/rapids/CLAUDE.md Rule #0.1

### 23.3.4 Rule #0.2: Three Mandatory TB Methods (MANDATORY)

📖 See: /GLOBAL\_REQUIREMENTS.md Section 2.2 for complete requirement

#### STREAM-Specific Context:

STREAM has simpler config requirements than RAPIDS - most config can be set after reset.

#### Standard STREAM Pattern:

```
class StreamSchedulerTB(TBBase):
    async def setup_clocks_and_reset(self):
        """Standard STREAM initialization"""
        await self.start_clock('clk', freq=10, units='ns')
        await self.assert_reset()
        await self.wait_clocks('clk', 10)
        await self.deassert_reset()
        await self.wait_clocks('clk', 5)

    async def assert_reset(self):
        self.dut.rst_n.value = 0

    async def deassert_reset(self):
        self.dut.rst_n.value = 1
```

**Note:** Unlike RAPIDS, STREAM typically doesn't need config set before reset.

### 23.3.5 Rule #1: REUSE from RAPIDS Where Appropriate

**Direct Reuse (No Modification):** - ✓ descriptor\_engine.sv - Works with STREAM descriptors - ✓ simple\_sram.sv - Standard dual-port SRAM - ✓ Descriptor engine tests - Adapt imports only

**Adapt from RAPIDS:** - ⚠ scheduler.sv - Simplify FSM (no credit management, no control engines) - ⚠ AXI engines - Create simplified versions (no alignment fixup)

**Create New for STREAM:** - 🖌 APB config interface - PeakRDL-generated (like HPET), 8 channels, kick-off registers - 🖌 Top-level integration - Different interface set

**Always Ask Yourself:** “Can I reuse from RAPIDS instead of creating new?”

### 23.3.6 Rule #2: Descriptor Format is DIFFERENT from RAPIDS

**STREAM Descriptor (256-bit):** - src\_addr (64-bit), dst\_addr (64-bit), length (**beats**), next\_descriptor\_ptr (32-bit) - **Length is in BEATS, not chunks!**

**RAPIDS Descriptor:** - Uses chunks (4-byte units) - Has alignment metadata

**When comparing/referencing RAPIDS:** - ✓ “RAPIDS uses chunks, STREAM uses beats for tutorial simplicity” - ✗ Don’t assume RAPIDS descriptor format applies to STREAM

### 23.3.7 Rule #3: Know the Shared Resources

**All 8 channels share:** 1. Descriptor fetch AXI master 2. Data read AXI master 3. Data write AXI master 4. SRAM buffer 5. MonBus reporter

**Arbitration is required!** - Never assume exclusive access - All shared resources need arbiter logic - Priority-based round-robin (descriptor priority field)

---

## 23.4 Architecture Quick Reference

---

### 23.4.1 Block Organization

STREAM Architecture (Estimated: 8-10 modules)

APB Config	apb_config_slave.sv	(To be created)
Scheduler Group	descriptor_engine.sv scheduler.sv channel_arbiter.sv	(FROM RAPIDS - adapt imports) (Simplified from RAPIDS) (To be created)
Data Path	axi_read_engine_v1.sv axi_read_engine_v2.sv axi_write_engine_v1.sv axi_write_engine_v2.sv simple_sram.sv	(Version 1: Basic) (Version 2: Pipelined) (Version 1: Basic) (Version 2: Pipelined) (FROM RAPIDS - no changes)
MonBus Reporter	monbus_reporter.sv	(To be created)

### 23.4.2 Module Status

Module	Source	Status	Notes
descriptor_engine.sv	RAPIDS (copy)	✓ Copied	Adapt #include only
simple_sram.sv	RAPIDS (copy)	✓ Copied	No changes needed
stream_pkg.sv	New	✓ Created	Descriptor format defined
stream_imports.svh	New	✓ Created	Package imports
scheduler.sv	RAPIDS (simplify)	⌚ Pending	Remove credit

Module	Source	Status	Notes
			mgmt, control engines
apb_config_slave.sv	New	Pending	8 channel registers
axi_read_engine_v1.sv	New	Pending	Basic version
axi_write_engine_v1.sv	New	Pending	Basic version
channel_arbiter.sv	New	Pending	Priority-based round-robin
monbus_reporter.sv	New	Pending	STREAM event codes
stream_top.sv	New	Pending	Top-level integration

## 23.5 Common User Questions and Responses

---

### 23.5.1 Q: "How is STREAM different from RAPIDS?"

A: STREAM is intentionally simplified for tutorial purposes:

