

RTL Design Sherpa

STREAM Subsystem Micro-Architecture Specification 0.90

January 2, 2026

Table of Contents

1 Stream Index.....	61
2 Document Information.....	61
2.1 References.....	61
2.1.1 Related Documents.....	61
2.2 Terminology.....	61
2.3 Revision History.....	63
3 STREAM Architecture Overview.....	63
3.1 Introduction.....	64
3.1.1 Design Philosophy.....	64
3.2 System Architecture.....	65
3.2.1 High-Level Block Diagram.....	65
3.2.2 Figure 1.1.1: STREAM Architecture Overview.....	65
3.3 Key Architectural Concepts.....	65
3.3.1 1. Multi-Channel Design.....	65
3.3.2 2. Descriptor-Based Operation.....	66
3.3.3 3. Concurrent Read/Write.....	66
3.3.4 4. Space-Aware Flow Control.....	66
3.3.5 5. Priority-Based Arbitration.....	66
3.4 Data Flow Example.....	67
3.4.1 Single Channel Transfer.....	67
3.4.2 Figure 1.1.2: Single Channel Transfer Sequence.....	67
3.5 Component Hierarchy.....	67

3.5.1 Top-Level Integration (stream_top_ch8.sv).....	67
3.5.2 Core DMA Engine (stream_core.sv).....	68
3.6 Interface Summary.....	68
3.6.1 External Interfaces (stream_top_ch8.sv).....	68
3.6.2 Internal MonBus Interface (stream_core.sv).....	69
3.6.3 Configuration Registers (APB).....	69
3.7 Resource Utilization.....	69
3.7.1 Area Breakdown (Typical 512-bit Data Width).....	69
3.8 Performance Characteristics.....	70
3.8.1 Throughput.....	70
3.8.2 Latency.....	70
3.9 Design Decisions.....	70
3.9.1 Why These Simplifications?.....	70
3.9.2 Production Enhancements (Future).....	71
3.10 Testing Strategy.....	71
3.10.1 Verification Layers.....	71
3.10.2 Coverage Targets.....	71
3.11 Related Documentation.....	72
3.12 Revision History.....	72
4 STREAM Top-Level Port List.....	72
4.1 Overview.....	72
4.2 Clock and Reset.....	73
4.3 APB Programming Interface.....	73
4.4 Configuration Interface.....	74

4.4.1 Per-Channel Configuration.....	74
4.4.2 Global Scheduler Configuration.....	74
4.4.3 Descriptor Engine Configuration.....	75
4.4.4 AXI Monitor Configuration.....	75
4.4.5 AXI Transfer Configuration.....	77
4.4.6 Performance Profiler Configuration.....	77
4.5 Status Interface.....	78
4.5.1 Per-Channel Status.....	78
4.6 Performance Profiler Interface.....	79
4.7 AXI4 Master - Descriptor Fetch (256-bit).....	80
4.7.1 AR Channel (Read Address).....	80
4.7.2 R Channel (Read Data).....	81
4.8 AXI4 Master - Data Read (Parameterizable Width).....	81
4.8.1 AR Channel (Read Address).....	81
4.8.2 R Channel (Read Data).....	82
4.9 AXI4 Master - Data Write (Parameterizable Width).....	83
4.9.1 AW Channel (Write Address).....	83
4.9.2 W Channel (Write Data).....	84
4.9.3 B Channel (Write Response).....	84
4.10 Status/Debug Outputs.....	85
4.10.1 Descriptor AXI Monitor Status.....	85
4.10.2 Read Engine AXI Monitor Status.....	85
4.10.3 Write Engine AXI Monitor Status.....	86
4.11 Unified Monitor Bus Interface.....	86

4.12 Port Count Summary.....	87
4.13 Default Parameter Values.....	88
4.14 Related Documentation.....	89
4.15 Revision History.....	89
 5 Clocks and Reset Specification.....	89
5.1 Overview.....	89
5.2 Clock Domain.....	90
5.2.1 Primary Clock: aclk.....	90
5.2.2 Secondary Clock: pclk (APB Clock).....	90
5.3 Reset.....	90
5.3.1 Primary Reset: aresetn.....	90
5.3.2 Secondary Reset: presetn.....	91
5.4 Reset Sequencing.....	91
5.4.1 Power-On Reset.....	91
5.4.2 Functional Reset.....	91
5.4.3 Reset Recovery.....	92
5.5 Clock Requirements by Module.....	92
5.5.1 Functional Unit Blocks (FUB).....	92
5.5.2 Integration Blocks (MAC).....	92
5.6 Timing Constraints.....	93
5.6.1 Setup and Hold Times.....	93
5.6.2 Critical Paths.....	93
5.7 Clock Domain Crossing (CDC).....	94
5.7.1 APB Configuration CDC.....	94

5.7.2 No CDC Required.....	94
5.8 Reset State Initialization.....	95
5.8.1 Register Reset Values.....	95
5.8.2 SRAM Reset.....	95
5.9 Clock Gating (Optional).....	96
5.9.1 Per-Channel Clock Gating.....	96
5.10 Verification Requirements.....	96
5.10.1 Clock Checks.....	96
5.10.2 Reset Checks.....	96
5.10.3 CDC Checks.....	96
5.11 Example Reset Testbench.....	97
5.12 Related Documentation.....	98
Top-Level: mac_04_stream_top.md - Clock/reset integration.....	98
5.13 Revision History.....	98
6 STREAM Core Specification.....	98
6.1 Overview.....	98
6.1.1 Key Features.....	98
6.1.2 Block Diagram.....	99
6.1.3 Figure 2.1.1: STREAM Core Block Diagram.....	100
6.2 Architecture.....	102
6.2.1 Component Hierarchy.....	102
6.2.2 Data Flow.....	102
6.3 Parameters.....	103
6.3.1 Primary Configuration.....	103

6.3.2 Monitor Control.....	104
6.3.3 Outstanding Transaction Limits.....	104
6.3.4 AXI Skid Buffer Depths.....	104
6.3.5 MonBus Agent IDs.....	105
6.4 Port List.....	105
6.4.1 Clock and Reset.....	105
6.4.2 APB Programming Interface.....	105
6.4.3 Configuration Interface.....	106
6.4.4 Status Interface.....	109
6.4.5 AXI4 Master - Descriptor Fetch (256-bit).....	110
6.4.6 AXI4 Master - Data Read (Parameterizable Width).....	111
6.4.7 AXI4 Master - Data Write (Parameterizable Width).....	113
6.4.8 Status/Debug Outputs.....	114
6.4.9 Unified Monitor Bus Interface.....	116
6.5 Operation.....	116
6.5.1 Transfer Initialization.....	116
6.5.2 Monitoring Transfer Progress.....	117
6.6 Timing Diagrams.....	117
6.6.1 Complete Data Flow Timing.....	117
6.7 Testing.....	120
6.8 Resource Utilization.....	120
6.9 Integration Example.....	121
6.10 Related Documentation.....	122

MonBus AXI-Lite Group: 16_monbus_axil_group.md - Monitor bus arbitration	123
.....	123
6.11 Revision History.....	123
7 Scheduler Group Array.....	123
7.1 Overview.....	123
7.1.1 Key Features.....	123
7.1.2 Simplified from RAPIDS.....	124
7.2 Architecture.....	124
7.2.1 Component Hierarchy.....	124
7.2.2 Descriptor AXI Arbitration.....	124
7.2.3 Data Path Interfaces.....	125
7.3 Parameters.....	125
7.3.1 Monitor Bus Agent IDs.....	125
7.4 Port List.....	126
7.4.1 Clock and Reset.....	126
7.4.2 APB Programming Interface (Per-Channel).....	126
7.4.3 Configuration Interface (Per-Channel).....	126
7.4.4 Global Scheduler Configuration.....	126
7.4.5 Descriptor Engine Configuration.....	127
7.4.6 Status Interface (Per-Channel).....	128
7.4.7 Shared Descriptor AXI4 Master Read Interface.....	128
7.4.8 Shared Data Read Interface (Per-Channel Arrays).....	129
7.4.9 Shared Data Write Interface (Per-Channel Arrays).....	130
7.4.10 Unified Monitor Bus Interface.....	130

7.5 Operation.....	131
7.5.1 Descriptor Fetch Arbitration.....	131
7.5.2 Data Path Flow.....	131
7.5.3 MonBus Aggregation.....	131
7.6 Integration Example.....	131
7.7 Related Documentation.....	132
MonBus Arbiter: 16_monbus_axil_group.md - Monitor bus aggregation.....	133
7.8 Revision History.....	133
8 Scheduler Group.....	133
8.1 Overview.....	133
8.1.1 Key Features.....	133
8.1.2 Simplified from RAPIDS.....	134
8.2 Architecture.....	134
8.2.1 Component Hierarchy.....	134
8.2.2 Data Flow.....	134
8.3 Parameters.....	135
8.3.1 Monitor Bus Agent IDs.....	135
8.4 Port List.....	136
8.4.1 Clock and Reset.....	136
8.4.2 APB Programming Interface.....	136
8.4.3 Configuration Interface.....	136
8.4.4 Scheduler Configuration.....	136
8.4.5 Descriptor Engine Configuration.....	137
8.4.6 Status Interface.....	137

8.4.7 Descriptor AXI Interface (to external arbiter).....	138
8.4.8 Data Read Interface (to shared read engine).....	139
8.4.9 Data Write Interface (to shared write engine).....	139
8.4.10 Monitor Bus Interface.....	140
8.5 Operation.....	140
8.5.1 Transfer Sequence.....	140
8.5.2 MonBus Sources.....	140
8.6 Integration Example.....	141
8.7 Related Documentation.....	142
AXI Write Engine: 12_axi_write_engine.md - Shared write engine.....	142
8.8 Revision History.....	142
9 Scheduler Specification.....	142
9.1 Overview.....	143
9.1.1 Key Features.....	143
9.1.2 Block Diagram.....	144
9.1.3 Figure 2.4.1: Scheduler Block Diagram.....	144
9.2 CRITICAL: Concurrent Read/Write Design.....	144
9.3 Parameters.....	145
9.4 Port List.....	145
9.4.1 Clock and Reset.....	145
9.4.2 Configuration Interface.....	146
9.4.3 Status Interface.....	146
9.4.4 Descriptor Engine Interface.....	146
9.4.5 Data Read Interface.....	147

9.4.6 Data Write Interface.....	147
9.4.7 Error Signals.....	148
9.4.8 Monitor Bus Interface.....	148
9.5 Interface.....	149
9.5.1 Clock and Reset.....	149
9.5.2 Configuration Interface.....	149
9.5.3 Status Interface.....	149
9.5.4 Descriptor Engine Interface.....	149
9.5.5 Data Read Interface (to AXI Read Engine).....	149
9.5.6 Data Write Interface (to AXI Write Engine).....	150
9.5.7 Monitor Bus Interface.....	151
9.6 Descriptor Format.....	151
9.6.1 STREAM Descriptor (256-bit).....	151
9.7 FSM Operation.....	152
9.7.1 State Machine.....	152
9.7.2 Figure 2.4.2: Scheduler FSM.....	153
9.7.3 State Transitions.....	153
9.8 Beat Tracking.....	154
9.8.1 Independent Counters.....	154
9.8.2 Completion Detection.....	154
9.8.3 Multiple Requests per Descriptor.....	155
9.9 Address Management.....	155
9.9.1 Static Base Address.....	155
9.10 Timeout Detection.....	156

9.10.1 Timeout Counter.....	156
9.11 Error Handling.....	156
9.11.1 Error Sources.....	156
9.11.2 Sticky Error Flags.....	157
9.11.3 Error Recovery.....	157
9.12 Interrupt Generation.....	158
9.12.1 IRQ via MonBus.....	158
9.13 Descriptor Chaining.....	158
9.13.1 Chain Detection.....	158
9.13.2 Chain Termination.....	158
9.14 MonBus Integration.....	159
9.14.1 Event Types.....	159
9.14.2 MonBus Packet Format.....	159
9.15 Timing Diagrams.....	160
9.15.1 Normal Transfer (No Chaining).....	160
9.15.2 Figure 2.4.3: Scheduler Normal Transfer Timing.....	160
9.15.3 Descriptor Chaining.....	161
9.15.4 Figure 2.4.4: Scheduler Descriptor Chaining Timing.....	161
9.15.5 Single-Descriptor Transfer (Detailed).....	161
9.15.6 Chained Descriptors (Detailed).....	162
9.16 Testing.....	162
9.17 Performance Considerations.....	163
9.17.1 Concurrent Operation Benefit.....	163
9.17.2 Example Performance.....	163

9.18 Related Documentation.....	163
9.19 Revision History.....	164
10 Descriptor Engine.....	165
10.1 Overview.....	165
10.1.1 Key Features.....	165
10.2 Architecture.....	165
10.2.1 Operating Flow.....	165
10.2.2 FSM States.....	166
10.2.3 Descriptor Format (256-bit STREAM).....	166
10.3 Parameters.....	167
10.3.1 Monitor Bus Parameters.....	167
10.4 Port List.....	168
10.4.1 Clock and Reset.....	168
10.4.2 APB Programming Interface.....	168
10.4.3 Scheduler Interface.....	168
10.4.4 Enhanced Control Outputs.....	169
10.4.5 AXI4 AR Channel.....	169
10.4.6 AXI4 R Channel (FIXED 256-bit).....	170
10.4.7 Configuration Interface.....	170
10.4.8 Status Interface.....	171
10.4.9 Monitor Bus Interface.....	171
10.5 Operation.....	171
10.5.1 APB Acceptance Policy.....	171
10.5.2 Autonomous Chaining Decision Tree.....	171

10.5.3 AXI Transaction.....	172
10.6 Timing Diagrams.....	172
10.6.1 APB Kick-off Sequence.....	172
10.7 Integration Example.....	174
10.8 Common Issues.....	175
10.8.1 Issue 1: APB Write Not Accepted.....	175
10.8.2 Issue 2: Descriptor Chain Stops Early.....	175
10.9 Related Documentation.....	175
APB Interface: 13_apbtodescr.md - APB to descriptor kick-off.....	175
10.10 Revision History.....	175
11 AXI Read Engine.....	176
11.1 Overview.....	176
11.1.1 Key Features.....	176
11.2 Architecture.....	176
11.2.1 Operation Flow.....	176
11.2.2 Key Design Decisions.....	177
11.2.3 System Idle Behavior.....	177
11.3 Parameters.....	177
11.3.1 Derived Parameters.....	178
11.4 Port List.....	178
11.4.1 Clock and Reset.....	178
11.4.2 Configuration Interface.....	178
11.4.3 Scheduler Interface (Per-Channel).....	178
11.4.4 Completion Interface (Per-Channel).....	179

11.4.5 SRAM Allocation Interface.....	179
11.4.6 SRAM Write Interface.....	179
11.4.7 AXI4 AR Channel.....	180
11.4.8 AXI4 R Channel.....	180
11.4.9 Error Interface.....	180
11.4.10 Debug Interface.....	181
11.5 Operation.....	181
11.5.1 Space-Aware Request Masking.....	181
11.5.2 Transfer Size Calculation.....	181
11.5.3 Outstanding Transaction Tracking.....	181
11.5.4 Completion Strobe.....	182
11.6 Timing Diagrams.....	182
11.6.1 Perfect Streaming - AXI Read Transaction.....	182
11.6.2 Multi-Channel Streaming.....	183
11.7 Integration Example.....	184
11.8 Related Documentation.....	186
Write Engine: 12_axi_write_engine.md - Complementary write datapath.....	186
11.9 Revision History.....	186
12 Stream Allocation Controller.....	186
12.1 Overview.....	187
12.1.1 What Makes This a “Virtual FIFO”.....	187
12.2 The Allocation Problem.....	187
12.2.1 Without Allocation Controller.....	187
12.2.2 With Allocation Controller.....	187

12.3 Architecture.....	188
12.3.1 Two-Pointer System.....	188
12.3.2 Figure 2.7.1: Stream Allocation Controller Block Diagram.....	188
12.3.3 CRITICAL: Confusing Naming Convention.....	190
12.4 Parameters.....	190
12.5 Port List.....	191
12.5.1 Clock and Reset.....	191
12.5.2 Write Interface (Allocation Requests).....	191
12.5.3 Read Interface (Actual Data Written).....	192
12.5.4 Status Outputs.....	192
12.6 Interfaces.....	192
12.6.1 Write Interface (Allocation Requests).....	192
12.6.2 Read Interface (Actual Data Written).....	193
12.6.3 Status Outputs.....	193
12.7 Operation.....	194
12.7.1 Allocation Flow.....	194
12.7.2 Pointer Arithmetic.....	194
12.8 Timing Behavior.....	195
12.8.1 Allocation Latency.....	195
12.8.2 Release Latency.....	195
12.9 Integration Example.....	196
12.9.1 In sram_controller_unit.sv.....	196
12.10 Debug Support.....	196
12.10.1 Display Statements.....	196

12.10.2 Waveform Analysis.....	197
12.11 Common Issues.....	197
12.11.1 Issue 1: Space Not Released.....	197
12.11.2 Issue 2: Overflow Despite Allocation.....	197
12.11.3 Issue 3: Allocation Pointer Overflow.....	198
12.12 Comparison with Drain Controller.....	198
12.13 Resource Utilization.....	198
12.14 Related Modules.....	199
12.15 Related Documentation.....	199
AXI Read Engine: 06_axi_read_engine.md.....	199
12.16 Revision History.....	199
13 SRAM Controller Specification.....	199
13.1 Overview.....	199
13.1.1 Key Features.....	200
13.1.2 Design Rationale.....	200
13.2 Architecture.....	200
13.2.1 Block Diagram.....	200
13.2.2 Figure 2.8.1: SRAM Controller Block Diagram.....	200
13.2.3 Per-Channel Architecture.....	201
13.3 Parameters.....	201
13.4 Port List.....	202
13.4.1 Clock and Reset.....	202
13.4.2 Allocation Interface.....	202
13.4.3 Write Interface.....	202

13.4.4 Drain Interface.....	203
13.4.5 Read Interface.....	203
13.4.6 Debug Interface.....	204
13.5 Interface.....	204
13.5.1 Clock and Reset.....	204
13.5.2 Allocation Interface (AXI Read Engine Flow Control).....	204
13.5.3 Write Interface (AXI Read Engine → FIFO).....	205
13.5.4 Drain Interface (AXI Write Engine Flow Control).....	205
13.5.5 Read Interface (FIFO → AXI Write Engine).....	206
13.5.6 Debug Interface.....	206
13.6 ID Decode Logic.....	206
13.6.1 Write Valid Decode.....	206
13.6.2 Read/Drain Decode.....	207
13.6.3 Allocation Decode.....	207
13.7 Per-Channel Unit.....	208
13.8 Operation Flows.....	209
13.8.1 Write Flow (AXI Read Data → FIFO).....	209
13.8.2 Read Flow (FIFO → AXI Write Data).....	209
13.9 Allocation vs. Drain Controllers.....	210
13.9.1 Why Separate Controllers?.....	210
13.9.2 Flow Control Comparison.....	210
13.10 Timing Diagrams.....	211
13.10.1 Combined Write and Read Timing.....	211
13.10.2 Write Path (R Data → FIFO).....	212

13.10.3 Read Path (FIFO → W Data).....	212
13.11 Error Conditions.....	213
13.11.1 Invalid Channel ID.....	213
13.11.2 FIFO Overflow.....	213
13.11.3 FIFO Underflow.....	213
13.12 Performance Considerations.....	213
13.12.1 Per-Channel FIFO Benefits.....	213
13.12.2 SRAM Resource Usage.....	213
13.13 Testing.....	214
13.14 Related Documentation.....	214
13.15 Revision History.....	215
14 SRAM Controller Unit.....	215
14.1 Overview.....	215
14.1.1 Key Features.....	216
14.2 Architecture.....	216
14.2.1 Block Diagram.....	216
14.2.2 Figure 1: SRAM Controller Unit Block Diagram.....	216
14.2.3 Component Hierarchy.....	216
14.2.4 Data Flow.....	216
14.2.5 Controller Naming Convention (CRITICAL).....	217
14.3 Parameters.....	217
14.3.1 Derived Parameters.....	217
14.4 Port List.....	217
14.4.1 Clock and Reset.....	217

14.4.2 Allocation Interface (Read Engine Flow Control).....	218
14.4.3 Write Interface (AXI Read Engine to FIFO).....	218
14.4.4 Drain Interface (Write Engine Flow Control).....	218
14.4.5 Read Interface (FIFO to AXI Write Engine).....	219
14.4.6 Debug Interface.....	219
14.5 Operation.....	219
14.5.1 Allocation Flow.....	219
14.5.2 Drain Flow.....	219
14.5.3 Data Available Calculation.....	220
14.6 Integration Example.....	220
14.7 Common Issues.....	221
14.7.1 Issue 1: Space Accounting Mismatch.....	221
14.7.2 Issue 2: Data Available Undercount.....	221
14.8 Related Documentation.....	221
Write Engine: 12_axi_write_engine.md - Data consumer.....	221
14.9 Revision History.....	221
15 Stream Latency Bridge.....	222
15.1 Overview.....	222
15.1.1 Key Features.....	222
15.2 Architecture.....	223
15.2.1 Block Diagram.....	223
15.2.2 Figure 2.10.1: Stream Latency Bridge Block Diagram.....	223
15.2.3 Operation Flow.....	223
15.2.4 Component Diagram.....	223