**Simplifications:** | Feature | RAPIDS | STREAM | |——|——|——| | **Interfaces** | APB + AXI + Network | APB + AXI only | | **Address Alignment** | Complex fixup | Aligned only | | **Credit Management** | Exponential encoding | Simple limits | | **Descriptor Length** | Chunks (4-byte) | Beats (data width) | | **Control Engines** | Control read/write | None (direct APB) |

**Learning Path:** 1. **STREAM** - Basic DMA with scatter-gather 2. **STREAM Extended** - Add alignment fixup 3. **RAPIDS** - Full complexity with network + credit management

📖 See: PRD.md Section 2.1 for complete comparison table

### 23.5.2 Q: "How do I kick off a transfer?"

A: Single APB write starts descriptor chain:

```
// Software writes descriptor address to channel register
write_apb(ADDR_CH0_CTRL, 0x1000_0000); // Start address of descriptor

// STREAM automatically:
// 1. Fetches descriptor from 0x1000_0000
// 2. Parses src_addr, dst_addr, length
```

```

// 3. Reads source data → SRAM
// 4. Writes SRAM → destination
// 5. Checks next_descriptor_ptr
//     - If != 0: Fetch next descriptor, repeat
//     - If == 0 or last flag set: Complete

```

#### **Chained Descriptors:**

Descriptor 0 @ 0x1000\_0000:  
src\_addr = 0x2000\_0000  
dst\_addr = 0x3000\_0000  
length = 64 beats  
next\_descriptor\_ptr = 0x1000\_0100 ← Points to next descriptor

Descriptor 1 @ 0x1000\_0100:  
src\_addr = 0x2000\_1000  
dst\_addr = 0x3000\_1000  
length = 32 beats  
next\_descriptor\_ptr = 0x0000\_0000 ← Last descriptor (0 = stop)

See: PRD.md Section 3.2 for complete data flow

#### **23.5.3 Q: "How many channels can I use?"**

A: Maximum 8 independent channels:

**Channel Operation:** - Each channel has own FSM state - Each channel can have separate descriptor chain - All channels share: AXI masters, SRAM, MonBus

**Arbitration:** - Priority-based (descriptor priority field) - Round-robin within same priority - Prevents starvation with timeout

**Example:**

```

// Kick off 3 channels concurrently
write_apb(ADDR_CH0_CTRL, desc0_addr); // Channel 0: Priority 7
write_apb(ADDR_CH1_CTRL, desc1_addr); // Channel 1: Priority 5
write_apb(ADDR_CH2_CTRL, desc2_addr); // Channel 2: Priority 5

// Service order: CH0 → CH1 → CH2 → CH0 → CH1 → CH2 ...

```

See: PRD.md Section 7 for arbitration details

#### **23.5.4 Q: "What's the descriptor format?"**

A: 256-bit descriptor (4 × 64-bit words):

```

typedef struct packed {
    logic [63:0] reserved;           // [255:192] Reserved
    logic [7:0]  priority;          // [207:200] Priority

```

```

logic [3:0]    channel_id;           // [199:196] Channel ID
logic          error;              // [195] Error flag
logic          last;               // [194] Last in chain
logic          interrupt;          // [193] Generate interrupt
logic          valid;              // [192] Valid descriptor
logic [31:0]   next_descriptor_ptr; // [191:160] Next descriptor
address
    logic [31:0]   length;             // [159:128] Length in BEATS
★
    logic [63:0]   dst_addr;           // [127:64] Destination
address
    logic [63:0]   src_addr;           // [63:0] Source address
} descriptor_t;

```

★ **CRITICAL:** length is in **BEATS**, not bytes or chunks!

**Example:**

Data width = 512 bits = 64 bytes  
Length = 16 beats  
Total transfer =  $16 \times 64 = 1024$  bytes

📖 See: PRD.md Section 4.2 for complete descriptor specification

### 23.5.5 Q: “How do I run STREAM tests?”

**A: Multi-layered test approach (same as RAPIDS):**

```

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

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

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

```

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

---

## 23.6 Integration Patterns

---

### 23.6.1 Pattern 1: Basic STREAM Instantiation

```
stream_top #(
    .NUM_CHANNELS(8),
    .DATA_WIDTH(512),
    .ADDR_WIDTH(64),
    .SRAM_DEPTH(4096)
) u_stream (
    // Clock and Reset
    .aclk            (system_clk),
    .aresetn        (system_rst_n),

    // APB Configuration Interface
    .s_apb_paddr     (apb_paddr),
    .s_apb_psel      (apb_psel),
    .s_apb_penable   (apb_penable),
    .s_apb_pwrite    (apb_pwrite),
    .s_apb_pwdata   (apb_pwdata),
    .s_apb_pready   (apb_pready),
    .s_apb_prdata   (apb_prdata),
    .s_apb_pslverr  (apb_pslverr),

    // AXI Master - Descriptor Fetch (256-bit)
    .m_axi_desc_araddr (desc_araddr),
    .m_axi_desc_arlen  (desc_arlen),
    .m_axi_desc_arsize (desc_arsize),
    .m_axi_desc_arvalid (desc_arvalid),
    .m_axi_desc_arready (desc_arready),
    .m_axi_desc_rdata  (desc_rdata),
    .m_axi_desc_rresp  (desc_rresp),
    .m_axi_desc_rlast  (desc_rlast),
    .m_axi_desc_rvalid (desc_rvalid),
    .m_axi_desc_rready (desc_rready),

    // AXI Master - Data Read (parameterizable width)
    .m_axi_rd_araddr  (rd_araddr),
    // ... (full AXI4 AR + R channels)

    // AXI Master - Data Write (parameterizable width)
    .m_axi_wr_awaddr  (wr_awaddr),
    // ... (full AXI4 AW + W + B channels)

    // MonBus Output
    .monbus_pkt_valid (stream_mon_valid),
    .monbus_pkt_ready (stream_mon_ready),
```

```

    .monbus_pkt_data      (stream_mon_data)
};

23.6.2 Pattern 2: Descriptor Creation (Software Model)

// C/C++ software model for descriptor creation
typedef struct {
    uint64_t src_addr;           // Source address (must be aligned!)
    uint64_t dst_addr;          // Destination address (must be
aligned!)
    uint32_t length;            // Transfer length in BEATS
    uint32_t next_descriptor_ptr; // Next descriptor address (0 =
last)
    uint8_t control;           // valid | interrupt | last | error |
channel_id | priority
} stream_descriptor_t;

// Create descriptor chain
stream_descriptor_t desc[2];

// Descriptor 0
desc[0].src_addr = 0x80000000ULL; // Aligned to 64B (512-bit data)
desc[0].dst_addr = 0x90000000ULL;
desc[0].length = 64; // 64 beats × 64 bytes = 4KB transfer
desc[0].next_descriptor_ptr = (uint32_t)&desc[1]; // Chain to next
desc[0].control = 0x01; // valid = 1

// Descriptor 1 (last)
desc[1].src_addr = 0x80001000ULL;
desc[1].dst_addr = 0x90001000ULL;
desc[1].length = 32; // 32 beats × 64 bytes = 2KB transfer
desc[1].next_descriptor_ptr = 0; // Last descriptor
desc[1].control = 0x45; // valid | last | interrupt

// Kick off transfer
write_apb_reg(CH0_CTRL, (uint32_t)&desc[0]);

```

### 23.6.3 Pattern 3: MonBus Integration

```

// Always add downstream FIFO for MonBus
gaxi_fifo_sync #(
    .DATA_WIDTH(64),
    .DEPTH(256)
) u_stream_mon_fifo (
    .i_clk      (aclk),
    .i_rst_n    (aresetn),
    .i_data     (monbus_pkt_data),
    .i_valid    (monbus_pkt_valid),
    .o_ready    (monbus_pkt_ready),

```

```
.o_data      (fifo_mon_data),
.o_valid     (fifo_mon_valid),
.i_ready     (consumer_ready)
);
```

---

## 23.7 Anti-Patterns to Catch

### 23.7.1 X Anti-Pattern 1: Adding Alignment Fixup

✗ WRONG:

```
"Let me add alignment fixup logic to handle unaligned addresses..."
```

✓ CORRECTED:

```
"STREAM intentionally requires aligned addresses for tutorial  
simplicity.
```

If you need unaligned transfers, that's a great learning exercise **for**  
extending STREAM, **or use** RAPIDS which has full alignment support."

### 23.7.2 X Anti-Pattern 2: Using Length in Bytes

✗ WRONG:

```
descriptor.length = 4096; // Thinking this is 4096 bytes
```

✓ CORRECTED:

```
// Length is in BEATS, not bytes!
// For 512-bit data width (64 bytes per beat):
descriptor.length = 4096 / 64; // = 64 beats for 4096 bytes
```

### 23.7.3 X Anti-Pattern 3: Circular Buffer Descriptors

✗ WRONG:

```
// Descriptor chain that loops back
desc[9].next_descriptor_ptr = &desc[0]; // Circular!
```

✓ CORRECTED:

```
"STREAM does not support circular buffers (no stop condition).  
Always terminate chains explicitly:  
desc[last].next_descriptor_ptr = 0; // Stop  
desc[last].last = 1; // Explicit last flag
```

### 23.7.4 X Anti-Pattern 4: Assuming Exclusive Channel Access

✗ WRONG:

```
// Assume channel has exclusive AXI master access
assign m_axi_arvalid = channel_request; // No arbitration!
```

✓ CORRECTED:

```
// All channels share AXI masters - arbitration required
channel_arbiter u_arbiter (
    .ch_requests (channel_requests[7:0]),
    .ch_grant    (channel_grant[7:0]),
    .axi_master_available (axi_master_idle)
);
```

---

## 23.8 Debugging Workflow

### 23.8.1 Issue: Descriptor Not Fetched

**Check in order:** 1. ✓ APB write to CHx\_CTRL register successful? 2. ✓ Descriptor address valid? 3. ✓ AXI descriptor master not stalled? 4. ✓ Descriptor memory accessible? 5. ✓ Arbiter granting descriptor fetch?