15.2.5 Backpressure Logic.....	224
15.3 Parameters.....	225
15.3.1 Derived Parameters.....	225
15.4 Port List.....	225
15.4.1 Clock and Reset.....	225
15.4.2 Upstream Interface (from registered FIFO).....	225
15.4.3 Downstream Interface (to consumer).....	226
15.4.4 Status Interface.....	226
15.4.5 Debug Interface.....	226
15.5 Operation.....	226
15.5.1 Glue Logic.....	226
15.5.2 Skid Buffer.....	227
15.5.3 Occupancy Calculation.....	227
15.6 Timing Diagrams.....	227
15.6.1 Backpressure Handling.....	227
15.7 Integration Example.....	228
15.8 Design Rationale.....	229
15.8.1 Why Use a Skid Buffer?.....	229
15.8.2 Why 4-Deep?.....	229
15.9 Related Documentation.....	229
Producer: 06_axi_read_engine.md - Feeds upstream FIFO.....	230
15.10 Revision History.....	230
16 Stream Drain Controller.....	230
16.1 Overview.....	230

16.1.1 Key Features.....	230
16.2 Architecture.....	231
16.2.1 Block Diagram.....	231
16.2.2 Figure 1: Stream Drain Controller Block Diagram.....	231
16.2.3 Virtual FIFO Concept.....	231
16.2.4 Pointer Operations.....	231
16.3 Parameters.....	232
16.3.1 Derived Parameters.....	232
16.4 Port List.....	232
16.4.1 Clock and Reset.....	232
16.4.2 Write Interface (Data Entry).....	232
16.4.3 Read Interface (Drain Requests).....	233
16.4.4 Status Outputs.....	233
16.5 Operation.....	233
16.5.1 Data Entry (Write Side).....	233
16.5.2 Drain Requests (Read Side).....	234
16.5.3 Status Generation.....	234
16.6 Comparison with stream_alloc_ctrl.....	235
16.7 Integration Example.....	235
16.8 Common Issues.....	236
16.8.1 Issue 1: Data Available Undercount.....	236
16.8.2 Issue 2: Drain Request Overflow.....	236
16.9 Related Documentation.....	236
Control Block: See fifo_control in rtl/amba/gaxi/.....	236

16.10 Revision History.....	236
17 AXI Write Engine.....	237
17.1 Overview.....	237
17.1.1 Key Features.....	237
17.2 Architecture.....	237
17.2.1 Block Diagram.....	237
17.2.2 Figure 2.12.1: AXI Write Engine Block Diagram.....	237
17.2.3 Operation Flow.....	238
17.2.4 Key Design Decisions.....	238
17.3 Parameters.....	238
17.3.1 Derived Parameters.....	239
17.4 Port List.....	239
17.4.1 Clock and Reset.....	239
17.4.2 Configuration Interface.....	240
17.4.3 Scheduler Interface (Per-Channel).....	240
17.4.4 Completion Interface (Per-Channel).....	240
17.4.5 SRAM Drain Interface.....	240
17.4.6 SRAM Read Interface.....	241
17.4.7 AXI4 AW Channel.....	241
17.4.8 AXI4 W Channel.....	241
17.4.9 AXI4 B Channel.....	242
17.4.10 Error Interface.....	242
17.4.11 Debug Interface.....	242
17.5 Operation.....	243

17.5.1 Data-Aware Request Masking.....	243
17.5.2 W-Phase Transaction FIFO.....	243
17.5.3 B-Phase Transaction FIFOs.....	243
17.5.4 Outstanding Transaction Tracking.....	244
17.6 Timing Diagrams.....	244
17.6.1 Perfect Streaming - AXI Write Transaction.....	244
17.6.2 Multi-Channel Streaming.....	245
17.7 Integration Example.....	246
17.8 Related Documentation.....	248
Read Engine: 06_axi_read_engine.md - Complementary read datapath.....	248
17.9 Revision History.....	248
18 APB to Descriptor Router.....	248
18.1 Overview.....	249
18.1.1 Key Features.....	249
18.2 Architecture.....	249
18.2.1 Address Map.....	249
18.2.2 Block Diagram.....	249
18.2.3 Figure 2.13.1: APB to Descriptor Block Diagram.....	249
18.2.4 FSM Diagram.....	250
18.2.5 Figure 2.13.2: APB to Descriptor FSM.....	250
18.2.6 Write Flow.....	250
18.2.7 FSM States.....	251
18.3 Parameters.....	251
18.4 Port List.....	251

18.4.1 Clock and Reset.....	251
18.4.2 APB Slave CMD Interface.....	252
18.4.3 APB Slave RSP Interface.....	252
18.4.4 Descriptor Engine APB Ports.....	252
18.4.5 Integration Control.....	253
18.5 Operation.....	253
18.5.1 Address Decode.....	253
18.5.2 64-bit Address Assembly.....	253
18.5.3 Error Conditions.....	253
18.6 Timing Diagrams.....	254
18.6.1 Normal Write Sequence.....	254
18.6.2 Backpressure Handling.....	254
18.6.3 Channel Kick-off Examples.....	256
18.6.4 Multi-Channel Operation.....	257
18.7 Integration Example.....	258
18.8 Common Issues.....	259
18.8.1 Issue 1: APB Response Not Returning.....	259
18.8.2 Issue 2: Wrong Channel Kicked Off.....	259
18.9 Related Documentation.....	259
Register File: PeakRDL-generated stream_regs.sv.....	259
18.10 Revision History.....	259
19 APB Configuration Block.....	260
19.1 Overview.....	260
19.1.1 Key Features.....	260

19.2 Architecture.....	261
19.2.1 Block Diagram.....	261
19.2.2 Figure 1: Stream Config Block Diagram.....	261
19.2.3 Configuration Groups.....	261
19.3 Parameters.....	262
19.4 Port List.....	262
19.4.1 Clock and Reset.....	262
19.4.2 PeakRDL Register Inputs.....	263
19.4.3 Configuration Outputs.....	264
19.5 Operation.....	266
19.5.1 Global Enable Gating.....	266
19.5.2 Global Reset OR.....	266
19.5.3 Address Zero Extension.....	266
19.6 Integration Example.....	266
19.7 Related Documentation.....	267
Monitors: Monitor documentation for each AXI monitor.....	267
19.8 Revision History.....	267
20 Performance Profiler.....	268
20.1 Overview.....	268
20.1.1 Key Features.....	268
20.2 Architecture.....	269
20.2.1 Block Diagram.....	269
20.2.2 Figure 2.15.1: Performance Profiler Block Diagram.....	269
20.2.3 Profiling Modes.....	269

20.2.4 FIFO Entry Format (36-bit).....	270
20.3 Parameters.....	270
20.4 Port List.....	271
20.4.1 Clock and Reset.....	271
20.4.2 Channel Monitoring.....	271
20.4.3 Configuration Interface.....	271
20.4.4 FIFO Read Interface.....	271
20.5 Operation.....	272
20.5.1 Edge Detection.....	272
20.5.2 Timestamp Counter.....	272
20.5.3 Start Time Capture (Elapsed Mode).....	272
20.5.4 Two-Register Read Sequence.....	273
20.6 Integration Example.....	273
20.7 Software Usage.....	274
20.7.1 Example: Measure Channel 0 Transfer Time.....	274
20.8 Common Issues.....	274
20.8.1 Issue 1: FIFO Overflow.....	274
20.8.2 Issue 2: Timestamp Rollover.....	275
20.9 Related Documentation.....	275
Configuration: 14_apb_config.md - Config register mapping.....	275
20.10 Revision History.....	275
21 MonBus AXI-Lite Group.....	275
21.1 Overview.....	276
21.1.1 Key Features.....	276

21.1.2 Simplified from RAPIDS.....	276
21.2 Architecture.....	277
21.2.1 Block Diagram.....	277
21.2.2 Figure 2.16.1: MonBus AXI-Lite Group Block Diagram.....	277
21.2.3 Filter Decision Tree.....	278
21.3 Parameters.....	278
21.4 Port List.....	278
21.4.1 Clock and Reset.....	278
21.4.2 Monitor Bus Input.....	279
21.4.3 AXI-Lite Slave Read Interface.....	279
21.4.4 AXI-Lite Master Write Interface.....	279
21.4.5 Interrupt Output.....	280
21.4.6 Configuration Interface.....	280
21.4.7 Protocol Configuration (per protocol).....	280
21.4.8 Debug/Status.....	281
21.5 Operation.....	282
21.5.1 Packet Type Filtering.....	282
21.5.2 Master Write Address Generation.....	282
21.5.3 Master Write FSM.....	282
21.6 Integration Example.....	283
21.7 Common Issues.....	284
21.7.1 Issue 1: Missing Monitor Packets.....	284
21.7.2 Issue 2: Master Writes Stall.....	284
21.8 Related Documentation.....	284

Configuration: 14_apb_config.md - Register mapping.....	284
21.9 Revision History.....	284
22 Chapter 3: External Interfaces.....	285
22.1 Interface Specifications.....	285
22.1.1 01_axi4_interface_spec.md.....	285
22.1.2 02_axil4_interface_spec.md.....	285
22.1.3 03_apb_interface_spec.md.....	285
22.1.4 05_monbus_interface_spec.md.....	285
22.2 STREAM Interface Summary (stream_top_ch8.sv).....	286
23 AXI4 Interface Specification for STREAM.....	286
23.1 Overview.....	286
23.2 STREAM AXI4 Interface Summary.....	287
23.2.1 Number of Interfaces.....	287
23.2.2 Interface Parameters.....	287
23.2.3 Interface Types and Transfer Modes.....	288
23.2.4 Interface Group Parameter Settings.....	289
23.2.5 Interface Configuration Summary.....	289
23.3 Transfer Mode Specifications.....	290
23.3.1 Mode 1: Simplified Transfer Mode (Control Interfaces).....	290
23.3.2 Mode 2: Flexible Transfer Mode (Data Interfaces).....	291
23.4 Mode 1: Simplified Transfer Mode Specification.....	291
23.4.1 Assumption 1: Address Alignment to Data Bus Width.....	291
23.4.2 Assumption 2: Fixed Transfer Size.....	292
23.5 Mode 2: Flexible Transfer Mode Specification.....	292

23.5.1 Assumption 1: 4-Byte Address Alignment.....	292
23.5.2 Assumption 2: Multiple Transfer Sizes.....	293
23.5.3 Assumption 3: Progressive Alignment Strategy.....	293
23.5.4 Assumption 4: Chunk Enable Support.....	293
23.6 Common Protocol Assumptions (Both Modes).....	294
23.6.1 Assumption 1: Incrementing Bursts Only.....	294
23.6.2 Assumption 2: No Address Wraparound.....	294
23.7 Flexible Mode: Address Calculation Examples.....	295
23.7.1 Progressive Alignment Examples.....	295
23.7.2 Chunk Enable Pattern Examples.....	296
23.8 Master Read Interface Specification.....	296
23.8.1 Read Address Channel (AR).....	296
23.8.2 Read Data Channel (R).....	297
23.9 Master Write Interface Specification.....	298
23.9.1 Write Address Channel (AW).....	298
23.9.2 Write Data Channel (W).....	299
23.9.3 Write Response Channel (B).....	299
23.10 Address Calculation Rules.....	300
23.10.1 Simplified Mode Address Generation.....	300
23.10.2 Flexible Mode Address Generation.....	300
23.10.3 4KB Boundary Considerations (Both Modes).....	300
23.11 Write Strobe Generation.....	301
23.11.1 Simplified Mode Strobe Generation.....	301
23.11.2 Flexible Mode Strobe Generation.....	301

23.12 Response Codes.....	302
23.12.1 Response Code Specification.....	302
23.13 Implementation Benefits.....	302
23.13.1 Simplified Mode Benefits.....	302
23.13.2 Flexible Mode Benefits.....	303
23.13.3 Mode Selection Guidelines.....	303
23.14 Validation Requirements.....	304
23.14.1 Simplified Mode Validation.....	304
23.14.2 Flexible Mode Validation.....	304
23.14.3 Common Validation.....	304
23.15 Performance Characteristics.....	305
23.15.1 Simplified Mode Performance.....	305
23.15.2 Flexible Mode Performance.....	305
23.15.3 Performance Optimization Strategy.....	305
23.16 This dual-mode approach provides the best of both worlds: simplified, predictable operation for control interfaces and flexible, high-performance operation for data interfaces.....	306
23.17 Revision History.....	306
24 AXI4-Lite Interface Specification and Assumptions.....	306
24.1 Overview.....	306
24.2 Interface Summary.....	306
24.2.1 Number of Interfaces.....	306
24.2.2 Interface Parameters.....	307
24.3 Core Protocol Assumptions.....	307

24.3.1 Inherent AXI4-Lite Simplifications.....	307
24.3.2 Implementation Assumptions.....	307
24.4 Master Read Interface Specification.....	309
24.4.1 Read Address Channel (AR).....	309
24.4.2 Read Data Channel (R).....	309
24.4.3 AXI4-Lite Simplifications (Read).....	309
24.5 Master Write Interface Specification.....	310
24.5.1 Write Address Channel (AW).....	310
24.5.2 Write Data Channel (W).....	310
24.5.3 Write Response Channel (B).....	311
24.5.4 AXI4-Lite Simplifications (Write).....	311
24.6 Address Requirements.....	312
24.6.1 Address Alignment Rules.....	312
24.6.2 Address Validation Examples.....	312
24.7 Response Codes.....	312
24.7.1 Response Code Specification.....	312
24.7.2 Response Usage Guidelines.....	313
24.8 Protection Signal Usage.....	313
24.8.1 Protection Signal Encoding.....	313
24.8.2 Common Protection Patterns.....	313
24.9 Implementation Benefits.....	314
24.9.1 Simplified Control Register Interface.....	314
24.9.2 Address Decode Implementation.....	314
24.9.3 Error Generation Logic.....	314

24.10 Timing Requirements.....	315
24.10.1 Handshake Protocol.....	315
24.10.2 Channel Dependencies.....	315
24.10.3 Reset Behavior.....	316
24.11 Validation Requirements.....	316
24.11.1 Functional Validation.....	316
24.11.2 Timing Validation.....	316
24.11.3 Error Injection Testing.....	317
24.12 Example Transactions.....	317
24.12.1 64-bit Register Write.....	317
24.12.2 AW Transaction Flow.....	317
24.12.3 64-bit Register Read.....	318
24.12.4 AR Transaction Flow.....	318
24.12.5 Misaligned Address Example.....	319
24.13 Common Use Cases.....	319
24.13.1 Typical Applications.....	319
24.13.2 Performance Considerations.....	319
24.14 Revision History.....	320
25 APB Programming Interface Specification for STREAM.....	320
25.1 Overview.....	320
25.2 Interface Summary.....	320
25.2.1 STREAM Programming Interface.....	320
25.2.2 Interface Parameters.....	321
25.3 Core Protocol Assumptions.....	321

25.3.1 Inherent APB Simplifications.....	321
25.3.2 Implementation Assumptions.....	322
25.4 Interface Signal Specification.....	323
25.4.1 Clock and Reset Signals.....	323
25.4.2 Data Signals.....	324
25.4.3 Response Signals.....	324
25.5 Address Space Configuration.....	324
25.5.1 Address Range Specification.....	324
25.5.2 Address Decode Implementation.....	324
25.6 Transaction Protocol.....	325
25.6.1 2-Phase Transaction States.....	325
25.6.2 State Transitions.....	325
25.6.3 Transaction Timing Requirements.....	326
25.7 Advanced Features.....	326
25.7.1 Dynamic Clock Gating.....	326
25.7.2 Clock Domain Crossing (CDC) Handshaking.....	326
25.7.3 Buffering and Flow Control.....	327
25.8 Error Handling.....	327
25.8.1 Error Response Specification.....	327
25.8.2 Error Response Timing.....	328
25.9 Reset Behavior.....	328
25.9.1 Reset Requirements.....	328
25.10 Implementation Variants.....	328
25.10.1 Available Implementations.....	328

25.11 Performance Characteristics.....	329
25.11.1 Latency Specifications.....	329
25.11.2 Throughput Specifications.....	329
25.11.3 Power Consumption.....	329
25.12 Validation Requirements.....	330
25.12.1 Functional Validation.....	330
25.12.2 Timing Validation.....	330
25.13 Example Transactions.....	331
25.13.1 32-bit Register Write.....	331
25.13.2 32-bit Register Read with Wait State.....	331
25.14 Revision History.....	331
26 Monitor Bus Architecture and Event Code Organization.....	332
26.1 STREAM-Specific Context.....	332
26.2 Overview.....	332
26.3 Interface Summary.....	332
26.3.1 Number of Interfaces.....	332
26.3.2 Interface Parameters.....	332
26.4 Core Design Assumptions.....	333
26.4.1 Assumption 1: Hierarchical Event Organization.....	333
26.4.2 Assumption 2: Protocol Isolation.....	333
26.4.3 Assumption 3: Two-Tier Memory Architecture.....	334
26.4.4 Assumption 4: Configurable Packet Routing.....	334
26.5 Interface Signal Specification.....	334
26.5.1 Monitor Bus Output Interface.....	334

26.5.2 Protocol Input Interfaces.....	335
26.5.3 Control and Status Signals.....	336
26.6 Packet Format and Field Allocation.....	336
26.6.1 64-bit Monitor Bus Packet Structure.....	336
26.6.2 Packet Type Definitions.....	337
26.7 Protocol-Specific Event Codes.....	338
26.7.1 AXI Protocol Events.....	338
26.7.2 APB Protocol Events.....	340
26.7.3 MNOC Protocol Events.....	344
26.7.4 ARB Protocol Events.....	346
26.7.5 CORE Protocol Events.....	351
26.8 Memory Architecture and Packet Routing.....	356
26.8.1 Two-Tier Memory Architecture.....	356
26.8.2 Routing Configuration.....	357
26.8.3 Address Space Management.....	357
26.8.4 Transaction State and Bus Transaction Structure.....	358
26.8.5 APB Transaction Phases.....	360
26.8.6 APB Protection Types.....	361
26.8.7 MNOC Payload Types.....	361
26.8.8 MNOC ACK Types.....	361
26.8.9 ARB State Types.....	362
26.8.10 CORE State Types.....	362
26.9 Configuration and Control.....	363
26.9.1 Monitor Configuration Registers.....	363

26.9.2 Protocol-Specific Configuration.....	364
26.10 Validation Requirements.....	366
26.10.1 Functional Validation.....	366
26.10.2 Performance Validation.....	367
26.10.3 Error Handling Validation.....	367
26.11 Usage Examples.....	367
26.11.1 Creating Monitor Packets.....	367
26.11.2 Packet Decoding.....	368
26.11.3 Monitor Bus Packet Helper Functions.....	368
26.12 Debug and Monitoring Signals.....	371
26.12.1 Essential Debug Signals.....	371
26.12.2 Performance Counters.....	372
26.13 Protocol Coverage Summary.....	372
26.13.1 Complete Protocol Event Matrix.....	372
26.14 Total Event Codes: 544 defined across all protocols and packet types....	373
26.15 Revision History.....	373
27 STREAM Register Map.....	373
27.1 Overview.....	373
27.2 Address Space Layout.....	373
27.3 Register Details.....	374
27.3.1 Channel Kick-off Registers (0x000 - 0x03F).....	374
27.3.2 Global Control and Status (0x100 - 0x11F).....	376
27.3.3 Per-Channel Control and Status (0x120 - 0x17F).....	377
27.3.4 Engine Completion and Error Status (0x170 - 0x17F).....	379

27.3.5 Monitor FIFO Status (0x180 - 0x1FF).....	380
27.3.6 Scheduler Configuration (0x200 - 0x21F).....	381
27.3.7 Descriptor Engine Configuration (0x220 - 0x23F).....	382
27.3.8 Descriptor AXI Monitor Configuration (0x240 - 0x25F).....	383
27.3.9 Read Engine AXI Monitor Configuration (0x260 - 0x27F).....	385
27.3.10 Write Engine AXI Monitor Configuration (0x280 - 0x29F).....	388
27.3.11 AXI Transfer Configuration (0x2A0 - 0x2AF).....	390
27.3.12 Performance Profiler Configuration (0x2B0 - 0x2BF).....	391
27.4 Typical Usage Flow.....	392
27.4.1 Initialization.....	392
27.4.2 Start Transfer.....	392
27.4.3 Poll for Completion.....	392
27.4.4 Error Handling.....	393
27.5 Register Summary Table.....	393
27.6 PeakRDL Generation.....	393
28 Chapter 5: Programming Models.....	394
28.1 Contents.....	394
28.1.1 01_initialization.md.....	394
28.1.2 02_single_transfer.md.....	394
28.1.3 03_chained_transfers.md.....	395
28.1.4 04_multi_channel.md.....	395
28.1.5 05_error_handling.md.....	395
28.2 Planned (Future).....	395
28.2.1 06_performance_tuning.md.....	395

28.2.2 07_software_examples.md.....	395
29 Chapter 6: Configuration Reference.....	396
29.1 Overview.....	396
29.1.1 Configuration Categories.....	396
29.2 1. Channel Control Configuration.....	397
29.2.1 cfg_channel_enable[NUM_CHANNELS-1:0].....	397
29.2.2 cfg_channel_reset[NUM_CHANNELS-1:0].....	397
29.3 2. Scheduler Configuration.....	398
29.3.1 cfg_sched_timeout_cycles[15:0].....	398
29.3.2 cfg_sched_enable.....	398
29.3.3 cfg_sched_timeout_enable.....	398
29.3.4 cfg_sched_err_enable.....	399
29.3.5 cfg_sched_compl_enable.....	399
29.3.6 cfg_sched_perf_enable.....	399
29.4 3. Descriptor Engine Configuration.....	399
29.4.1 cfg_desceng_enable.....	399
29.4.2 cfg_desceng_prefetch.....	399
29.4.3 cfg_desceng_fifo_thresh[3:0].....	400
29.4.4 cfg_desceng_addr0_base[31:0].....	400
29.4.5 cfg_desceng_addr0_limit[31:0].....	400
29.4.6 cfg_desceng_addr1_base[31:0].....	400
29.4.7 cfg_desceng_addr1_limit[31:0].....	401
29.5 4. AXI Monitor Configuration.....	401
29.5.1 4.1 Descriptor AXI Monitor (cfg_desc_mon_*).....	401

29.5.2 4.2 Read Engine Monitor (cfg_rdeng_mon_*).....	405
29.5.3 4.3 Write Engine Monitor (cfg_wreng_mon_*).....	405
29.6 5. AXI Transfer Configuration.....	405
29.6.1 cfg_axi_rd_xfer_beats[7:0].....	405
29.6.2 cfg_axi_wr_xfer_beats[7:0].....	406
29.7 6. Performance Profiler Configuration.....	406
29.7.1 cfg_perf_enable.....	406
29.7.2 cfg_perf_mode.....	406
29.7.3 cfg_perf_clear.....	406
29.8 7. Configuration Presets.....	407
29.8.1 7.1 Minimal Configuration (Tutorial/Embedded).....	407
29.8.2 7.2 Balanced Configuration (Typical FPGA).....	407
29.8.3 7.3 High-Performance Configuration (ASIC/Datacenter).....	408
29.8.4 7.4 Debug Configuration (Verbose Monitoring).....	409
29.9 8. Configuration Best Practices.....	410
29.9.1 8.1 Monitor Configuration Guidelines.....	410
29.9.2 8.2 Timeout Configuration.....	411
29.9.3 8.3 Prefetch Configuration.....	411
29.9.4 8.4 Burst Size Selection.....	412
29.10 9. Configuration Register Map Summary.....	412
29.11 10. Software Configuration Examples.....	413
29.11.1 10.1 C/C++ Initialization (Minimal).....	413
29.11.2 10.2 C/C++ Initialization (Balanced).....	414
29.12 11. Troubleshooting Configuration Issues.....	415

29.12.1 Problem: No transfers occurring.....	415
29.12.2 Problem: Timeout errors.....	415
29.12.3 Problem: MonBus overflow.....	415
29.12.4 Problem: Low throughput.....	415
29.13 Related Documentation.....	416
30 Product Requirements Document (PRD).....	416
30.1 STREAM - Scatter-gather Transfer Rapid Engine for AXI Memory.....	416
30.2 1. Executive Summary.....	416
30.2.1 1.1 Quick Stats.....	416
30.2.2 1.2 Project Goals.....	416
30.3 2. Key Design Principles.....	417
30.3.1 2.1 Simplifications from RAPIDS.....	417
30.3.2 2.2 Tutorial-Friendly Features.....	417
30.4 3. Architecture Overview.....	418
30.4.1 3.1 Top-Level Block Diagram.....	418
30.4.2 3.2 Data Flow.....	418
30.5 4. Interfaces.....	419
30.5.1 4.1 External Interfaces.....	419
30.5.2 4.2 Descriptor Format.....	419
30.6 5. Key Components.....	420
30.6.1 5.1 Descriptor Engine (APB-Only for STREAM).....	420
30.6.2 5.2 Scheduler Group (Integration Wrapper).....	420
30.6.3 5.3 Scheduler (Simplified from RAPIDS).....	422
30.6.4 5.3 AXI Read Engine (Streaming Pipeline - NO FSM).....	423

30.6.5 5.4 AXI Write Engine (Streaming Pipeline - NO FSM).....	424
30.6.6 5.5 Simple SRAM.....	425
30.7 6. Configuration and Control.....	425
30.7.1 6.1 APB Register Map.....	425
30.7.2 6.2 Channel Configuration.....	426
30.8 7. Resource Sharing and Arbitration.....	426
30.8.1 7.1 Shared Resources.....	426
30.8.2 7.2 Arbitration Strategy.....	426
30.9 8. Error Detection and Recovery.....	426
30.9.1 8.1 Error Types.....	426
30.9.2 8.2 Error Recovery.....	427
30.10 9. MonBus Integration.....	427
30.10.1 9.1 Standard MonBus Format.....	427
30.10.2 9.2 STREAM Event Codes.....	427
30.10.3 9.3 Default Configuration.....	428
30.11 10. Design Constraints.....	428
30.11.1 10.1 Tutorial Constraints (Intentional Simplifications).....	428
30.11.2 10.2 Implementation Constraints.....	428
30.12 11. Verification Strategy.....	428
30.12.1 11.1 Test Organization.....	428
30.12.2 11.2 Test Levels.....	429
30.13 12. Performance Characteristics.....	429
30.13.1 12.1 Throughput by Engine Version.....	429
30.13.2 12.2 Latency.....	430

30.13.3 12.3 Resource Utilization by Engine Version.....	430
30.14 13. Development Roadmap.....	430
30.14.1 13.1 Phase 1: Foundation (Current).....	430
30.14.2 13.2 Phase 2: Core Blocks.....	431
30.14.3 13.3 Phase 3: Data Path.....	431
30.14.4 13.4 Phase 4: Integration.....	431
30.14.5 13.5 Phase 5: Multi-Channel.....	431
30.14.6 13.6 Phase 6: Advanced Engines (Future - V2/V3).....	431
30.15 14. Educational Value.....	432
30.15.1 14.1 Learning Objectives.....	432
30.15.2 14.2 Progression Path.....	432
30.16 15. Success Criteria.....	433
30.16.1 15.1 Functional.....	433
30.16.2 15.2 Quality.....	433
30.16.3 15.3 Performance.....	433
30.17 16. Open Questions (For Review).....	433
30.17.1 16.1 Descriptor Engine Adaptation.....	433
30.17.2 16.2 AXI Descriptor Master.....	433
30.17.3 16.3 Channel Arbitration.....	433
30.17.4 16.4 SRAM Partitioning.....	434
30.18 16. Attribution and Contribution Guidelines.....	434
30.18.1 16.1 Git Commit Attribution.....	434
30.19 16.2 PDF Generation Location.....	434
30.20 17. References.....	435

30.20.1 17.1 Internal Documentation.....	435
30.20.2 17.2 External References.....	435
30.21 Navigation.....	435
31 Claude Code Guide: STREAM Subsystem.....	435
31.1 Quick Context.....	436
31.2 Global Requirements Reference.....	436
31.3 Critical Rules for This Subsystem.....	436
31.3.1 Rule #0: Attribution Format for Git Commits.....	436
31.3.2 Rule #0.1: TUTORIAL FOCUS - Intentional Simplifications.....	436
31.3.3 Rule #0.1: Testbench Location and Test Structure (MANDATORY)....	437
31.3.4 Rule #0.2: Three Mandatory TB Methods (MANDATORY).....	437
31.3.5 Rule #1: REUSE from RAPIDS Where Appropriate.....	438
31.3.6 Rule #2: Descriptor Format is DIFFERENT from RAPIDS.....	438
31.3.7 Rule #3: Know the Shared Resources.....	438
31.4 Architecture Quick Reference.....	438
31.4.1 Block Organization.....	438
31.4.2 Module Status.....	439
31.5 Common User Questions and Responses.....	440
31.5.1 Q: “How is STREAM different from RAPIDS?”	440
31.5.2 Q: “How do I kick off a transfer?”	440
31.5.3 Q: “How many channels can I use?”	441
31.5.4 Q: “What’s the descriptor format?”	441
31.5.5 Q: “How do I run STREAM tests?”	442
31.6 Integration Patterns.....	442

31.6.1 Pattern 1: Basic STREAM Instantiation.....	442
31.6.2 Pattern 2: Descriptor Creation (Software Model).....	443
31.6.3 Pattern 3: MonBus Integration.....	444
31.7 Anti-Patterns to Catch.....	444
31.7.1 ✗ Anti-Pattern 1: Adding Alignment Fixup.....	444
31.7.2 ✗ Anti-Pattern 2: Using Length in Bytes.....	445
31.7.3 ✗ Anti-Pattern 3: Circular Buffer Descriptors.....	445
31.7.4 ✗ Anti-Pattern 4: Assuming Exclusive Channel Access.....	445
31.8 Debugging Workflow.....	445
31.8.1 Issue: Descriptor Not Fetched.....	445
31.8.2 Issue: Data Transfer Stalls.....	446
31.8.3 Issue: Chained Descriptors Not Following.....	446
31.9 Testing Guidance.....	446
31.9.1 Test Organization.....	446
31.9.2 Test Levels.....	446
31.10 Key Documentation Links.....	447
31.10.1 Always Reference These.....	447
31.11 Quick Commands.....	447
31.12 Remember.....	447
31.13 PDF Generation Location.....	448
32 STREAM Register Definitions.....	448
32.1 Directory Structure.....	448
32.2 Register Generation Workflow.....	449
32.2.1 Phase 1: Define Register Map (Future).....	449

32.2.2 Phase 2: PeakRDL-Generated Registers (Future).....	449
32.3 Register Map (Planned).....	450
32.3.1 Global Registers.....	450
32.3.2 Channel Registers ($8 \times 0x10$ bytes).....	450
32.4 Integration Pattern.....	451
32.4.1 PeakRDL-Generated Implementation (Future):.....	451
32.5 Reference Examples.....	451
32.6 Status.....	451

List of Figures

Figure 1.1.1: STREAM Architecture Overview.....	65
Figure 1.1.2: Single Channel Transfer Sequence.....	67
Figure 2.1.1: STREAM Core Block Diagram.....	100
Figure 2.4.1: Scheduler Block Diagram.....	144
Figure 2.4.2: Scheduler FSM.....	153
Figure 2.4.3: Scheduler Normal Transfer Timing.....	160
Figure 2.4.4: Scheduler Descriptor Chaining Timing.....	161
Figure 2.7.1: Stream Allocation Controller Block Diagram.....	188
Figure 2.8.1: SRAM Controller Block Diagram.....	200
Figure 1: SRAM Controller Unit Block Diagram.....	216
Figure 2.10.1: Stream Latency Bridge Block Diagram.....	223
Figure 1: Stream Drain Controller Block Diagram.....	231
Figure 2.12.1: AXI Write Engine Block Diagram.....	237
Figure 2.13.1: APB to Descriptor Block Diagram.....	249
Figure 2.13.2: APB to Descriptor FSM.....	250
Figure 1: Stream Config Block Diagram.....	261
Figure 2.15.1: Performance Profiler Block Diagram.....	269
Figure 2.16.1: MonBus AXI-Lite Group Block Diagram.....	277

List of Tables

Table 1: Related Documents and Specifications.....	61
Table 2: STREAM MAS Document Revision History.....	63
Table 3: STREAM External Interfaces.....	68
Table 4: STREAM Internal MonBus Interface.....	69
Table 5: STREAM Resource Utilization Breakdown.....	69
Table 6: STREAM Architecture Overview Revision History.....	72
Table 7: Clock and Reset Signals.....	73
Table 8: APB Programming Interface Signals.....	73
Table 9: Per-Channel Configuration Signals.....	74
Table 10: Global Scheduler Configuration Signals.....	74
Table 11: Descriptor Engine Configuration Signals.....	75
Table 12: AXI Monitor Signal Prefixes.....	76
Table 13: AXI Monitor Configuration Signal Suffixes.....	76
Table 14: AXI Transfer Configuration Signals.....	77
Table 15: Performance Profiler Configuration Signals.....	77
Table 16: Per-Channel Status Signals.....	78
Table 17: Performance Profiler Interface Signals.....	79
Table 18: Descriptor AXI Master AR Channel Signals.....	80
Table 19: R Channel.....	81
Table 20: AR Channel.....	81
Table 21: R Channel.....	82
Table 22: AW Channel.....	83
Table 23: W Channel.....	84
Table 24: B Channel.....	84
Table 25: Descriptor AXI Monitor Status.....	85
Table 26: Read Engine AXI Monitor Status.....	85
Table 27: Write Engine AXI Monitor Status.....	86
Table 28: Unified Monitor Bus Interface.....	86
Table 29: Port Count Summary.....	87
Table 30: Default Parameter Values.....	88
Table 31: STREAM Top-Level Port List Revision History.....	89
Table 32: Reset Recovery.....	92
Table 33: Functional Unit Blocks.....	92
Table 34: Integration Blocks.....	92
Table 35: Clocks and Reset Specification Revision History.....	98
Table 36: Primary Configuration.....	103

Table 37: Monitor Control.....	104
Table 38: Outstanding Transaction Limits.....	104
Table 39: AXI Skid Buffer Depths.....	104
Table 40: MonBus Agent IDs.....	105
Table 41: Clock and Reset.....	105
Table 42: APB Programming Interface.....	105
Table 43: Configuration Interface.....	106
Table 44: Configuration Interface.....	106
Table 45: Configuration Interface.....	107
Table 46: Configuration Interface.....	107
Table 47: Status Interface.....	109
Table 48: Status Interface.....	109
Table 49: Status Interface.....	110
Table 50: AXI4 Master - Descriptor Fetch.....	110
Table 51: AXI4 Master - Descriptor Fetch.....	111
Table 52: AXI4 Master - Data Read.....	111
Table 53: AXI4 Master - Data Read.....	112
Table 54: AXI4 Master - Data Write.....	113
Table 55: AXI4 Master - Data Write.....	114
Table 56: AXI4 Master - Data Write.....	114
Table 57: Status/Debug Outputs.....	115
Table 58: Status/Debug Outputs.....	115
Table 59: Status/Debug Outputs.....	115
Table 60: Unified Monitor Bus Interface.....	116
Table 61: Resource Utilization.....	120
Table 62: STREAM Core Specification Revision History.....	123
Table 63: Parameters.....	125
Table 64: Monitor Bus Agent IDs.....	125
Table 65: Clock and Reset.....	126
Table 66: APB Programming Interface.....	126
Table 67: Configuration Interface.....	126
Table 68: Global Scheduler Configuration.....	126
Table 69: Descriptor Engine Configuration.....	127
Table 70: Status Interface.....	128
Table 71: Shared Descriptor AXI4 Master Read Interface.....	128
Table 72: Shared Descriptor AXI4 Master Read Interface.....	129
Table 73: Shared Data Read Interface.....	129
Table 74: Shared Data Write Interface.....	130

Table 75: Unified Monitor Bus Interface.....	130
Table 76: Scheduler Group Array Revision History.....	133
Table 77: Parameters.....	135
Table 78: Monitor Bus Agent IDs.....	135
Table 79: Clock and Reset.....	136
Table 80: APB Programming Interface.....	136
Table 81: Configuration Interface.....	136
Table 82: Scheduler Configuration.....	136
Table 83: Descriptor Engine Configuration.....	137
Table 84: Status Interface.....	137
Table 85: Descriptor AXI Interface.....	138
Table 86: Descriptor AXI Interface.....	138
Table 87: Data Read Interface.....	139
Table 88: Data Write Interface.....	139
Table 89: Monitor Bus Interface.....	140
Table 90: Scheduler Group Revision History.....	142
Table 91: Clock and Reset.....	145
Table 92: Configuration Interface.....	146
Table 93: Status Interface.....	146
Table 94: Descriptor Engine Interface.....	146
Table 95: Data Read Interface.....	147
Table 96: Data Read Interface.....	147
Table 97: Data Write Interface.....	147
Table 98: Data Write Interface.....	148
Table 99: Error Signals.....	148
Table 100: Monitor Bus Interface.....	148
Table 101: Revision History.....	164
Table 102: Descriptor Format.....	166
Table 103: Parameters.....	167
Table 104: Monitor Bus Parameters.....	167
Table 105: Clock and Reset.....	168
Table 106: APB Programming Interface.....	168
Table 107: Scheduler Interface.....	168
Table 108: Enhanced Control Outputs.....	169
Table 109: AXI4 AR Channel.....	169
Table 110: AXI4 R Channel.....	170
Table 111: Configuration Interface.....	170
Table 112: Status Interface.....	171

Table 113: Monitor Bus Interface.....	171
Table 114: Descriptor Engine Revision History.....	175
Table 115: Parameters.....	177
Table 116: Derived Parameters.....	178
Table 117: Clock and Reset.....	178
Table 118: Configuration Interface.....	178
Table 119: Scheduler Interface.....	178
Table 120: Completion Interface.....	179
Table 121: SRAM Allocation Interface.....	179
Table 122: SRAM Write Interface.....	179
Table 123: AXI4 AR Channel.....	180
Table 124: AXI4 R Channel.....	180
Table 125: Error Interface.....	180
Table 126: Debug Interface.....	181
Table 127: AXI Read Engine Revision History.....	186
Table 128: Parameters.....	190
Table 129: Clock and Reset.....	191
Table 130: Write Interface.....	191
Table 131: Read Interface.....	192
Table 132: Status Outputs.....	192
Table 133: Write Interface.....	192
Table 134: Read Interface.....	193
Table 135: Status Outputs.....	193
Table 136: Comparison with Drain Controller.....	198
Table 137: Stream Allocation Controller Revision History.....	199
Table 138: Clock and Reset.....	202
Table 139: Allocation Interface.....	202
Table 140: Write Interface.....	202
Table 141: Drain Interface.....	203
Table 142: Read Interface.....	203
Table 143: Debug Interface.....	204
Table 144: Revision History.....	215
Table 145: Parameters.....	217
Table 146: Derived Parameters.....	217
Table 147: Clock and Reset.....	217
Table 148: Allocation Interface.....	218
Table 149: Write Interface.....	218
Table 150: Drain Interface.....	218

Table 151: Read Interface.....	219
Table 152: Debug Interface.....	219
Table 153: SRAM Controller Unit Revision History.....	221
Table 154: Parameters.....	225
Table 155: Derived Parameters.....	225
Table 156: Clock and Reset.....	225
Table 157: Upstream Interface.....	225
Table 158: Downstream Interface.....	226
Table 159: Status Interface.....	226
Table 160: Debug Interface.....	226
Table 161: Stream Latency Bridge Revision History.....	230
Table 162: Parameters.....	232
Table 163: Derived Parameters.....	232
Table 164: Clock and Reset.....	232
Table 165: Write Interface.....	232
Table 166: Read Interface.....	233
Table 167: Status Outputs.....	233
Table 168: Comparison with stream_alloc_ctrl.....	235
Table 169: Stream Drain Controller Revision History.....	236
Table 170: Parameters.....	238
Table 171: Derived Parameters.....	239
Table 172: Clock and Reset.....	239
Table 173: Configuration Interface.....	240
Table 174: Scheduler Interface.....	240
Table 175: Completion Interface.....	240
Table 176: SRAM Drain Interface.....	240
Table 177: SRAM Read Interface.....	241
Table 178: AXI4 AW Channel.....	241
Table 179: AXI4 W Channel.....	241
Table 180: AXI4 B Channel.....	242
Table 181: Error Interface.....	242
Table 182: Debug Interface.....	242
Table 183: AXI Write Engine Revision History.....	248
Table 184: Parameters.....	251
Table 185: Clock and Reset.....	251
Table 186: APB Slave CMD Interface.....	252
Table 187: APB Slave RSP Interface.....	252
Table 188: Descriptor Engine APB Ports.....	252

Table 189: Integration Control.....	253
Table 190: APB to Descriptor Router Revision History.....	259
Table 191: Parameters.....	262
Table 192: Clock and Reset.....	262
Table 193: PeakRDL Register Inputs.....	263
Table 194: PeakRDL Register Inputs.....	263
Table 195: PeakRDL Register Inputs.....	263
Table 196: PeakRDL Register Inputs.....	264
Table 197: PeakRDL Register Inputs.....	264
Table 198: Configuration Outputs.....	264
Table 199: Configuration Outputs.....	264
Table 200: Configuration Outputs.....	265
Table 201: Configuration Outputs.....	265
Table 202: Configuration Outputs.....	265
Table 203: APB Configuration Block Revision History.....	267
Table 204: FIFO Entry Format.....	270
Table 205: Parameters.....	270
Table 206: Clock and Reset.....	271
Table 207: Channel Monitoring.....	271
Table 208: Configuration Interface.....	271
Table 209: FIFO Read Interface.....	271
Table 210: Performance Profiler Revision History.....	275
Table 211: Parameters.....	278
Table 212: Clock and Reset.....	278
Table 213: Monitor Bus Input.....	279
Table 214: AXI-Lite Slave Read Interface.....	279
Table 215: AXI-Lite Master Write Interface.....	279
Table 216: Interrupt Output.....	280
Table 217: Configuration Interface.....	280
Table 218: Protocol Configuration.....	281
Table 219: Protocol Configuration.....	281
Table 220: Protocol Configuration.....	281
Table 221: Debug/Status.....	281
Table 222: MonBus AXI-Lite Group Revision History.....	284
Table 223: Number of Interfaces.....	287
Table 224: Interface Parameters.....	287
Table 225: Interface Types and Transfer Modes.....	288
Table 226: Interface Group Parameter Settings.....	289