**Debug commands:**

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

### 23.8.2 Issue: Data Transfer Stalls

**Check in order:** 1. ✓ SRAM buffer depth sufficient? 2. ✓ Source/destination addresses aligned? 3. ✓ AXI read/write engines getting grants? 4. ✓ Downstream AXI ready signals asserted? 5. ✓ Channel not in error state?

**Waveform Analysis:** - Check SRAM read/write pointers - Verify AXI AR/AW/R/W/B handshakes - Inspect scheduler FSM state - Check arbiter grant signals

### 23.8.3 Issue: Chained Descriptors Not Following

**Check in order:** 1. ✓ next\_descriptor\_ptr != 0? 2. ✓ last flag not set prematurely? 3. ✓ Descriptor fetch completing successfully? 4. ✓ Scheduler transitioning to CHAIN\_CHECK state? 5. ✓ MonBus showing CHAIN\_FETCH event?

---

## 23.9 Testing Guidance

### 23.9.1 Test Organization

```
projects/components/stream/dv/tests/
└── fub_tests/                      # Individual block tests
```

```

    └── descriptor_engine/      # Adapt from RAPIDS tests
    └── scheduler/             # Simplified scheduler tests
    └── axi_engines/           # Read/write engine tests
    └── sram/                  # SRAM tests (from RAPIDS)

└── integration_tests/      # Multi-block scenarios
    ├── single_channel/       # Single channel transfers
    ├── multi_channel/        # 8-channel concurrent
    ├── chained_descriptors/  # Descriptor chain tests
    └── error_handling/       # Error recovery tests

```

### 23.9.2 Test Levels

**Basic (Quick Smoke Tests):** - Single descriptor, single channel - Aligned addresses, simple transfers - ~30 seconds runtime

**Medium (Typical Scenarios):** - Multiple descriptors, 2-4 channels - Chained descriptors (2-3 deep) - ~90 seconds runtime

**Full (Comprehensive Validation):** - All 8 channels concurrent - Long descriptor chains (10+ deep) - Error injection, stress testing - ~180 seconds runtime

---

## 23.10 Key Documentation Links

---

### 23.10.1 Always Reference These

**This Subsystem:** - projects/components/stream/PRD.md - **Complete specification** - projects/components/stream/README.md - Quick start guide (to be created) - projects/components/stream/known\_issues/ - Bug tracking

**Related:** - projects/components/rapids/PRD.md - Parent architecture (for comparison) - rtl/amba/PRD.md - MonBus integration - /PRD.md - Master requirements - /CLAUDE.md - Repository guide

---

## 23.11 Quick Commands

---

```
# View complete specification
cat projects/components/stream/PRD.md
```

```
# Check package definition
cat projects/components/stream/rtl/includes/stream_pkg.sv
```

```
# Run tests (once created)
pytest projects/components/stream/dv/tests/fub_tests/ -v
```

```

pytest projects/components/stream/dv/tests/integration_tests/ -v

# Lint (once RTL created)
iverilog --lint-only
projects/components/stream/rtl/stream_fub/scheduler.sv

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

```

---

## 23.12 Remember

1. **Tutorial focus** - Intentional simplifications for learning
  2. **Reuse from RAPIDS** - Descriptor engine, SRAM, patterns
  3. **Length in beats** - Not bytes or chunks!
  4. **Aligned addresses** - No fixup logic
  5. **Chained descriptors** - No circular buffers
  6. **8 channels** - Shared resources, arbitration required
  7. **MonBus standard** - Same format as AMBA/RAPIDS
  8. **Testbench reuse** - Always create TB classes in  
bin/CocoTBFramework/tbclasses/stream/
- 

## 23.13 PDF Generation Location

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

```
/mnt/data/github/rtl_design_sherpa/projects/components/stream/docs/
```

**Quick Command:** Use the provided shell script:

```
cd /mnt/data/github/rtl_design_sherpa/projects/components/stream/docs
./generate_pdf.sh
```

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

**See:** bin/md\_to\_docx.py for complete implementation details

---

**Version:** 1.0 **Last Updated:** 2025-10-17 **Maintained By:** RTL Design Sherpa Project