Table 227: Interface Configuration Summary.....	289
Table 228: Simplified Mode Assumptions.....	291
Table 229: Flexible Mode Assumptions.....	291
Table 230: Assumption 1: Address Alignment to Data Bus Width.....	291
Table 231: Assumption 2: Fixed Transfer Size.....	292
Table 232: Assumption 1: 4-Byte Address Alignment.....	292
Table 233: Assumption 2: Multiple Transfer Sizes.....	293
Table 234: Assumption 3: Progressive Alignment Strategy.....	293
Table 235: Assumption 4: Chunk Enable Support.....	293
Table 236: Assumption 1: Incrementing Bursts Only.....	294
Table 237: Assumption 2: No Address Wraparound.....	294
Table 238: Progressive Alignment Examples.....	295
Table 239: Progressive Alignment Examples.....	295
Table 240: Chunk Enable Pattern Examples.....	296
Table 241: Read Address Channel.....	296
Table 242: Read Data Channel.....	297
Table 243: Write Address Channel.....	298
Table 244: Write Data Channel.....	299
Table 245: Write Response Channel.....	299
Table 246: Simplified Mode Address Generation.....	300
Table 247: Flexible Mode Address Generation.....	300
Table 248: 4KB Boundary Considerations.....	300
Table 249: Simplified Mode Strobe Generation.....	301
Table 250: Flexible Mode Strobe Generation.....	301
Table 251: Response Code Specification.....	302
Table 252: Simplified Mode Benefits.....	302
Table 253: Flexible Mode Benefits.....	303
Table 254: Mode Selection Guidelines.....	303
Table 255: Simplified Mode Validation.....	304
Table 256: Flexible Mode Validation.....	304
Table 257: Common Validation.....	304
Table 258: Simplified Mode Performance.....	305
Table 259: Flexible Mode Performance.....	305
Table 260: AXI4 Interface Specification for STREAM Revision History.....	306
Table 261: Interface Parameters.....	307
Table 262: Inherent AXI4-Lite Simplifications.....	307
Table 263: Assumption 1: Address Alignment to Data Bus Width.....	307
Table 264: Assumption 2: Fixed Transfer Size.....	308

Table 265: Assumption 3: No Address Wraparound.....	308
Table 266: Assumption 4: Standard Protection Attributes.....	308
Table 267: Read Address Channel.....	309
Table 268: Read Data Channel.....	309
Table 269: AXI4-Lite Simplifications.....	309
Table 270: Write Address Channel.....	310
Table 271: Write Data Channel.....	310
Table 272: Write Response Channel.....	311
Table 273: AXI4-Lite Simplifications.....	311
Table 274: Address Alignment Rules.....	312
Table 275: Address Validation Examples.....	312
Table 276: Response Code Specification.....	312
Table 277: Response Usage Guidelines.....	313
Table 278: Protection Signal Encoding.....	313
Table 279: Common Protection Patterns.....	313
Table 280: Simplified Control Register Interface.....	314
Table 281: Address Decode Implementation.....	314
Table 282: Error Generation Logic.....	314
Table 283: Handshake Protocol.....	315
Table 284: Channel Dependencies.....	315
Table 285: Reset Behavior.....	316
Table 286: Functional Validation.....	316
Table 287: Timing Validation.....	316
Table 288: Error Injection Testing.....	317
Table 289: 64-bit Register Write.....	317
Table 290: AW Transaction Flow.....	317
Table 291: 64-bit Register Read.....	318
Table 292: AR Transaction Flow.....	318
Table 293: Misaligned Address Example.....	319
Table 294: Typical Applications.....	319
Table 295: Performance Considerations —.....	319
Table 296: AXI4-Lite Interface Specification and Assumptions Revision History	320
Table 297: STREAM Programming Interface.....	321
Table 298: Interface Parameters.....	321
Table 299: Assumption 1: Word-Aligned Access Only.....	322
Table 300: Assumption 2: Standard Transfer Sizes.....	322
Table 301: Assumption 3: Single-Cycle Default Operation.....	322
Table 302: Clock and Reset Signals ### Address and Control Signals.....	323

Table 303: Address and Control Signals.....	323
Table 304: Data Signals.....	324
Table 305: Response Signals.....	324
Table 306: Address Range Specification.....	324
Table 307: Address Decode Implementation.....	324
Table 308: 2-Phase Transaction States.....	325
Table 309: State Transitions.....	325
Table 310: Transaction Timing Requirements.....	326
Table 311: Dynamic Clock Gating.....	326
Table 312: Clock Domain Crossing (CDC) Handshaking.....	326
Table 313: Buffering and Flow Control.....	327
Table 314: Error Response Specification.....	327
Table 315: Error Response Timing.....	328
Table 316: Reset Requirements.....	328
Table 317: Available Implementations.....	328
Table 318: Latency Specifications.....	329
Table 319: Throughput Specifications.....	329
Table 320: Power Consumption.....	329
Table 321: Functional Validation.....	330
Table 322: Timing Validation.....	330
Table 323: 32-bit Register Write.....	331
Table 324: 32-bit Register Read with Wait State —.....	331
Table 325: APB Programming Interface Specification for STREAM Revision History.....	331
Table 326: Interface Parameters.....	332
Table 327: Assumption 1: Hierarchical Event Organization.....	333
Table 328: Assumption 2: Protocol Isolation.....	333
Table 329: Assumption 3: Two-Tier Memory Architecture.....	334
Table 330: Assumption 4: Configurable Packet Routing.....	334
Table 331: Monitor Bus Output Interface.....	334
Table 332: Protocol Input Interfaces.....	335
Table 333: Control and Status Signals.....	336
Table 334: 64-bit Monitor Bus Packet Structure.....	336
Table 335: Packet Type Definitions.....	337
Table 336: Error Events.....	338
Table 337: Timeout Events.....	339
Table 338: Performance Events.....	339
Table 339: Error Events.....	340

Table 340: Timeout Events.....	341
Table 341: Completion Events.....	341
Table 342: Threshold Events.....	342
Table 343: Performance Events.....	342
Table 344: Debug Events.....	343
Table 345: Error Events.....	344
Table 346: Credit Events.....	344
Table 347: Channel Events.....	345
Table 348: Stream Events.....	346
Table 349: Error Events.....	346
Table 350: Timeout Events.....	347
Table 351: Completion Events.....	348
Table 352: Threshold Events.....	349
Table 353: Performance Events.....	349
Table 354: Debug Events.....	350
Table 355: Error Events.....	351
Table 356: Timeout Events.....	352
Table 357: Completion Events.....	353
Table 358: Threshold Events.....	353
Table 359: Performance Events.....	354
Table 360: Debug Events.....	355
Table 361: Local Error/Interrupt Memory.....	356
Table 362: Configurable External Memory.....	356
Table 363: Base and Limit Registers.....	357
Table 364: Routing Decision Logic.....	357
Table 365: Memory Layout Example.....	357
Table 366: Transaction State Enumeration.....	358
Table 367: Enhanced Transaction Structure.....	359
Table 368: Phase Completion Flags.....	359
Table 369: Protocol-Specific Tracking Fields.....	360
Table 370: APB Transaction Phases.....	360
Table 371: APB Protection Types.....	361
Table 372: MNOC Payload Types.....	361
Table 373: MNOC ACK Types.....	361
Table 374: ARB State Types.....	362
Table 375: CORE State Types.....	362
Table 376: Global Configuration.....	363
Table 377: Packet Type Enable Mapping.....	363

Table 378: AXI Monitor Configuration.....	364
Table 379: MNOC Monitor Configuration.....	365
Table 380: ARB Monitor Configuration.....	365
Table 381: CORE Monitor Configuration.....	366
Table 382: Functional Validation.....	366
Table 383: Performance Validation.....	367
Table 384: Error Handling Validation.....	367
Table 385: Creating Monitor Packets.....	367
Table 386: Packet Decoding.....	368
Table 387: Packet Field Extraction.....	368
Table 388: Packet Creation Function.....	369
Table 389: Event Code Creation Functions.....	369
Table 390: Validation Functions.....	371
Table 391: String Functions for Debugging.....	371
Table 392: Essential Debug Signals.....	371
Table 393: Performance Counters.....	372
Table 394: Complete Protocol Event Matrix.....	372
Table 395: Monitor Bus Architecture and Event Code Organization Revision History.....	373
Table 396: Channel Kick-off Registers.....	374
Table 397: GLOBAL_CTRL.....	376
Table 398: GLOBAL_STATUS.....	376
Table 399: VERSION.....	377
Table 400: CHANNEL_ENABLE.....	377
Table 401: CHANNEL_RESET.....	378
Table 402: CHANNEL_IDLE.....	378
Table 403: DESC_ENGINE_IDLE.....	378
Table 404: SCHEDULER_IDLE.....	378
Table 405: CH_STATE[0..7].....	379
Table 406: SCHED_ERROR.....	380
Table 407: AXI_RD_COMPLETE.....	380
Table 408: AXI_WR_COMPLETE.....	380
Table 409: MON_FIFO_STATUS.....	380
Table 410: MON_FIFO_COUNT.....	381
Table 411: SCHED_TIMEOUT_CYCLES.....	381
Table 412: SCHED_CONFIG.....	381
Table 413: DESCENG_CONFIG.....	382
Table 414: DESCENG_ADDR0_BASE.....	382

Table 415: DESCENG_ADDR0_LIMIT.....	382
Table 416: DESCENG_ADDR1_BASE.....	383
Table 417: DESCENG_ADDR1_LIMIT.....	383
Table 418: DAXMON_ENABLE.....	383
Table 419: DAXMON_TIMEOUT.....	384
Table 420: DAXMON_LATENCY_THRESH.....	384
Table 421: DAXMON_PKT_MASK.....	384
Table 422: DAXMON_ERR_CFG.....	384
Table 423: DAXMON_MASK1.....	385
Table 424: DAXMON_MASK2.....	385
Table 425: DAXMON_MASK3.....	385
Table 426: RDMON_ENABLE.....	386
Table 427: RDMON_TIMEOUT.....	386
Table 428: RDMON_LATENCY_THRESH.....	386
Table 429: RDMON_PKT_MASK.....	386
Table 430: RDMON_ERR_CFG.....	387
Table 431: RDMON_MASK1.....	387
Table 432: RDMON_MASK2.....	387
Table 433: RDMON_MASK3.....	388
Table 434: WRMON_ENABLE.....	388
Table 435: WRMON_TIMEOUT.....	388
Table 436: WRMON_LATENCY_THRESH.....	389
Table 437: WRMON_PKT_MASK.....	389
Table 438: WRMON_ERR_CFG.....	389
Table 439: WRMON_MASK1.....	389
Table 440: WRMON_MASK2.....	390
Table 441: WRMON_MASK3.....	390
Table 442: AXI_XFER_CONFIG.....	390
Table 443: PERF_CONFIG.....	391
Table 444: Register Summary Table.....	393
Table 445: PeakRDL Generation.....	394
Table 446: Configuration Categories.....	396
Table 447: 9. Configuration Register Map Summary.....	412

List of Waveforms

Waveform 2.1.1: STREAM Core Data Flow.....	118
Waveform 2.5.1: Descriptor Engine APB Basic Kick-off.....	173
Waveform 2.6.1: AXI Read Engine - Perfect Streaming.....	182
Waveform 2.6.2: Datapath Read - Multi-Channel.....	184
Waveform 2.8.1: SRAM Controller Write/Read Operation.....	211
Waveform 2.10.1: Stream Latency Bridge Backpressure.....	228
Waveform 2.12.1: AXI Write Engine - Perfect Streaming.....	244
Waveform 2.12.2: Datapath Write - Multi-Channel.....	246
Waveform 2.13.1: APB Normal Write Sequence.....	254
Waveform 2.13.2: APB Write with Backpressure.....	255
Waveform 2.13.3: Channel 0 Kick-off.....	256
Waveform 2.13.4: Channel 7 Kick-off.....	257
Waveform 2.13.5: Multi-Channel Kick-off Sequence.....	258

1 Stream Index

Generated: 2026-01-03

2 Document Information

This document describes the STREAM subsystem, a Scatter-gather Transfer Rapid Engine for AXI Memory. STREAM provides descriptor-based DMA functionality with multi-channel support, enabling efficient memory-to-memory data transfers via AXI4 interfaces. The subsystem is designed as a tutorial-focused implementation with intentional simplifications for educational purposes.

2.1 References

2.1.1 Related Documents

Related Documents and Specifications

Source	Title	Version
RTL Design Sherpa	STREAM Product Requirements Document	1.0
RTL Design Sherpa	RAPIDS MAS (Parent Architecture)	0.90
ARM	AMBA AXI and ACE Protocol Specification	IHI0022H
ARM	AMBA APB Protocol Specification	IHI0024C

2.2 Terminology

AXI

Advanced eXtensible Interface. ARM's high-performance, high-frequency protocol for interconnect.

AXI4

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

AXIL

AXI4-Lite. A simplified subset of AXI4 for low-throughput register access.

APB

Advanced Peripheral Bus. ARM's simple, low-power bus protocol for peripheral register access.

Beat

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

Burst

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

Channel

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

Descriptor

A 256-bit data structure containing transfer parameters: source address, destination address, length, and chain pointer.

Descriptor Chain

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

DMA

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

FUB

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

MAC

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

MonBus

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

Outstanding Transaction

A transaction that has been issued but not yet completed. V2/V3 performance modes support multiple outstanding transactions.

Scatter-Gather

A DMA technique where non-contiguous memory regions are transferred using a descriptor chain.

SRAM

Static Random Access Memory. Used for internal data buffering between read and write datapaths.

V1/V2/V3

STREAM performance modes: - V1 (Low): Single outstanding transaction, tutorial-focused - V2 (Medium): Command pipelined, best area efficiency - V3 (High): Out-of-order completion, maximum throughput

2.3 Revision History

Revisions follow the convention x.y, where x is the major version and y is the minor version. Minor versions indicate incremental updates; major versions indicate significant architectural changes.

STREAM MAS Document Revision History

Rev	Date	Author	Notes
0.90	2025-11-22	seang	Initial STREAM MAS structure and block documentation
0.91	2026-01-02	seang	Added front matter, figure/table captions, document styling

Last Updated: 2026-01-02

3 STREAM Architecture Overview

Component: STREAM (Scatter-gather Transfer Rapid Engine for AXI Memory) **Version:** 0.90
Status: Pre-release

3.1 Introduction

STREAM is a high-performance, multi-channel descriptor-based DMA engine designed for memory-to-memory scatter-gather transfers. It provides an educational yet production-capable architecture demonstrating key DMA concepts while maintaining professional design practices.

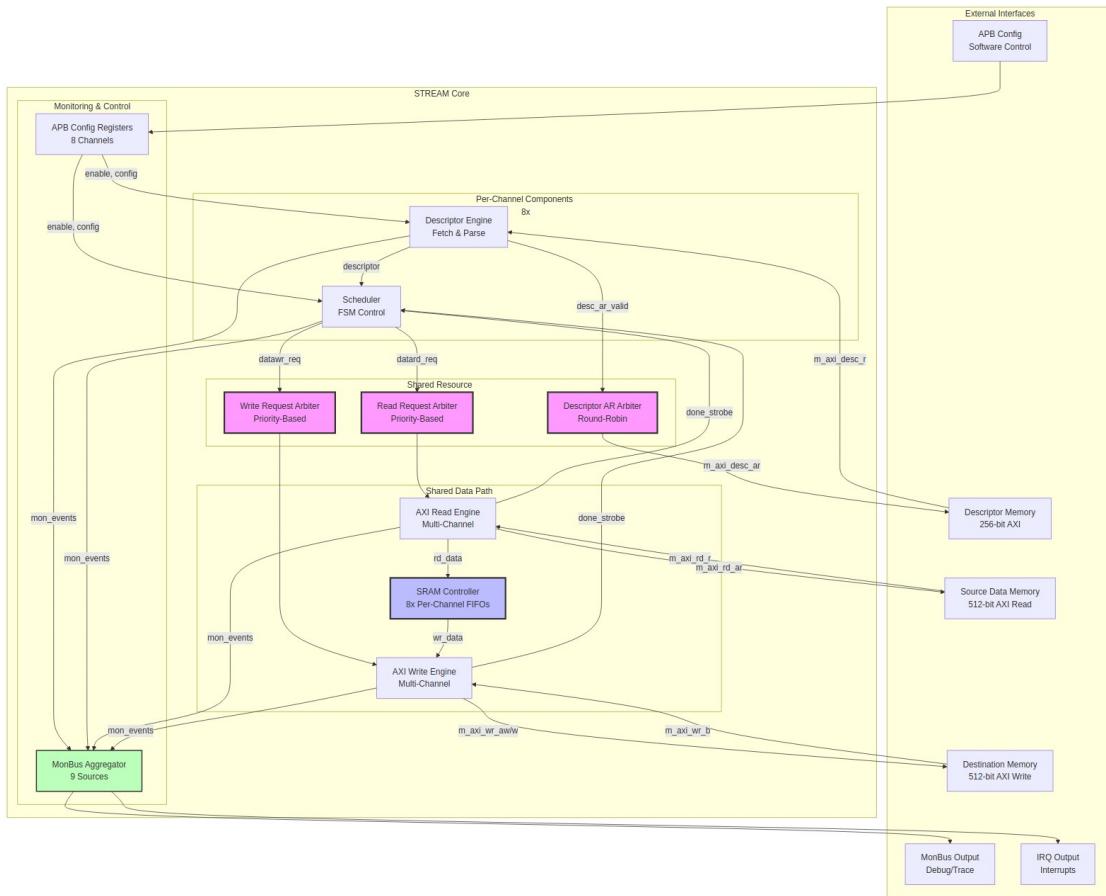
3.1.1 Design Philosophy

1. **Tutorial Focus** - Intentional simplifications for learning
 2. **Production Quality** - Industry-standard interfaces and verification
 3. **Scalability** - 8 independent channels with shared resource arbitration
 4. **Modularity** - Clear separation of concerns across functional blocks
-

3.2 System Architecture

3.2.1 High-Level Block Diagram

3.2.2 Figure 1.1.1: STREAM Architecture Overview



Diagram

3.3 Key Architectural Concepts

3.3.1 1. Multi-Channel Design

8 Independent Channels: - Each channel operates autonomously - Separate descriptor chains per channel - Independent FSM state per channel - Concurrent transfers across all channels

Resource Sharing: - Descriptor AXI master (shared via round-robin arbiter) - Data read AXI master (shared via priority arbiter) - Data write AXI master (shared via priority arbiter) - SRAM buffer (per-channel FIFOs, no arbitration)

3.3.2 2. Descriptor-Based Operation

Descriptor Format (256-bit):

[255:192] Reserved
[191:160] Next Descriptor Pointer
[159:128] Length (in beats)
[127:64] Destination Address
[63:0] Source Address

Descriptor Chain: - Single APB write kicks off chain - Automatic chaining via next_descriptor_ptr
- Explicit termination (ptr = 0 or last flag)

3.3.3 3. Concurrent Read/Write

CRITICAL Design Pattern:

STREAM uses concurrent read and write operations to handle transfers larger than the SRAM buffer:

Example: 100MB transfer with 64KB SRAM buffer

Sequential (WRONG) :

Read 100MB → DEADLOCK at 64KB (SRAM full)

Concurrent (CORRECT) :

Read fills SRAM → SRAM full → Read pauses
Write drains SRAM → SRAM space freed → Read resumes
Both continue until 100MB complete

Implementation: - Scheduler runs read and write FSMs concurrently in XFER_DATA state -
Independent beat counters for read vs write - Transfer completes when BOTH counters reach zero

3.3.4 4. Space-Aware Flow Control

Allocation Controller (Read Path): - Reserves SRAM space BEFORE issuing AXI AR - Prevents overflow due to AR/R latency gap - 2x space margin (accounts for in-flight allocations)

Drain Controller (Write Path): - Reserves SRAM data BEFORE issuing AXI AW - Prevents underflow due to AW/W latency gap - Includes latency bridge occupancy in count

3.3.5 5. Priority-Based Arbitration

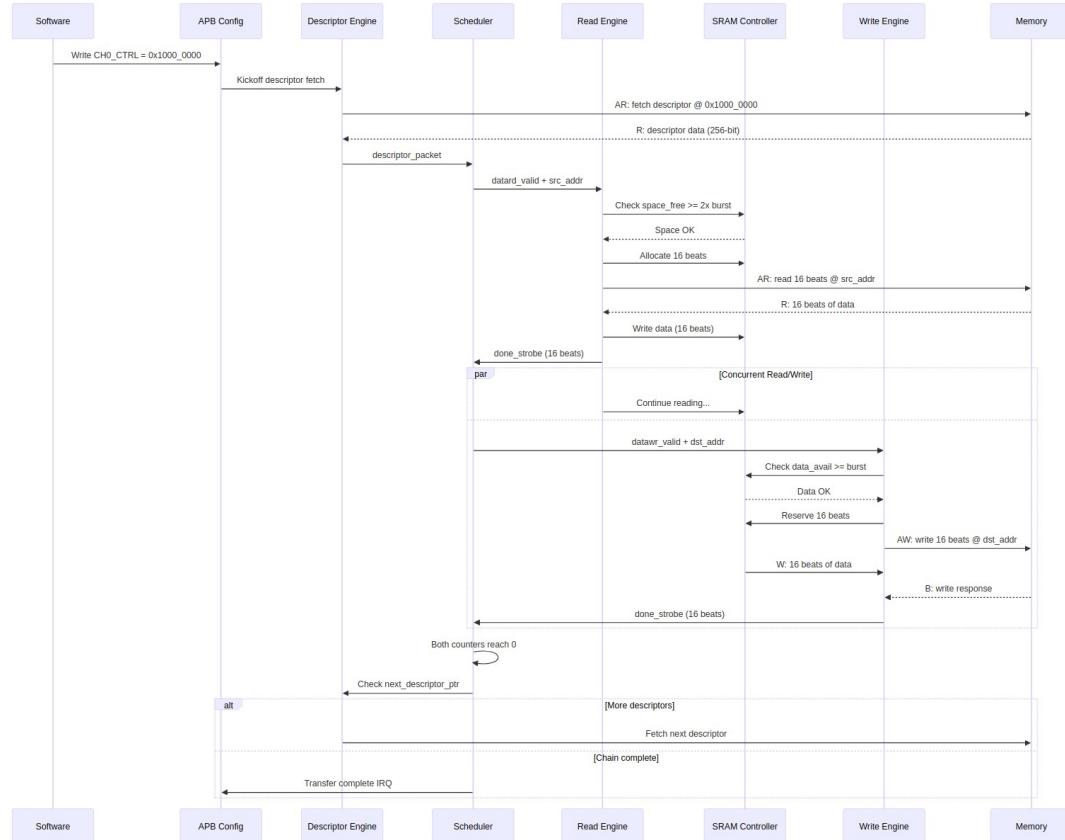
Descriptor Fetch: - Round-robin arbitration (fair access) - All channels equal priority

Data Read/Write: - Priority-based arbitration - Priority from descriptor - Round-robin within same priority tier - Timeout prevention for low-priority channels

3.4 Data Flow Example

3.4.1 Single Channel Transfer

3.4.2 Figure 1.1.2: Single Channel Transfer Sequence

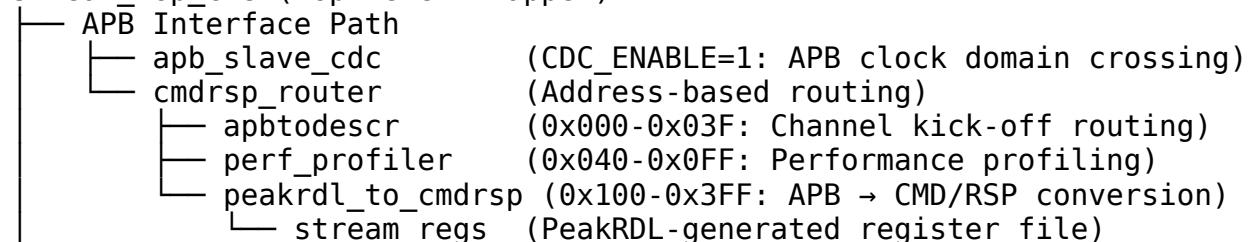


Diagram

3.5 Component Hierarchy

3.5.1 Top-Level Integration (stream_top_ch8.sv)

stream_top_ch8 (Top-Level Wrapper)

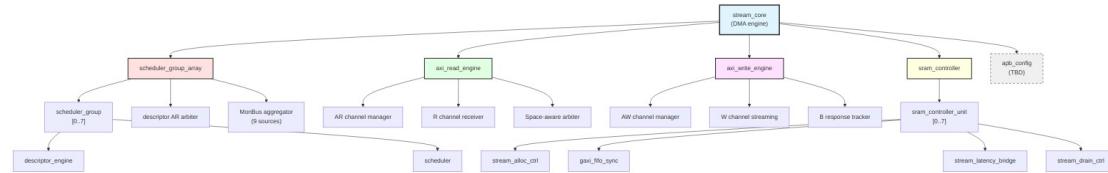


```

└── stream_config_block      (Register → config signal mapping)
└── stream_core              (Main DMA engine - see below)
└── monbus_axil_group        (USE_AXI_MONITORS=1: MonBus → AXI-Lite)
    └── Error FIFO (s_axil_err_* slave read)
    └── Master Writer (m_axil_mon_* master write)

```

3.5.2 Core DMA Engine (stream_core.sv)



1.3: Core DMA Engine Hierarchy

3.6 Interface Summary

3.6.1 External Interfaces (stream_top_ch8.sv)

STREAM External Interfaces

Interface	Type	Width	Purpose
APB	Slave	32-bit	Configuration, control, status (with optional CDC)
Descriptor AXI	Master	256-bit	Fetch descriptors from memory
Data Read AXI	Master	512-bit	Read source data
Data Write AXI	Master	512-bit	Write destination data
Error FIFO AXIL	Slave	32-bit	Read monitor error/interrupt FIFO
Monitor Write AXIL	Master	32-bit	Write monitor data to memory
IRQ	Output	1-bit	Interrupt (error FIFO not empty)

3.6.2 Internal MonBus Interface (stream_core.sv)

STREAM Internal MonBus Interface

Interface	Type	Width	Purpose
MonBus	Output	64-bit	Debug/trace event stream to monbus_axil_group

3.6.3 Configuration Registers (APB)

APB Address Map:

0x000-0x03F: Channel kick-off registers (apbtodescr routing)

0x040-0x0FF: Performance profiler interface

0x100-0x3FF: PeakRDL configuration registers

Key Register Groups: - 0x100: Global Control (enable, reset, version) - 0x120: Per-Channel Control (enable, reset) - 0x140: Per-Channel Status (idle, state, completion) - 0x180: Monitor FIFO Status - 0x200: Scheduler Configuration - 0x220: Descriptor Engine Configuration - 0x240-0x29F: AXI Monitor Configuration (DAXMON, RDMON, WRMON) - 0x2A0: AXI Transfer Configuration - 0x2B0: Performance Profiler Configuration

See: [Register Map](#) for complete documentation.

3.7 Resource Utilization

3.7.1 Area Breakdown (Typical 512-bit Data Width)

STREAM Resource Utilization Breakdown

Component	Quantity	Logic (LUTs)	Memory
Descriptor Engine	8	~300 each	Minimal
Scheduler	8	~400 each	Minimal
AXI Read Engine	1	~800	Minimal
AXI Write Engine	1	~800	Minimal
SRAM Controller	1	~600	$8 \times 64\text{KB} = 512\text{KB}$

Component	Quantity	Logic (LUTs)	Memory
Arbiters	3	~150 each	Minimal
APB Config	1	~400	Register file
MonBus	1	~200	Small FIFO
Aggregator			
Total	-	~10K LUTs	~512KB RAM

Notes: - SRAM depth configurable (typical: 512 entries \times 512 bits \times 8 channels = 512KB) - Logic utilization scales with NUM_CHANNELS - Memory utilization scales with SRAM_DEPTH

3.8 Performance Characteristics

3.8.1 Throughput

Theoretical Maximum (512-bit @ 250MHz): - Single channel: 16 GB/s (512 bits \times 250 MHz) - 8 channels concurrent: Limited by memory bandwidth, not DMA

Practical Performance: - Dependent on: - Memory controller latency - Burst sizes (larger = better efficiency) - Transfer alignment - Channel contention

Typical Real-World: - 8-12 GB/s sustained (single channel) - 50-75% of theoretical maximum

3.8.2 Latency

Descriptor Fetch Latency: - AR issue to R data: ~10-50 cycles (memory dependent) - Descriptor parsing: 1 cycle - **Total:** ~11-51 cycles per descriptor

Transfer Initiation: - APB write to first AR: ~5-10 cycles - AR to first R data: ~10-50 cycles - **Total:** ~15-60 cycles from kickoff to first data

Transfer Completion: - Last W beat to B response: ~5-20 cycles - B response to IRQ: 1 cycle - **Total:** ~6-21 cycles from last data to interrupt

3.9 Design Decisions

3.9.1 Why These Simplifications?

1. Aligned Addresses Only:

- Eliminates complex alignment fixup logic
- Clear data path with no byte shifting

- Focus: Core DMA operation, not edge cases
2. **Length in Beats:**
 - Direct mapping to AXI burst length
 - No unit conversion overhead
 - Focus: AXI protocol understanding
 3. **No Credit Management:**
 - Simpler resource arbitration
 - Transaction limits via configuration
 - Focus: Arbitration basics, not complex flow control
 4. **No Circular Buffers:**
 - Explicit chain termination
 - Clear end-of-transfer detection
 - Focus: Descriptor chaining, not circular logic

3.9.2 Production Enhancements (Future)

If extending STREAM for production use, consider: - [] Alignment fixup (byte-level granularity) - [] Length in bytes/chunks - [] Credit-based flow control - [] Circular buffer support - [] Advanced error recovery - [] Power management - [] Multiple SRAM segments

3.10 Testing Strategy

3.10.1 Verification Layers

- 1. Unit Tests (FUB Level):** - Descriptor engine: Fetch, parse, chain - Scheduler: FSM states, concurrent read/write - AXI engines: AR/R and AW/W/B channels - SRAM controller: Allocation, drain, FIFO - Individual tests per module
- 2. Integration Tests (Multi-Block):** - Scheduler + engines: Data flow - Descriptor + scheduler: Descriptor chaining - Full path: APB → Descriptor → Transfer → IRQ
- 3. System Tests (Full Core):** - Single channel end-to-end - Multi-channel concurrent - Error injection and recovery - Performance validation - Long-duration stress tests

3.10.2 Coverage Targets

- **Code Coverage:** >95%
- **Functional Coverage:** >90%
- **Corner Cases:** 100% tested
- **Error Paths:** 100% tested

3.11 Related Documentation

Chapter 2 - Block Specifications: - [Scheduler](#) - Core FSM controller - [Descriptor Engine](#) - Descriptor fetch/parse - [AXI Read Engine](#) - Source data read - [AXI Write Engine](#) - Destination data write - [SRAM Controller](#) - Buffering and flow control - [Scheduler Group](#) - Per-channel wrapper - [Scheduler Group Array](#) - 8-channel array

Other Resources: - [STREAM PRD](#) - Product requirements - [Test Plan](#) - Verification strategy

Last Updated: 2025-12-01 **Document Version:** 0.91 —

3.12 Revision History

STREAM Architecture Overview Revision History

Version	Date	Author	Description
0.90	2025-11-22	seang	Initial block specification
0.91	2026-01-02	seang	Added table captions and figure numbers

Last Updated: 2026-01-02

4 STREAM Top-Level Port List

Module: stream_core.sv **Location:** projects/components/stream/rtl/macro/ **Last Updated:** 2025-12-01

4.1 Overview

This document provides a complete reference for all top-level ports of the STREAM Core module, organized by interface type. All port names, directions, widths, and descriptions are extracted directly from the RTL implementation.

Note: This documents `stream_core.sv` which is instantiated inside `stream_top_ch8.sv`. The top-level wrapper adds:

- APB slave interface (replaces `apb_valid/ready/addr` with standard APB protocol)
- AXI-Lite interfaces for monitor bus (`s_axil_err_`, `m_axil_mon_`)
- Optional APB CDC crossing (`CDC_ENABLE` parameter)

For top-level integration, see the [Architecture Overview](#).

Quick Navigation: - [Clock and Reset](#) - [APB Programming Interface](#) - [Configuration Interface](#) - [Status Interface](#) - [Performance Profiler Interface](#) - [AXI4 Master](#) - [Descriptor Fetch](#) - [AXI4 Master](#) - [Data Read](#) - [AXI4 Master](#) - [Data Write](#) - [Status/Debug Outputs](#) - [Unified Monitor Bus](#)

4.2 Clock and Reset

Clock and Reset Signals

Signal	Direction	Width	Description
<code>clk</code>	input	1	System clock (100-500 MHz typical)
<code>rst_n</code>	input	1	Active-low asynchronous reset

Notes: - All STREAM logic operates in the `clk` domain - Reset is asynchronous assert, synchronous deassert - All registers use reset macros from `reset_defs.svh`

4.3 APB Programming Interface

Per-channel descriptor kick-off interface. Each channel has independent valid/ready/address signals for descriptor chain start.

APB Programming Interface Signals

Signal	Direction	Width	Description
<code>apb_valid[ch]</code>	input	NUM_CHAN NELS	Channel descriptor address valid
<code>apb_ready[ch]</code>	output	NUM_CHAN NELS	Channel ready to accept descriptor address

Signal	Direction	Width	Description
apb_addr[ch]	input	NUM_CHAN NELS × ADDR_WIDTH H	Descriptor address per channel (64-bit default)

Usage Pattern:

```
// Kick off channel 0 with descriptor at 0x1000_0000
apb_valid[0] = 1'b1;
apb_addr[0] = 64'h0000_0000_1000_0000;
// Wait for handshake
wait (apb_ready[0]);
apb_valid[0] = 1'b0;
```

Notes: - Standard valid/ready handshake protocol - Descriptor address must be aligned to descriptor size (32 bytes) - Channel starts operation immediately after handshake - Default: NUM_CHANNELS = 8, ADDR_WIDTH = 64

4.4 Configuration Interface

4.4.1 Per-Channel Configuration

Per-Channel Configuration Signals

Signal	Direction	Width	Description
cfg_channel_enable[ch]	input	NUM_CHAN NELS	Enable channel (0=disabled, 1=enabled)
cfg_channel_reset[ch]	input	NUM_CHAN NELS	Soft reset channel (FSM → IDLE state)

Notes: - Channel must be enabled before accepting descriptors - Soft reset clears channel FSM but preserves config

4.4.2 Global Scheduler Configuration

Global Scheduler Configuration Signals

Signal	Direction	Width	Description
cfg_sched_enable	input	1	Global scheduler enable
cfg_sched_timeout_cycles	input	16	Timeout threshold (clock cycles)

Signal	Direction	Width	Description
cfg_sched_ti meout_enable	input	1	Enable timeout detection
cfg_sched_er r_enable	input	1	Enable error event reporting
cfg_sched_co mpl_enable	input	1	Enable completion event reporting
cfg_sched_pe rf_enable	input	1	Enable performance event reporting

Notes: - cfg_sched_enable is master enable for all schedulers - Timeout measured from descriptor fetch to completion - Event enables control MonBus traffic

4.4.3 Descriptor Engine Configuration

Descriptor Engine Configuration Signals

Signal	Direction	Width	Description
cfg_desceng_ enable	input	1	Enable descriptor engine
cfg_desceng_ prefetch	input	1	Enable descriptor prefetch
cfg_desceng_ fifo_thresh	input	4	FIFO threshold for prefetch trigger
cfg_desceng_ addr0_base	input	ADDR_WIDT H	Address range 0 base (protection)
cfg_desceng_ addr0_limit	input	ADDR_WIDT H	Address range 0 limit (protection)
cfg_desceng_ addr1_base	input	ADDR_WIDT H	Address range 1 base (protection)
cfg_desceng_ addr1_limit	input	ADDR_WIDT H	Address range 1 limit (protection)

Notes: - Descriptor engine shared across all channels - Prefetch improves latency for chained descriptors - Address ranges provide optional descriptor memory protection

4.4.4 AXI Monitor Configuration

Three identical sets of monitor config signals for descriptor, read, and write AXI masters:

AXI Monitor Signal Prefixes

Signal Prefix	Applies To	Description
cfg_desc_mon_*	Descriptor AXI	Descriptor fetch monitoring
cfg_rdeng_mon_*	Read AXI	Data read monitoring
cfg_wreng_mon_*	Write AXI	Data write monitoring

Each monitor has the following configuration signals:

AXI Monitor Configuration Signal Suffixes

Signal Suffix	Width	Description
_enable	1	Enable monitor
_err_enable	1	Enable error packet reporting
_perf_enable	1	Enable performance packet reporting
_timeout_enable	1	Enable timeout detection
_timeout_cycles	32	Timeout threshold (cycles)
_latency_thresh	32	Latency threshold for events
_pkt_mask	16	Packet type mask (filter events)
_err_select	4	Error type selector
_err_mask	8	Error event mask
_timeout_mask	8	Timeout event mask
_compl_mask	8	Completion event mask
_thresh_mask	8	Threshold event mask
_perf_mask	8	Performance event

Signal Suffix	Width	Description
		mask
_addr_mask	8	Address event mask
_debug_mask	8	Debug event mask

Example Signals:

```
cfg_desc_mon_enable      // Descriptor monitor enable
cfg_desc_mon_timeout_cycles // Descriptor timeout threshold
cfg_rdeng_mon_err_enable // Read engine error reporting enable
cfg_wreng_mon_perf_enable // Write engine perf reporting enable
```

Notes: - Monitors generate MonBus packets for events - Masks control which event types are reported - Timeout measured from AR/AW valid to last R/B response

4.4.5 AXI Transfer Configuration

AXI Transfer Configuration Signals

Signal	Direction	Width	Description
cfg_axi_rd_xfer_beats	input	8	Read burst size (beats, 1-256)
cfg_axi_wr_xfer_beats	input	8	Write burst size (beats, 1-256)

Notes: - Configures AXI burst length for read/write engines - Larger bursts improve bandwidth but increase latency - Typical: 16 beats for DDR4, 8 beats for low-latency SRAM

4.4.6 Performance Profiler Configuration

Performance Profiler Configuration Signals

Signal	Direction	Width	Description
cfg_perf_enable	input	1	Enable profiler
cfg_perf_mode	input	1	Profiler mode (0=timestamp, 1=elapsed)
cfg_perf_clear	input	1	Clear profiler counters and FIFO

Notes: - Timestamp mode: Records start/end timestamps (software calculates elapsed) - Elapsed mode: Hardware calculates elapsed time directly - Clear is a single-cycle pulse

4.5 Status Interface

4.5.1 Per-Channel Status

Per-Channel Status Signals

Signal	Direction	Width	Description
descriptor_engine_idle[c]	output	NUM_CHAN NELS	Descriptor engine idle (no pending fetch)
scheduler_id	output	NUM_CHAN NELS	Scheduler in IDLE state (waiting for descriptor)
scheduler_state[ch]	output	NUM_CHAN NELS × 7	Scheduler FSM state (ONE-HOT encoding)
sched_error[ch]	output	NUM_CHAN NELS	Scheduler error flag (sticky, cleared by reset)
axi_rd_all_complete[ch]	output	NUM_CHAN NELS	All read transactions complete for channel
axi_wr_all_complete[ch]	output	NUM_CHAN NELS	All write transactions complete for channel

Scheduler State Encoding (ONE-HOT):

```
7'b0000001 // CH_IDLE - Waiting for descriptor
7'b0000010 // CH_FETCH_DESC - Fetching descriptor
7'b0000100 // CH_XFER_DATA - Transfer in progress
7'b0001000 // CH_COMPLETE - Transfer complete
7'b0010000 // CH_NEXT_DESC - Chaining to next descriptor
7'b0100000 // CH_ERROR - Error occurred
7'b1000000 // (Reserved)
```

Transfer Complete Condition:

```
channel_complete = scheduler_idle[ch] &&
                  axi_rd_all_complete[ch] &&
                  axi_wr_all_complete[ch];
```

4.6 Performance Profiler Interface

Performance Profiler Interface Signals

Signal	Direction	Width	Description
perf_fifo_em pty	output	1	Profiler FIFO empty flag
perf_fifo_fu ll	output	1	Profiler FIFO full flag
perf_fifo_co unt	output	16	Profiler FIFO occupancy (0-256)
perf_fifo_rd	input	1	Read profiler entry (pop FIFO)
perf_fifo_da ta_low	output	32	Profiler data [31:0] (timestamp or elapsed)
perf_fifo_da ta_high	output	32	Profiler data [63:32] (metadata)

FIFO Entry Format:

Low Word [31:0]: - Timestamp mode: Timestamp in clock cycles - Elapsed mode: Elapsed time in clock cycles

High Word [31:0]: - Bits [31:4]: Reserved (0) - Bit [3]: Event type (0=start, 1=end) - Bits [2:0]: Channel ID (0-7)

Read Sequence:

```
// Check FIFO not empty
if (!perf_fifo_empty) begin
    // Read 36-bit entry (two registers)
    perf_fifo_rd = 1'b1; // Pulse to pop FIFO
    @(posedge clk);
    perf_fifo_rd = 1'b0;

    // Sample data on next cycle
    timestamp = perf_fifo_data_low;
    metadata = perf_fifo_data_high;
    channel_id = metadata[2:0];
    event_type = metadata[3];
end
```

4.7 AXI4 Master - Descriptor Fetch (256-bit)

Fixed 256-bit width AXI4 master for descriptor fetch from memory.

4.7.1 AR Channel (Read Address)

Descriptor AXI Master AR Channel Signals

Signal	Direction	Width	Description
m_axi_desc_ar_id	output	AXI_ID_WIDTH	Transaction ID
m_axi_desc_ar_addr	output	ADDR_WIDTH	Read address
m_axi_desc_ar_len	output	8	Burst length - 1 (AXI encoding)
m_axi_desc_ar_size	output	3	Burst size = $\log_2(\text{bytes per beat})$
m_axi_desc_ar_burst	output	2	Burst type (01=INCR)
m_axi_desc_ar_lock	output	1	Lock type (0=normal)
m_axi_desc_ar_cache	output	4	Cache attributes
m_axi_desc_ar_prot	output	3	Protection attributes
m_axi_desc_ar_qos	output	4	QoS value
m_axi_desc_ar_region	output	4	Region identifier
m_axi_desc_ar_user	output	CHAN_WIDTH	User signal (channel ID)
m_axi_desc_ar_valid	output	1	Address valid
m_axi_desc_ar_ready	input	1	Address ready

4.7.2 R Channel (Read Data)

R Channel

Signal	Direction	Width	Description
m_axi_desc_r_id	input	AXI_ID_WID TH	Transaction ID
m_axi_desc_r_data	input	256	Read data (FIXED 256-bit descriptor)
m_axi_desc_r_resp	input	2	Response (00=OKAY, 01=EXOKAY, 10=SLVERR, 11=DECERR)
m_axi_desc_r_last	input	1	Last beat of burst
m_axi_desc_r_user	input	CHAN_WIDTH	User signal (channel ID)
m_axi_desc_r_valid	input	1	Read data valid
m_axi_desc_r_ready	output	1	Read data ready

Notes: - Data width is FIXED at 256 bits (descriptor size) - Typical burst: 1 beat (arlen=0) for single descriptor fetch - arsize = 3'b101 (32 bytes = 2^5) - ID encodes channel number for response routing

4.8 AXI4 Master - Data Read (Parameterizable Width)

Parameterizable width AXI4 master for reading source data from memory. Default 512-bit.

4.8.1 AR Channel (Read Address)

AR Channel

Signal	Direction	Width	Description
m_axi_rd_arid	output	AXI_ID_WIDTH	Transaction ID
m_axi_rd_arad_dr	output	ADDR_WIDTH	Read address
m_axi_rd_arlen	output	8	Burst length - 1
m_axi_rd_arsize	output	3	Burst size =

Signal	Direction	Width	Description
			log2(bytes per beat)
m_axi_rd_arbu_rst	output	2	Burst type (01=INCR)
m_axi_rd_arlo_ck	output	1	Lock type (0=normal)
m_axi_rd_arca_cke	output	4	Cache attributes
m_axi_rd_arpr_ot	output	3	Protection attributes
m_axi_rd_arqo_s	output	4	QoS value
m_axi_rd_arre_gion	output	4	Region identifier
m_axi_rd_arus_er	output	CHAN_WIDTH	User signal (channel ID)
m_axi_rd_arva_lid	output	1	Address valid
m_axi_rd_arre_ady	input	1	Address ready

4.8.2 R Channel (Read Data)

R Channel

Signal	Direction	Width	Description
m_axi_rd_rid	input	AXI_ID_WIDTH	Transaction ID
m_axi_rd_rdat_a	input	DATA_WIDTH	Read data (default 512-bit)
m_axi_rd_rres_p	input	2	Response
m_axi_rd_rlas_t	input	1	Last beat of burst
m_axi_rd_ruse_r	input	CHAN_WIDTH	User signal (channel ID)
m_axi_rd_rval	input	1	Read data

Signal	Direction	Width	Description
id			valid
m_axi_rd_rrea dy	output	1	Read data ready

Notes: - Data width configurable via DATA_WIDTH parameter (default 512) - Burst length configured via cfg_axi_rd_xfer_beats (default 16) - arsize = log2(DATA_WIDTH/8) automatically calculated - AXI skid buffers on external interface for timing closure

4.9 AXI4 Master - Data Write (Parameterizable Width)

Parameterizable width AXI4 master for writing destination data to memory. Default 512-bit.

4.9.1 AW Channel (Write Address)

AW Channel

Signal	Direction	Width	Description
m_axi_wr_awid	output	AXI_ID_WIDTH	Transaction ID
m_axi_wr_awad dr	output	ADDR_WIDTH	Write address
m_axi_wr_awle n	output	8	Burst length - 1
m_axi_wr_awsi ze	output	3	Burst size = log2(bytes per beat)
m_axi_wr_awbu rst	output	2	Burst type (01=INCR)
m_axi_wr_awlo ck	output	1	Lock type (0=normal)
m_axi_wr_awca che	output	4	Cache attributes
m_axi_wr_awpr ot	output	3	Protection attributes
m_axi_wr_awqo s	output	4	QoS value
m_axi_wr_awre gion	output	4	Region identifier

Signal	Direction	Width	Description
m_axi_wr_awuser	output	CHAN_WIDTH	User signal (channel ID)
m_axi_wr_awvalid	output	1	Address valid
m_axi_wr_awready	input	1	Address ready

4.9.2 W Channel (Write Data)

W Channel

Signal	Direction	Width	Description
m_axi_wr_wdata	output	DATA_WIDTH	Write data (default 512-bit)
m_axi_wr_wstrb	output	DATA_WIDTH/8	Write strobes (byte enables, all 1s)
m_axi_wr_wlast	output	1	Last beat of burst
m_axi_wr_wuser	output	CHAN_WIDTH	User signal (channel ID)
m_axi_wr_wvalid	output	1	Write data valid
m_axi_wr_wready	input	1	Write data ready

4.9.3 B Channel (Write Response)

B Channel

Signal	Direction	Width	Description
m_axi_wr_bid	input	AXI_ID_WIDTH	Transaction ID
m_axi_wr_bresp	input	2	Response (00=OKAY, 01=EXOKAY, 10=SLVERR, 11=DECERR)
m_axi_wr_buserr	input	CHAN_WIDTH	User signal (channel ID)
m_axi_wr_bvalid	input	1	Response valid
m_axi_wr_bready	output	1	Response ready

Signal	Direction	Width	Description
ady			Notes: - Data width configurable via DATA_WIDTH parameter (default 512) - Burst length configured via cfg_axi_wr_xfer_beats (default 16) - awsize = log2(DATA_WIDTH/8) automatically calculated - wstrb typically all 1s (full data width writes) - AXI skid buffers on external interface for timing closure

4.10 Status/Debug Outputs

4.10.1 Descriptor AXI Monitor Status

Descriptor AXI Monitor Status

Signal	Direction	Width	Description
cfg_sts_desc_mon_busy	output	1	Monitor busy processing transaction
cfg_sts_desc_mon_active_txns	output	8	Active transaction count (0-255)
cfg_sts_desc_mon_error_count	output	16	Cumulative error count
cfg_sts_desc_mon_txn_count	output	32	Total transaction count
cfg_sts_desc_mon_conflict_error	output	1	ID conflict detected (sticky)

4.10.2 Read Engine AXI Monitor Status

Read Engine AXI Monitor Status

Signal	Direction	Width	Description
cfg_sts_rden_g_skid_busy	output	1	Skid buffer busy (not empty)
cfg_sts_rden_g_mon_active_txns	output	8	Active transaction count
cfg_sts_rden_g_mon_error_count	output	16	Cumulative error count

Signal	Direction	Width	Description
cfg_sts_rden_g_mon_txn_count	output	32	Total transaction count
cfg_sts_rden_g_mon_conflict_error	output	1	ID conflict detected (sticky)

4.10.3 Write Engine AXI Monitor Status

Write Engine AXI Monitor Status

Signal	Direction	Width	Description
cfg_sts_wren_g_skid_busy	output	1	Skid buffer busy (not empty)
cfg_sts_wren_g_mon_active_txns	output	8	Active transaction count
cfg_sts_wren_g_mon_error_count	output	16	Cumulative error count
cfg_sts_wren_g_mon_txn_count	output	32	Total transaction count
cfg_sts_wren_g_mon_conflict_error	output	1	ID conflict detected (sticky)

Notes: - Status signals for debug and performance analysis
 - Transaction counts never roll over (use for profiling)
 - Error counts increment on any AXI error response
 - ID conflicts indicate internal RTL bug (should never occur)

4.11 Unified Monitor Bus Interface

Single output interface for all STREAM monitoring events.

Unified Monitor Bus Interface

Signal	Direction	Width	Description
mon_valid	output	1	Monitor packet valid
mon_ready	input	1	Monitor packet ready (from downstream)

Signal	Direction	Width	Description
mon_packet	output	64	Monitor packet data (64-bit standard format) consumer)

MonBus Packet Format (64-bit):

[63:56] - Packet type (event code)
[55:48] - Agent ID (source identifier)
[47:40] - Unit ID (1 for STREAM)
[39:32] - Channel ID (0-7)
[31:0] - Event-specific data

MonBus Sources: - Descriptor engines: 8 sources, agent IDs 16-23 (0x10-0x17) - Schedulers: 8 sources, agent IDs 48-55 (0x30-0x37) - Descriptor AXI monitor: agent ID 8 (0x08) - Read AXI monitor: configurable agent ID - Write AXI monitor: configurable agent ID

Downstream Integration:

```
// Connect to MonBus FIFO for buffering
gaxi_fifo_sync #(
    .DATA_WIDTH(64),
    .DEPTH(256)
) u_mon_fifo (
    .i_clk      (clk),
    .i_rst_n   (rst_n),
    .i_data     (mon_packet),
    .i_valid    (mon_valid),
    .o_ready    (mon_ready),
    // ... downstream connection
);
```

Notes: - Standard AMBA monitor bus protocol - Always buffer with FIFO to prevent backpressure to STREAM - Packet format documented in `rtl/amba/includes/monitor_pkg.sv` - Event codes defined in STREAM package

4.12 Port Count Summary

Port Count Summary

Interface Type	Input Ports	Output Ports	Bidirectional
Clock/Reset	2	0	0
APB	NUM_CHANNELS	NUM_CHANNELS	0
Programming	LS × (1 +	LS	

Interface Type	Input Ports	Output Ports	Bidirectional
ADDR_WIDTH)			
Configuration	~150	0	0
Status	1 (perf_fifo_rd)	NUM_CHANNELS LS × 10 + 30	0
Perf Profiler	1	4	0
AXI Desc Master	7	14	0
AXI Read Master	7	14	0
AXI Write Master	10	19	0
MonBus	1	2	0

Approximate Total: ~350 ports (varies with NUM_CHANNELS and ADDR_WIDTH/DATA_WIDTH)

4.13 Default Parameter Values

Default Parameter Values

Parameter	Default	Description
NUM_CHANNELS	8	Number of DMA channels
CHAN_WIDTH	3	Channel ID width (log2(NUM_CHANNELS))
ADDR_WIDTH	64	Address bus width
DATA_WIDTH	512	Data bus width
AXI_ID_WIDTH	8	AXI transaction ID width
FIFO_DEPTH	512	Per-channel FIFO depth
AR_MAX_OUTSTANDING	8	Max concurrent read requests
AW_MAX_OUTSTANDING	8	Max concurrent

Parameter	Default	Description
		write requests

4.14 Related Documentation

- [Stream Core Block Spec](#) - Detailed block documentation
- [Clocks and Reset](#) - Timing specifications
- [Architecture](#) - System architecture overview

Last Updated: 2025-12-01 Maintained By: STREAM Architecture Team —

4.15 Revision History

STREAM Top-Level Port List Revision History

Version	Date	Author	Description
0.90	2025-11-22	seang	Initial block specification
0.91	2026-01-02	seang	Added table captions and figure numbers

Last Updated: 2026-01-02

5 Clocks and Reset Specification

Chapter: 01 Version: 0.90 Last Updated: 2025-11-22

5.1 Overview

STREAM operates in a single clock domain with a single asynchronous active-low reset. This chapter defines clock requirements, reset behavior, and timing constraints for the STREAM subsystem.

5.2 Clock Domain

5.2.1 Primary Clock: aclk

Specification: - **Name:** aclk (AXI clock) - **Type:** Synchronous, single-edge (rising edge) - **Frequency:** Configurable (typical: 100 MHz - 500 MHz) - **Duty Cycle:** 50% 5% - **Jitter:** < 100 ps peak-to-peak

Usage: - All STREAM internal logic - All AXI master interfaces - All AXIL interfaces - MonBus output - SRAM

5.2.2 Secondary Clock: pclk (APB Clock)

Specification: - **Name:** pclk (Peripheral clock) - **Type:** Synchronous, single-edge (rising edge) - **Frequency:** Configurable (typical: 50 MHz - 200 MHz) - **Relation to aclk:** May be asynchronous

Usage: - APB configuration interface only

Clock Domain Crossing (CDC): - If pclk aclk: CDC logic required in apb_config.sv - If pclk = aclk: Direct connection (no CDC)

CDC Implementation: - Use apb_slave_cdc wrapper (like HPET example) - apb_slave_cdc implements **handshake-based CDC** using cdc_handshake modules - One cdc_handshake for command interface (APB → core) - One cdc_handshake for response interface (core → APB) - Full req/ack handshake protocol (NOT async FIFO) - Works across all frequency ratios (slow-to-fast, fast-to-slow, any ratio)

5.3 Reset

5.3.1 Primary Reset: aresetn

Specification: - **Name:** aresetn (AXI reset, active-low) - **Type:** Asynchronous assertion, synchronous deassertion - **Polarity:** Active-low (0 = reset, 1 = normal operation) - **Assertion:** Asynchronous (can occur at any time) - **Deassertion:** Synchronous to aclk rising edge - **Duration:** Minimum 10 aclk cycles

Reset Behavior:

```
// Standard reset pattern for all STREAM modules
always_ff @(posedge aclk or negedge aresetn) begin
    if (!aresetn) begin
        // Asynchronous reset assertion
        r_state <= IDLE;
        r_counter <= '0;
        r_valid <= 1'b0;
        // ... all registers to known state
```

```

    end else begin
        // Synchronous operation
        r_state <= w_next_state;
        // ... normal logic
    end
end

```

5.3.2 Secondary Reset: presetn

Specification: - **Name:** presetn (APB reset, active-low) - **Type:** Asynchronous assertion, synchronous deassertion - **Polarity:** Active-low (0 = reset, 1 = normal operation) - **Synchronization:** May be asynchronous to aresetn

Usage: - APB configuration interface only - Typically tied to aresetn if pclk = aclk

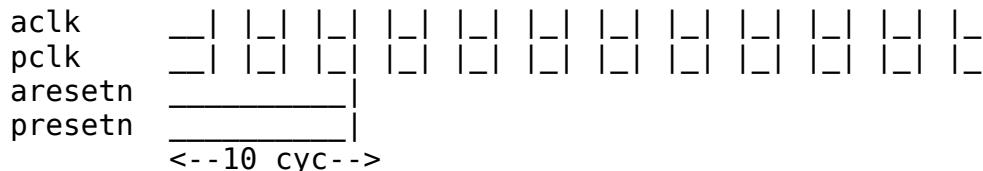
5.4 Reset Sequencing

5.4.1 Power-On Reset

Recommended sequence:

1. Assert aresetn (LOW)
2. Assert presetn (LOW)
3. Apply stable clocks (aclk, pclk)
4. Wait 10 aclk cycles
5. Deassert presetn (HIGH) on pclk rising edge
6. Deassert aresetn (HIGH) on aclk rising edge
7. Wait 5 aclk cycles for stabilization
8. Begin APB configuration

Timing diagram:



5.4.2 Functional Reset

Software-initiated reset (per channel):

```
// Reset specific channel via APB
write_apb(ADDR_GLOBAL_CTRL, CHANNEL_0_RESET); // Auto-clears after 1
cycle
```

// Hardware response:

```

// - Channel FSM returns to IDLE
// - Channel registers cleared
// - Outstanding transactions flushed
// - MonBus error packet generated (if mid-transfer)

```

5.4.3 Reset Recovery

After reset deassertion:

Reset Recovery

Cycle	Event
0	aresetn deasserted (rising edge)
1-5	Internal state stabilization
6+	Ready for APB configuration
10+	Ready for descriptor transfers

5.5 Clock Requirements by Module

5.5.1 Functional Unit Blocks (FUB)

Functional Unit Blocks

Module	Clock	Reset	Frequency	Notes
descriptor_engine	aclk	aresetn	100-500 MHz	AXI master timing
scheduler	aclk	aresetn	100-500 MHz	Single cycle FSM
axi_read_engine	aclk	aresetn	100-500 MHz	AXI master timing
axi_write_engine	aclk	aresetn	100-500 MHz	AXI master timing
simple_sram	aclk	aresetn	100-500 MHz	Synchronous SRAM

5.5.2 Integration Blocks (MAC)

Integration Blocks

Module	Clock(s)	Reset(s)	Frequency	Notes
channel_ar	aclk	aresetn	100-500 MHz	Single

Module	Clock(s)	Reset(s)	Frequency	Notes
biter				cycle arbitration
apb_config	pclk, (aclk)	presetn, (aresetn)	50-200 MHz (APB)	CDC if async
monbus_axi_l_group	aclk	aresetn	100-500 MHz	AXIL timing
stream_top	aclk, pclk	aresetn, presetn	Mixed	Top-level

5.6 Timing Constraints

5.6.1 Setup and Hold Times

Internal registers (relative to aclk): - Setup time: 0.5 ns (typical) - Hold time: 0.1 ns (typical) - Clock-to-Q: 0.3 ns (typical)

External interfaces: - AXI/AXIL: Per ARM IH10022E specification - APB: Per ARM IH10024C specification

5.6.2 Critical Paths

Identified critical paths:

1. **Arbiter -> Scheduler grant:**
 - Latency: 1 cycle
 - Path: Priority encoder -> One-hot grant
2. **AXI read -> SRAM write:**
 - Latency: 1 cycle
 - Path: R data -> SRAM write port
3. **SRAM read -> AXI write:**
 - Latency: 1 cycle
 - Path: SRAM read port -> W data

Maximum frequency estimation: - Typical FPGA (Xilinx 7-series): 250 MHz - High-end FPGA (UltraScale+): 400 MHz - ASIC (28nm): 500 MHz

5.7 Clock Domain Crossing (CDC)

5.7.1 APB Configuration CDC

When required: pclk aclk (asynchronous APB interface)

CDC Implementation:

```
// APB to STREAM domain (pclk -> aclk)
apb_slave_cdc #(
    .ADDR_WIDTH(32),
    .DATA_WIDTH(32),
    .SYNC_STAGES(2) // Dual-flop synchronizer
) u_apb_cdc (
    // APB side (pclk domain)
    .s_pclk(pclk),
    .s_presetn(presetn),
    .s_paddr(paddr),
    .s_pwrite(pwrite),
    .s_pwdata(pwdata),
    .s_prdata(prdata),
    // STREAM side (aclk domain)
    .m_pclk(aclk),
    .m_presetn(aresetn),
    .m_paddr(paddr_sync),
    .m_pwrite(pwrite_sync),
    .m_pwdata(pwdata_sync),
    .m_prdata(prdata_sync)
);
```

CDC Mechanism: - apb_slave_cdc uses **handshake-based CDC** via cdc_handshake modules (NOT async FIFOs) - **Command path** (pclk → aclk): APB write/read commands cross via req/ack handshake - **Response path** (aclk → pclk): APB read data crosses back via req/ack handshake - **Internal synchronizers:** Dual-flop synchronizers (2-3 stages) for handshake signals - **ASYNC_REG attribute:** Applied to synchronizer stages for timing tools - **Timing constraints:** Proper constraints required in SDC/XDC for synchronizer paths

Handshake Protocol Benefits: - Works across **all frequency ratios** (slow-to-fast, fast-to-slow, arbitrary ratios) - Guaranteed data integrity (req/ack ensures data stability before sampling) - No FIFO depth management or gray code pointer complexity - Latency: 4-6 APB clock cycles for register access (handshake round-trip)

5.7.2 No CDC Required

Single clock domain: If pclk = aclk and presetn = aresetn:

```
// Direct connection (no CDC wrapper)
apb_config #()
```

```

    .NUM_CHANNELS(8)
) u_apb_config (
    .pclk(aclk),           // Same clock
    .presetn(aresetn),    // Same reset
    // ... direct APB signals
);

```

5.8 Reset State Initialization

5.8.1 Register Reset Values

All STREAM modules must initialize to known state on reset:

```

// Descriptor Engine
if (!aresetn) begin
    r_desc_fifo_wr_ptr <= '0;
    r_desc_valid <= 1'b0;
    r_desc_error <= 1'b0;
end

// Scheduler
if (!aresetn) begin
    r_current_state <= CH_IDLE;
    r_read_beats_remaining <= '0;
    r_write_beats_remaining <= '0;
    r_timeout_counter <= '0;
end

// AXI Engines
if (!aresetn) begin
    r_burst_counter <= '0;
    m_axi_arvalid <= 1'b0;
    m_axi_awvalid <= 1'b0;
end

// Arbiter
if (!aresetn) begin
    r_last_grant_id <= '0;
    r_grant_valid <= 1'b0;
end

```

5.8.2 SRAM Reset

SRAM contents: Undefined after reset (no initialization required)

SRAM pointers:

```
if (!aresetn) begin
    wr_ptr <= '0;
    rd_ptr <= '0;
end
```

5.9 Clock Gating (Optional)

For power optimization in ASIC implementations:

5.9.1 Per-Channel Clock Gating

```
// Clock gate when channel idle
clock_gate_ctrl u_ch0_clk_gate (
    .clk_in(aclk),
    .enable(ch0_enable),
    .clk_out(ch0_gated_clk)
);

// Use gated clock for channel logic
scheduler #(CHANNEL_ID(0)) u_ch0_sched (
    .aclk(ch0_gated_clk), // Gated clock
    .aresetn(aresetn),
    // ...
);
```

Note: Clock gating typically not used in FPGA implementations (tutorial focus).

5.10 Verification Requirements

5.10.1 Clock Checks

Testbench must verify: - [Done] Clock period consistent - [Done] Clock duty cycle 50% tolerance -
[Done] No glitches on clock - [Done] Setup/hold times met

5.10.2 Reset Checks

Testbench must verify: - [Done] All registers initialize to known state - [Done] Reset assertion
clears FSMs to IDLE - [Done] Reset deassertion synchronous to clock - [Done] Minimum reset
duration (10 cycles) enforced - [Done] Operations don't start until stabilization complete

5.10.3 CDC Checks

For APB CDC (if present): - [Done] No metastability violations - [Done] Data integrity across
domains - [Done] Proper flag synchronization

5.11 Example Reset Testbench

```
# CocoTB testbench pattern
class StreamTB(TBBase):
    async def setup_clocks_and_reset(self):
        """Complete clock and reset initialization"""
        # Start clocks
        await self.start_clock('aclk', freq=10, units='ns') # 100 MHz
        await self.start_clock('pclk', freq=20, units='ns') # 50 MHz
    (async)
        # Assert reset
        await self.assert_reset()

        # Hold reset for 10 aclk cycles
        await self.wait_clocks('aclk', 10)

        # Deassert reset (synchronous to aclk)
        await self.deassert_reset()

        # Stabilization period
        await self.wait_clocks('aclk', 5)

        # Ready for operation

    async def assert_reset(self):
        """Assert both resets"""
        self.dut.aresetn.value = 0
        self.dut.presetn.value = 0

    async def deassert_reset(self):
        """Deassert both resets synchronously"""
        # Wait for rising edge of aclk
        await RisingEdge(self.dut.aclk)
        self.dut.aresetn.value = 1

        # Wait for rising edge of pclk
        await RisingEdge(self.dut.pclk)
        self.dut.presetn.value = 1
```

5.12 Related Documentation

- **Scheduler FSM:** fub_02_scheduler.md - Reset behavior
 - **APB Config:** mac_02_apb_config.md - CDC implementation
 - **Top-Level: mac_04_stream_top.md - Clock/reset integration**
-

5.13 Revision History

Clocks and Reset Specification Revision History

Version	Date	Author	Description
0.90	2025-11-22	seang	Initial block specification
0.91	2026-01-02	seang	Added table captions and figure numbers

Last Updated: 2026-01-02

6 STREAM Core Specification

Module: stream_core.sv **Location:** projects/components/stream/rtl/macro/ **Status:** Implemented **Last Updated:** 2025-11-30

6.1 Overview

The STREAM Core is the top-level integration module that combines all STREAM components into a complete scatter-gather DMA engine. It provides 8 independent channels with descriptor-based memory-to-memory transfers.

6.1.1 Key Features

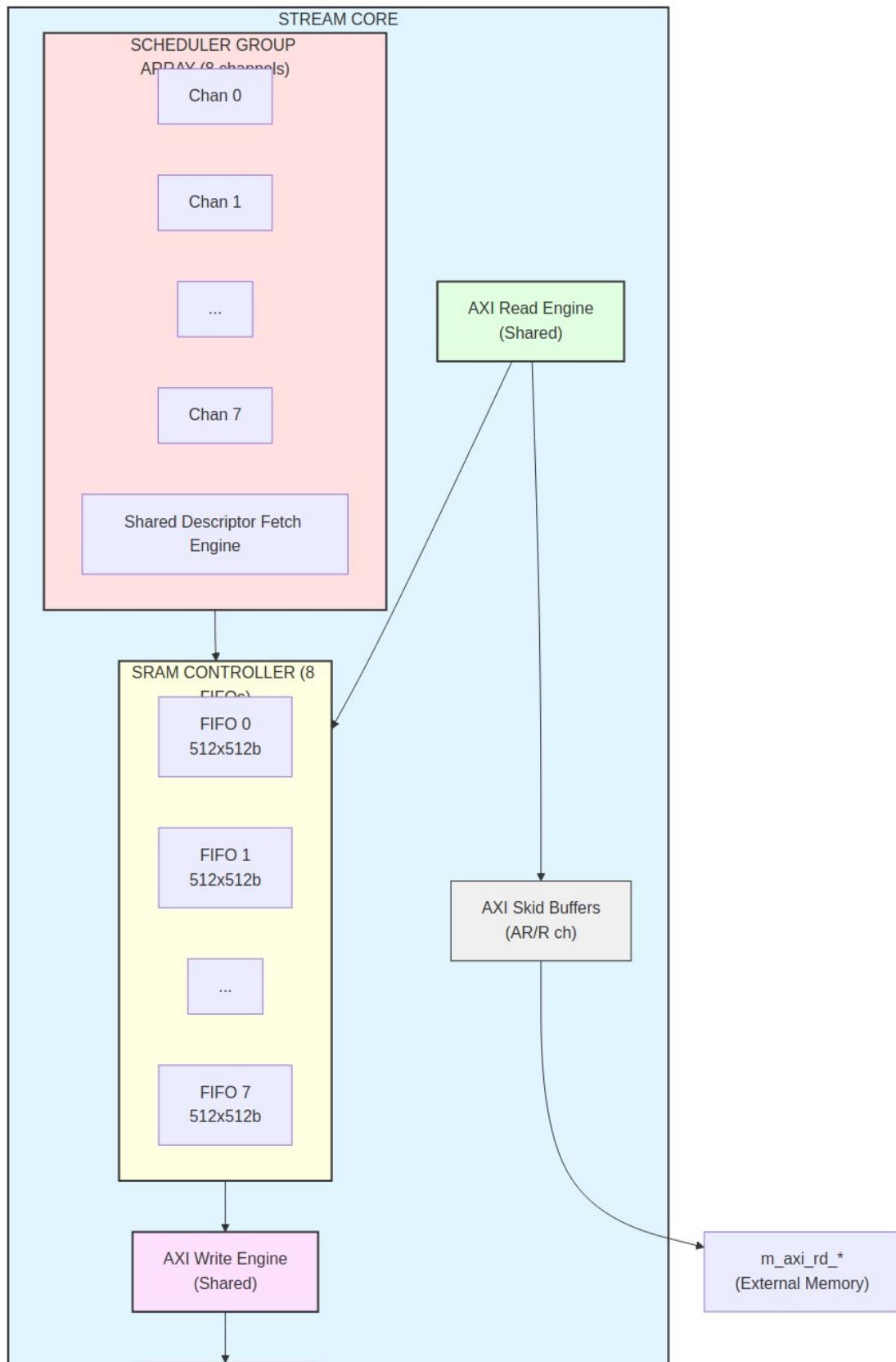
- **8 Independent Channels:** Concurrent descriptor-based transfers
- **Shared AXI Masters:** Three shared AXI4 masters for efficiency
 - Descriptor fetch (256-bit fixed)
 - Data read (parameterizable, default 512-bit)
 - Data write (parameterizable, default 512-bit)

- **Per-Channel Buffering:** Independent FIFO per channel (512 entries default)
- **Performance Monitoring:** Integrated profiler with FIFO readout
- **AXI Skid Buffers:** Timing closure on all external interfaces
- **Unified MonBus:** Single monitor bus output for all events

6.1.2 Block Diagram

The STREAM Core integrates the following major components:

6.1.3 Figure 2.1.1: STREAM Core Block Diagram



STREAM Core Block Diagram

Source: [01_stream_core_block.mmd](#)

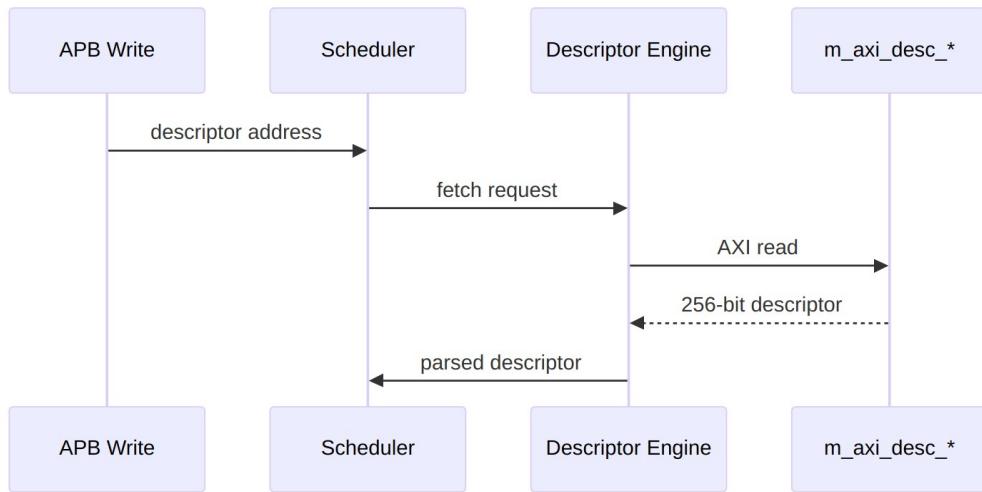
6.2 Architecture

6.2.1 Component Hierarchy

1. **scheduler_group_array** - Top scheduler layer
 - 8 × scheduler_group instances
 - Shared descriptor engine
 - Descriptor fetch arbitration
2. **sram_controller** - Buffering layer
 - 8 × independent FIFOs (gaxi_fifo_sync)
 - Per-channel allocation controllers
 - Per-channel drain controllers
3. **axi_read_engine** - Read datapath
 - Shared AXI master for all channels
 - ID-based routing to SRAM
 - Space allocation flow control
4. **axi_write_engine** - Write datapath
 - Shared AXI master for all channels
 - ID-based routing from SRAM
 - Drain reservation flow control
5. **perf_profiler** - Performance monitoring
 - Transaction counting
 - Bandwidth tracking
 - FIFO readout interface
6. **AXI Skid Buffers** - Timing closure
 - Descriptor AXI (AR/R)
 - Read AXI (AR/R)
 - Write AXI (AW/W/B)

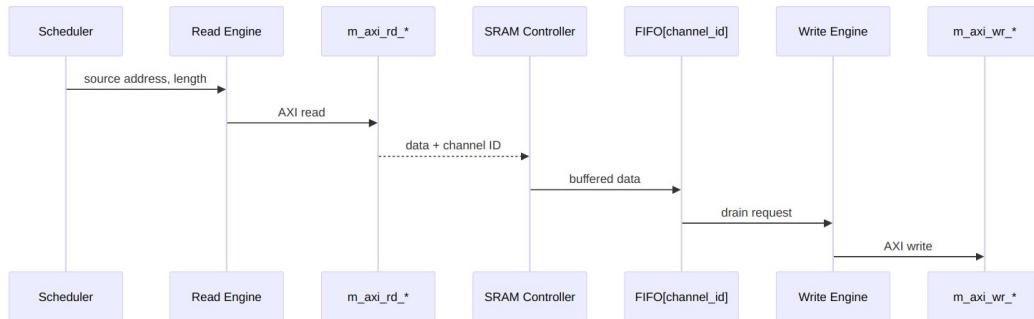
6.2.2 Data Flow

Descriptor Fetch Flow:



.1.2: Descriptor Fetch Flow

Data Transfer Flow:



.1.3: Data Transfer Flow

6.3 Parameters

6.3.1 Primary Configuration

Primary Configuration

Parameter	Type	Default	Description
<code>NUM_CHANNELS</code>	int	8	Number of independent DMA channels
<code>CHAN_WIDTH</code>	int	<code>\$clog2(NUM_CHANNELS)</code>	Channel ID width (3 for 8 channels)
<code>ADDR_WIDTH</code>	int	64	Address bus width

Parameter	Type	Default	Description
DATA_WIDTH	int	512	Data bus width (must match memory interface)
AXI_ID_WIDTH	int	8	AXI transaction ID width
FIFO_DEPTH	int	512	Per-channel FIFO depth

6.3.2 Monitor Control

Monitor Control

Parameter	Type	Default	Description
USE_AXI_MONITORS	int	1	Enable (1) or disable (0) AXI transaction monitors

Note: When USE_AXI_MONITORS = 0: - All monitor configuration inputs are tied off internally - MonBus output remains present but inactive (no packets generated) - Reduces resource usage for production systems

6.3.3 Outstanding Transaction Limits

Outstanding Transaction Limits

Parameter	Default	Description
AR_MAX_OUTSTANDING	8	Maximum concurrent read address requests
AW_MAX_OUTSTANDING	8	Maximum concurrent write address requests

6.3.4 AXI Skid Buffer Depths

AXI Skid Buffer Depths

Parameter	Default	Purpose
SKID_DEPTH_AR	2	AR channel timing closure
SKID_DEPTH_R	4	R channel timing closure
SKID_DEPTH_AW	2	AW channel timing closure

Parameter	Default	Purpose
SKID_DEPTH_W	4	W channel timing closure
SKID_DEPTH_B	2	B channel timing closure

Note: Deeper buffers on data channels (R/W) improve throughput.

6.3.5 MonBus Agent IDs

MonBus Agent IDs

Parameter	Default	Description
DESC_MON_BASE_AGENT_ID	16 (0x10)	Descriptor engines base (16-23)
SCHED_MON_BASE_AGENT_ID	48 (0x30)	Schedulers base (48-55)
DESC_AXI_MON_AGENT_ID	8 (0x08)	Descriptor AXI master monitor
MON_UNIT_ID	1 (0x1)	Unit ID for all STREAM events

6.4 Port List

6.4.1 Clock and Reset

Clock and Reset

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

6.4.2 APB Programming Interface

Per-channel descriptor kick-off interface:

APB Programming Interface

Signal	Direction	Width	Description
apb_valid[ch]	input	NUM_CHAN	Channel descriptor

Signal	Direction	Width	Description
		NELS	address valid
apb_ready[ch]	output	NUM_CHAN	Channel ready to accept descriptor address
apb_addr[ch]	input	NELS NUM_CHAN NELS × ADDR_WIDTH H	Descriptor address per channel

Usage:

```
// Kick off channel 0 with descriptor at 0x1000_0000
apb_valid[0] = 1'b1;
apb_addr[0] = 64'h0000_0000_1000_0000;
// Wait for handshake
wait (apb_ready[0]);
apb_valid[0] = 1'b0;
```

6.4.3 Configuration Interface

Per-Channel Configuration:

Configuration Interface

Signal	Direction	Width	Description
cfg_channel_enable[ch]	input	NUM_CHAN NELS	Enable channel
cfg_channel_reset[ch]	input	NUM_CHAN NELS	Soft reset channel (FSM → IDLE)

Global Scheduler Configuration:

Configuration Interface

Signal	Direction	Width	Description
cfg_sched_enable	input	1	Global scheduler enable
cfg_sched_timeout_cycles	input	16	Timeout threshold (cycles)
cfg_sched_timeout_enable	input	1	Enable timeout detection
cfg_sched_error_report_enable	input	1	Enable error event reporting

Signal	Direction	Width	Description
cfg_sched_co_mpl_enable	input	1	Enable completion event reporting
cfg_sched_pe_rf_enable	input	1	Enable performance event reporting

Descriptor Engine Configuration:

Configuration Interface

Signal	Direction	Width	Description
cfg_desceng_enable	input	1	Enable descriptor engine
cfg_desceng_prefetch	input	1	Enable descriptor prefetch
cfg_desceng_fifo_thresh	input	4	FIFO threshold for prefetch
cfg_desceng_addr0_base	input	ADDR_WIDT_H	Base address limit 0
cfg_desceng_addr0_limit	input	ADDR_WIDT_H	Limit address limit 0
cfg_desceng_addr1_base	input	ADDR_WIDT_H	Base address limit 1
cfg_desceng_addr1_limit	input	ADDR_WIDT_H	Limit address limit 1

AXI Monitor Configuration:

Three identical sets of monitor config signals (descriptor, read, write):

Configuration Interface

Signal Prefix	Applies To	Description
cfg_desc_mon_*	Descriptor AXI	Descriptor fetch monitoring
cfg_rdeng_mon_*	Read AXI	Data read monitoring
cfg_wreng_mon_*	Write AXI	Data write monitoring

Each monitor has:

Signal Suffix	Width	Description
_enable	1	Enable monitor
_err_enable	1	Enable error reporting
_perf_enable	1	Enable performance reporting
_timeout_enable	1	Enable timeout detection
_timeout_cycles	32	Timeout threshold
_latency_thresh	32	Latency threshold for events
_pkt_mask	16	Packet type mask
_err_select	4	Error type selector
_err_mask	8	Error event mask
_timeout_mask	8	Timeout event mask
_compl_mask	8	Completion event mask
_thresh_mask	8	Threshold event mask
_perf_mask	8	Performance event mask
_addr_mask	8	Address event mask
_debug_mask	8	Debug event mask

AXI Transfer Configuration:

Signal	Direction	Width	Description
cfg_axi_rd_xf er_beats	input	8	Read burst size (beats)
cfg_axi_wr_xf er_beats	input	8	Write burst size (beats)

Performance Profiler Configuration:

Signal	Direction	Width	Description
cfg_perf_enab_le	input	1	Enable profiler
cfg_perf_mode	input	1	Profiler mode (0=count, 1=latency)
cfg_perf_clea_r	input	1	Clear profiler counters

6.4.4 Status Interface

System-Level Status:

Status Interface

Signal	Direction	Width	Description
system_idle	output	1	All channels idle (AND of all scheduler_idle)

Per-Channel Status:

Status Interface

Signal	Direction	Width	Description
descriptor_engine_idle[ch]	output	NUM_CHAN NELS	Descriptor engine idle
scheduler_idle[ch]	output	NUM_CHAN NELS	Scheduler in IDLE state
scheduler_state[ch]	output	NUM_CHAN NELS × 7	Scheduler FSM state (ONE-HOT)
sched_error[ch]	output	NUM_CHAN NELS	Scheduler error (sticky)
axi_rd_all_complete[ch]	output	NUM_CHAN NELS	All read transactions complete
axi_wr_all_complete[ch]	output	NUM_CHAN NELS	All write transactions complete

Performance Profiler Status:

Status Interface

Signal	Direction	Width	Description
perf_fifo_empty	output	1	Profiler FIFO empty
perf_fifo_full	output	1	Profiler FIFO full
perf_fifo_count	output	16	Profiler FIFO occupancy
perf_fifo_rd	input	1	Read profiler entry
perf_fifo_data_low	output	32	Profiler data [31:0]
perf_fifo_data_high	output	32	Profiler data [63:32]

6.4.5 AXI4 Master - Descriptor Fetch (256-bit)

AR Channel:

AXI4 Master - Descriptor Fetch

Signal	Direction	Width	Description
m_axi_desc_ar_id	output	AXI_ID_WIDTH	Transaction ID
m_axi_desc_ar_addr	output	ADDR_WIDTH	Address
m_axi_desc_ar_len	output	8	Burst length - 1
m_axi_desc_ar_size	output	3	Burst size (log2 bytes)
m_axi_desc_ar_burst	output	2	Burst type (INCR)
m_axi_desc_ar_lock	output	1	Lock type
m_axi_desc_ar_cache	output	4	Cache attributes
m_axi_desc_ar_prot	output	3	Protection attributes

Signal	Direction	Width	Description
m_axi_desc_ar_qos	output	4	QoS value
m_axi_desc_ar_region	output	4	Region identifier
m_axi_desc_ar_user	output	CHAN_WIDTH	User signal (channel ID)
m_axi_desc_ar_valid	output	1	Address valid
m_axi_desc_ar_ready	input	1	Address ready

R Channel:

AXI4 Master - Descriptor Fetch

Signal	Direction	Width	Description
m_axi_desc_r_id	input	AXI_ID_WID TH	Transaction ID
m_axi_desc_r_data	input	256	Read data (FIXED 256-bit)
m_axi_desc_r_resp	input	2	Response (OKAY/EXOKAY/SLVERR/DECERR)
m_axi_desc_r_last	input	1	Last beat of burst
m_axi_desc_r_user	input	CHAN_WIDT H	User signal (channel ID)
m_axi_desc_r_valid	input	1	Read data valid
m_axi_desc_r_ready	output	1	Read data ready

6.4.6 AXI4 Master - Data Read (Parameterizable Width)

AR Channel:

AXI4 Master - Data Read

Signal	Direction	Width	Description
m_axi_rd_arid	output	AXI_ID_WIDTH	Transaction ID

Signal	Direction	Width	Description
m_axi_rd_arad dr	output	ADDR_WIDTH	Address
m_axi_rd_arle n	output	8	Burst length - 1
m_axi_rd_arsi ze	output	3	Burst size (log2 bytes)
m_axi_rd_arbu rst	output	2	Burst type (INCR)
m_axi_rd_arlo ck	output	1	Lock type
m_axi_rd_arca che	output	4	Cache attributes
m_axi_rd_arpr ot	output	3	Protection attributes
m_axi_rd_arqo s	output	4	QoS value
m_axi_rd_arre gion	output	4	Region identifier
m_axi_rd_arus er	output	CHAN_WIDTH	User signal (channel ID)
m_axi_rd_arva lid	output	1	Address valid
m_axi_rd_arre ady	input	1	Address ready

R Channel:

AXI4 Master - Data Read

Signal	Direction	Width	Description
m_axi_rd_rid	input	AXI_ID_WIDTH	Transaction ID
m_axi_rd_rdat a	input	DATA_WIDTH	Read data (default 512-bit)
m_axi_rd_rres p	input	2	Response
m_axi_rd_rlas t	input	1	Last beat of burst

Signal	Direction	Width	Description
m_axi_rd_ruse r	input	CHAN_WIDTH	User signal (channel ID)
m_axi_rd_rval id	input	1	Read data valid
m_axi_rd_rrea dy	output	1	Read data ready

6.4.7 AXI4 Master - Data Write (Parameterizable Width)

AW Channel:

AXI4 Master - Data Write

Signal	Direction	Width	Description
m_axi_wr_awid	output	AXI_ID_WIDTH	Transaction ID
m_axi_wr_awad dr	output	ADDR_WIDTH	Address
m_axi_wr_awle n	output	8	Burst length - 1
m_axi_wr_awsi ze	output	3	Burst size (log2 bytes)
m_axi_wr_awbu rst	output	2	Burst type (INCR)
m_axi_wr_awlo ck	output	1	Lock type
m_axi_wr_awca che	output	4	Cache attributes
m_axi_wr_awpr ot	output	3	Protection attributes
m_axi_wr_awqo s	output	4	QoS value
m_axi_wr_awre gion	output	4	Region identifier
m_axi_wr_awus er	output	CHAN_WIDTH	User signal (channel ID)
m_axi_wr_awva lid	output	1	Address valid

Signal	Direction	Width	Description
m_axi_wr_awre _{ady}	input	1	Address ready

W Channel:

AXI4 Master - Data Write

Signal	Direction	Width	Description
m_axi_wr_wda _{ta}	output	DATA_WIDTH	Write data (default 512-bit)
m_axi_wr_wst _{rb}	output	DATA_WIDTH/8	Write strobes (byte enables)
m_axi_wr_wla _{st}	output	1	Last beat of burst
m_axi_wr_wus _{er}	output	CHAN_WIDTH	User signal (channel ID)
m_axi_wr_wva _{lid}	output	1	Write data valid
m_axi_wr_wre _{ady}	input	1	Write data ready

B Channel:

AXI4 Master - Data Write

Signal	Direction	Width	Description
m_axi_wr_bid	input	AXI_ID_WIDTH	Transaction ID
m_axi_wr_bres _p	input	2	Response
m_axi_wr_buse _r	input	CHAN_WIDTH	User signal (channel ID)
m_axi_wr_bval _{id}	input	1	Response valid
m_axi_wr_brea _{dy}	output	1	Response ready

6.4.8 Status/Debug Outputs

Descriptor AXI Monitor:

Status/Debug Outputs

Signal	Direction	Width	Description
cfg_sts_desc_mon_busy	output	1	Monitor busy
cfg_sts_desc_mon_active_txns	output	8	Active transaction count
cfg_sts_desc_mon_error_count	output	16	Error count
cfg_sts_desc_mon_txn_count	output	32	Total transaction count
cfg_sts_desc_mon_conflict_error	output	1	ID conflict detected

Read Engine AXI Monitor:

Status/Debug Outputs

Signal	Direction	Width	Description
cfg_sts_rden_g_skid_busy	output	1	Skid buffer busy
cfg_sts_rden_g_mon_active_txns	output	8	Active transaction count
cfg_sts_rden_g_mon_error_count	output	16	Error count
cfg_sts_rden_g_mon_txn_count	output	32	Total transaction count
cfg_sts_rden_g_mon_conflict_error	output	1	ID conflict detected

Write Engine AXI Monitor:

Status/Debug Outputs

Signal	Direction	Width	Description
cfg_sts_wren_g_skid_busy	output	1	Skid buffer busy

Signal	Direction	Width	Description
cfg_sts_wren_g_mon_active_txns	output	8	Active transaction count
cfg_sts_wren_g_mon_error_count	output	16	Error count
cfg_sts_wren_g_mon_txn_count	output	32	Total transaction count
cfg_sts_wren_g_mon_conflict_error	output	1	ID conflict detected

6.4.9 Unified Monitor Bus Interface

Unified Monitor Bus Interface

Signal	Direction	Width	Description
mon_valid	output	1	Monitor packet valid
mon_ready	input	1	Monitor packet ready
mon_packet	output	64	Monitor packet data

MonBus Sources: - Descriptor engines (8 sources, agent IDs 16-23) - Schedulers (8 sources, agent IDs 48-55) - Descriptor AXI monitor (agent ID 8) - Read AXI monitor (configurable) - Write AXI monitor (configurable)

6.5 Operation

6.5.1 Transfer Initialization

Step 1: Configuration

1. Configure global settings (timeouts, monitors, transfer sizes)
2. Enable descriptor engine (cfg_desceng_enable = 1)
3. Enable scheduler (cfg_sched_enable = 1)
4. Enable target channel (cfg_channel_enable[ch] = 1)

Step 2: Descriptor Kick-off

1. Assert apb_valid[ch] = 1
2. Provide descriptor address on apb_addr[ch]
3. Wait for apb_ready[ch] handshake
4. Deassert apb_valid[ch]

Step 3: Automatic Transfer

1. Descriptor engine fetches descriptor via m_axi_desc_*
2. Scheduler receives descriptor, starts transfer
3. Read engine: memory → SRAM (via m_axi_rd_*)
4. Write engine: SRAM → memory (via m_axi_wr_*)
5. Scheduler reports completion via MonBus

6.5.2 Monitoring Transfer Progress

Check Scheduler State:

```
// Monitor scheduler state
case (scheduler_state[ch])
  7'b00000001: // CH_IDLE - waiting for descriptor
  7'b00000010: // CH_FETCH_DESC - fetching descriptor
  7'b00000100: // CH_XFER_DATA - transfer in progress
  7'b00010000: // CH_COMPLETE - transfer complete
  7'b00100000: // CH_NEXT_DESC - chaining to next
  7'b01000000: // CH_ERROR - error occurred
endcase
```

Check Completion:

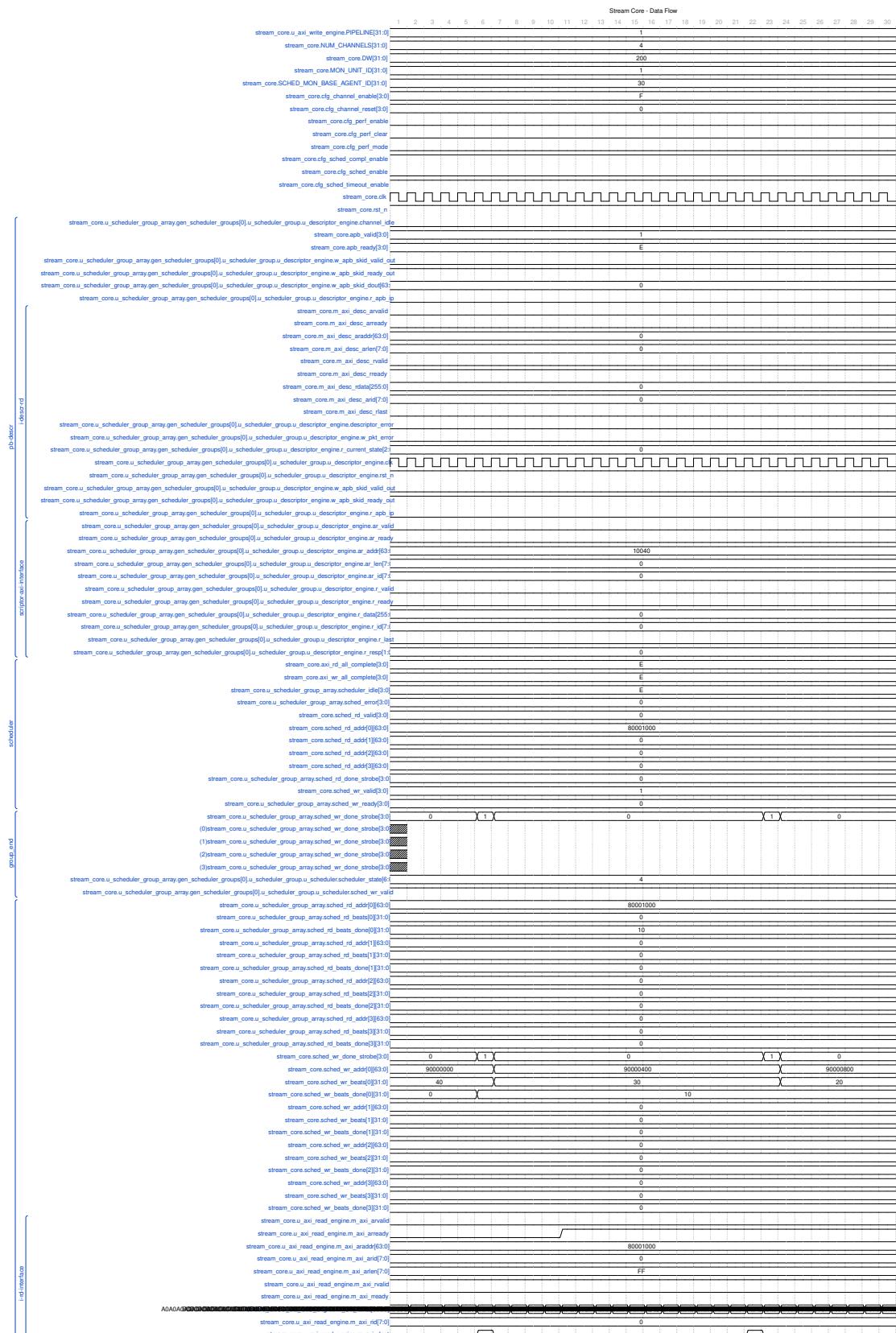
```
// All complete when:
complete = scheduler_idle[ch] &&
           axi_rd_all_complete[ch] &&
           axi_wr_all_complete[ch];
```

6.6 Timing Diagrams

6.6.1 Complete Data Flow Timing

The following diagram shows the complete data flow through the STREAM core, from descriptor kick-off through data transfer completion:

6.6.1.1 Waveform 2.1.1: STREAM Core Data Flow



STREAM Core Data Flow

Source: [stream_core_data_flow.json](#)

6.7 Testing

Test Location: projects/components/stream/dv/tests/top/

Key Test Scenarios:

1. **Single channel transfer** - Basic end-to-end operation
2. **Multi-channel concurrent** - 2-8 channels simultaneously
3. **Descriptor chaining** - 2-5 descriptors linked
4. **FIFO overflow prevention** - Large transfer with small FIFO
5. **Error handling** - AXI errors, timeouts
6. **Performance profiling** - Bandwidth measurements
7. **MonBus event checking** - Verify all events reported

Test Configuration:

```
# Basic configuration
NUM_CHANNELS = 4
DATA_WIDTH = 128
FIFO_DEPTH = 512
cfg_axi_rd_xfer_beats = 16
cfg_axi_wr_xfer_beats = 16
```

6.8 Resource Utilization

Estimated Resources (8 channels, 512-bit data, 512-deep FIFOs):

Resource Utilization

Component	Quantity	Est. Size
Schedulers	8	8 × ~500 FFs
Descriptor Engine	1	~1000 FFs
SRAM FIFOs	8	8 × 512 × 512-bit = 256KB
AXI Engines	2	2 × ~2000 FFs

Component	Quantity	Est. Size
Skid Buffers	3 sets	~2000 FFs
Monitors	3	3 × ~1000 FFs
Total		~20K FFs + 256KB SRAM

Critical Paths: - AXI handshake paths (improved by skid buffers) - SRAM address decode - MonBus arbiter

6.9 Integration Example

```

stream_core #(
    .NUM_CHANNELS(8),
    .DATA_WIDTH(512),
    .FIFO_DEPTH(512)
) u_stream (
    .clk                      (system_clk),
    .rst_n                    (system_rst_n),
    // APB kick-off
    .apb_valid                (stream_apb_valid),
    .apb_ready                (stream_apb_ready),
    .apb_addr                 (stream_apb_addr),
    // Configuration
    .cfg_channel_enable       (stream_ch_enable),
    .cfg_sched_enable         (1'b1),
    .cfg_axi_rd_xfer_beats   (8'd16),
    .cfg_axi_wr_xfer_beats   (8'd16),
    // ... other config

    // AXI Descriptor Master
    .m_axi_desc_arid          (desc_arid),
    .m_axi_desc_araddr         (desc_araddr),
    // ... full AXI AR/R

    // AXI Read Master
    .m_axi_rd_arid            (rd_arid),
    .m_axi_rd_araddr           (rd_araddr),
    // ... full AXI AR/R

    // AXI Write Master
    .m_axi_wr_awid             (wr_awid),

```

```

.m_axi_wr_awaddr      (wr_awaddr),
// ... full AXI AW/W/B

// MonBus
.mon_valid           (stream_mon_valid),
.mon_ready            (stream_mon_ready),
.mon_packet           (stream_mon_packet)
);

```

6.10 Related Documentation

- **Scheduler Group Array:** 02_scheduler_group_array.md - Multi-channel scheduler integration
- **Scheduler Group:** 03_scheduler_group.md - Single channel scheduler + descriptor engine
- **Scheduler:** 04_scheduler.md - Channel state machine and transfer coordination
- **Descriptor Engine:** 05_descriptor_engine.md - Descriptor fetch and parsing
- **AXI Read Engine:** 06_axi_read_engine.md - Read datapath
- **Stream Alloc Ctrl:** 07_stream_alloc_ctrl.md - Space allocation controller
- **SRAM Controller:** 08_sram_controller.md - Per-channel buffering
- **SRAM Controller Unit:** 09_sram_controller_unit.md - Single channel SRAM unit
- **Stream Latency Bridge:** 10_stream_latency_bridge.md - Timing bridge
- **Stream Drain Ctrl:** 11_stream_drain_ctrl.md - Drain flow controller
- **AXI Write Engine:** 12_axi_write_engine.md - Write datapath
- **APB to Descriptor:** 13_apbtodescr.md - APB configuration interface
- **Performance Profiler:** 15_perf_profiler.md - Performance monitoring

- **MonBus AXI-Lite Group: 16_monbus_axil_group.md - Monitor bus arbitration**
-

6.11 Revision History

STREAM Core Specification Revision History

Version	Date	Author	Description
0.90	2025-11-22	seang	Initial block specification
0.91	2026-01-02	seang	Added table captions and figure numbers

Last Updated: 2026-01-02

7 Scheduler Group Array

Module: scheduler_group_array.sv **Location:** projects/components/stream/rtl/macro/
Category: MACRO (Top-level Integration) **Parent:** stream_core.sv **Status:** Implemented **Last Updated:** 2025-11-30

7.1 Overview

The scheduler_group_array module is the top-level array of 8 scheduler_group instances that provides multi-channel DMA with shared resources. It serves as the scheduler layer of the STREAM engine, handling descriptor fetch arbitration and routing data path interfaces.

7.1.1 Key Features

- **8 Independent Channels:** Each channel has its own scheduler_group instance
- **Shared Descriptor AXI Master:** Round-robin arbitration for descriptor fetches
- **Per-Channel Data Interfaces:** Direct passthrough to AXI engines (no arbitration)

- **Unified MonBus Aggregation:** 9 sources (8 channels + descriptor AXI monitor)
- **ID-Based R Channel Routing:** Channel ID embedded in AXI ID for response demux

7.1.2 Simplified from RAPIDS

This module is intentionally simplified from RAPIDS for tutorial focus: - 8 channels (vs 32 in RAPIDS) - No control read/write engines - No program engine - No network interfaces (RDA, EOS completion) - No alignment bus - Direct data path interfaces (beats-based, no chunks)

7.2 Architecture

7.2.1 Component Hierarchy

```

scheduler_group_array
scheduler_group[0..7]           # 8 channel instances
    scheduler                   # Channel FSM
    descriptor_engine          # Descriptor fetch
arbiter_round_robin            # AR channel arbitration
axi4_master_rd_mon             # Descriptor AXI monitor
monbus_arbiter                 # 9-source MonBus aggregation

```

7.2.2 Descriptor AXI Arbitration

AR Channel Flow:

Channel 0..7 AR Requests -> Round-Robin Arbiter -> Single AR to External AXI

|
Embed channel_id in AXI ID lower bits

R Channel Demux:

External R Response -> Extract channel_id from RID[CHAN_WIDTH-1:0] -> Route to correct channel

Critical: Channel ID is embedded in the lower bits of the AXI ID field:

```

// In AR mux
desc_axi_int_arid = {{(AXI_ID_WIDTH-CHAN_WIDTH){1'b0}}, ch[CHAN_WIDTH-1:0]};

// In R demux
desc_r_channel_id = desc_axi_int_rid[CHAN_WIDTH-1:0];

```

7.2.3 Data Path Interfaces

IMPORTANT: Data read and write interfaces are NOT arbitrated at this level!

- `sched_rd_*` - Per-channel arrays passed directly to `axi_read_engine`
- `sched_wr_*` - Per-channel arrays passed directly to `axi_write_engine`

The AXI engines themselves handle channel arbitration internally.

7.3 Parameters

Parameters

Parameter	Type	Default	Description
<code>NUM_CHANNELS</code>	int	8	Number of DMA channels
<code>CHAN_WIDTH</code>	int	<code>\$clog2(NUM_CHANNELS)</code>	Channel ID width (3 bits for 8 channels)
<code>ADDR_WIDTH</code>	int	64	Address bus width
<code>DATA_WIDTH</code>	int	512	Data bus width
<code>AXI_ID_WIDTH</code>	int	8	AXI transaction ID width

7.3.1 Monitor Bus Agent IDs

Monitor Bus Agent IDs

Parameter	Default	Description
<code>DESC_MON_BASE_AGENT_ID</code>	16 (0x10)	Descriptor engine base (16-23)
<code>SCHED_MON_BASE_AGENT_ID</code>	48 (0x30)	Scheduler base (48-55)
<code>DESC_AXI_MON_AGENT_ID</code>	8 (0x08)	Descriptor AXI monitor
<code>MON_UNIT_ID</code>	1	Unit ID for all STREAM events
<code>MON_MAX_TRANSACTIONS</code>	16	Max outstanding transactions for monitor

7.4 Port List

7.4.1 Clock and Reset

Clock and Reset

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

7.4.2 APB Programming Interface (Per-Channel)

APB Programming Interface

Signal	Direction	Width	Description
apb_valid[ch]	input	NUM_CHAN NELS	Descriptor address valid
apb_ready[ch]	output	NUM_CHAN NELS	Ready to accept descriptor
apb_addr[ch]	input	NUM_CHAN NELS x ADDR_WIDT H	Descriptor address

7.4.3 Configuration Interface (Per-Channel)

Configuration Interface

Signal	Direction	Width	Description
cfg_channel_enable[ch]	input	NUM_CHANNEL LS	Enable channel
cfg_channel_reset[ch]	input	NUM_CHANNEL LS	Soft reset channel

7.4.4 Global Scheduler Configuration

Global Scheduler Configuration

Signal	Direction	Width	Description
cfg_sched_enable	input	1	Master scheduler

Signal	Direction	Width	Description
			enable
cfg_sched_tim_eout_cycles	input	16	Timeout threshold
cfg_sched_tim_eout_enable	input	1	Enable timeout detection
cfg_sched_err_enable	input	1	Enable error reporting
cfg_sched_compl_enable	input	1	Enable completion reporting
cfg_sched_perf_enable	input	1	Enable performance monitoring

7.4.5 Descriptor Engine Configuration

Descriptor Engine Configuration

Signal	Direction	Width	Description
cfg_desceng_enable	input	1	Enable descriptor engine
cfg_desceng_prefetch	input	1	Enable descriptor prefetch
cfg_desceng_fifo_thresh	input	4	FIFO threshold for prefetch
cfg_desceng_addr0_base	input	ADDR_WIDT_H	Address range 0 base
cfg_desceng_addr0_limit	input	ADDR_WIDT_H	Address range 0 limit
cfg_desceng_addr1_base	input	ADDR_WIDT_H	Address range 1 base
cfg_desceng_addr1_limit	input	ADDR_WIDT_H	Address range 1 limit

7.4.6 Status Interface (Per-Channel)

Status Interface

Signal	Direction	Width	Description
descriptor_engine_idle[ch]	output	NUM_CHAN NELS	Descriptor engine idle
scheduler_id_le[ch]	output	NUM_CHAN NELS	Scheduler idle
scheduler_state[ch]	output	NUM_CHAN NELS x 7	Scheduler FSM state (ONE-HOT)
sched_error[ch]	output	NUM_CHAN NELS	Scheduler error (sticky)

7.4.7 Shared Descriptor AXI4 Master Read Interface

AR Channel:

Shared Descriptor AXI4 Master Read Interface

Signal	Direction	Width	Description
desc_axi_arvalid	output	1	Address valid
desc_axi_arready	input	1	Address ready
desc_axi_araddr	output	ADDR_WIDTH	Address
desc_axi_arlen	output	8	Burst length - 1
desc_axi_arsize	output	3	Burst size (log2 bytes)
desc_axi_arburst	output	2	Burst type (INCR)
desc_axi_arid	output	AXI_ID_WIDTH	Transaction ID
desc_axi_arlock	output	1	Lock type
desc_axi_arcache	output	4	Cache attributes
desc_axi_arprot	output	3	Protection attributes

Signal	Direction	Width	Description
desc_axi_arqs	output	4	QoS value
desc_axi_arregion	output	4	Region identifier

R Channel (FIXED 256-bit):

Shared Descriptor AXI4 Master Read Interface

Signal	Direction	Width	Description
desc_axi_rvalid	input	1	Read data valid
desc_axi_rready	output	1	Read data ready
desc_axi_rdata	input	256	Read data (FIXED 256-bit descriptor)
desc_axi_rresp	input	2	Response
desc_axi_rlast	input	1	Last beat of burst
desc_axi_rid	input	AXI_ID_WID TH	Transaction ID

7.4.8 Shared Data Read Interface (Per-Channel Arrays)

Shared Data Read Interface

Signal	Direction	Width	Description
sched_rd_val_id[ch]	output	NUM_CHAN NELS	Read request valid
sched_rd_addr[ch]	output	NUM_CHAN NELS x ADDR_WIDTH H	Source address
sched_rd_beats[ch]	output	NUM_CHAN NELS x 32	Beats to read
sched_rd_done_strobe[ch]	input	NUM_CHAN NELS	Read completion strobe
sched_rd_beats_done[ch]	input	NUM_CHAN	Beats completed

Signal	Direction	Width	Description
		NELS x 32	
sched_rd_err or[ch]	input	NUM_CHAN NELS	Read error

7.4.9 Shared Data Write Interface (Per-Channel Arrays)

Shared Data Write Interface

Signal	Direction	Width	Description
sched_wr_val id[ch]	output	NUM_CHAN NELS	Write request valid
sched_wr_rea dy[ch]	input	NUM_CHAN NELS	Write ready
sched_wr_add r[ch]	output	NUM_CHAN NELS x ADDR_WIDT H	Destination address
sched_wr_bea ts[ch]	output	NUM_CHAN NELS x 32	Beats to write
sched_wr_don e_strobe[ch]	input	NUM_CHAN NELS	Write completion strobe
sched_wr_bea ts_done[ch]	input	NUM_CHAN NELS x 32	Beats completed
sched_wr_err or[ch]	input	NUM_CHAN NELS	Write error

7.4.10 Unified Monitor Bus Interface

Unified Monitor Bus Interface

Signal	Direction	Width	Description
mon_valid	output	1	Monitor packet valid
mon_ready	input	1	Monitor packet ready
mon_packet	output	64	Monitor packet data

7.5 Operation

7.5.1 Descriptor Fetch Arbitration

1. **Request Phase:** Channels assert desc_ar_valid[ch] when needing descriptor fetch
2. **Grant Phase:** Round-robin arbiter selects one channel
3. **AR Issue:** Granted channel's AR signals muxed to external interface
4. **ID Embedding:** Channel ID placed in lower bits of arid
5. **Grant ACK:** Arbiter acknowledged when arready received
6. **R Routing:** Response routed to correct channel based on rid[CHAN_WIDTH-1:0]

7.5.2 Data Path Flow

Data interfaces pass directly through without arbitration:

```
scheduler_group[ch].sched_rd_* -> sched_rd_*[ch] -> axi_read_engine  
(handles arbitration)  
scheduler_group[ch].sched_wr_* -> sched_wr_*[ch] -> axi_write_engine  
(handles arbitration)
```

7.5.3 MonBus Aggregation

9 sources aggregated via monbus_arbiter: - Sources 0-7: Per-channel mon_* from scheduler_groups - Source 8: Descriptor AXI master monitor

7.6 Integration Example

```
scheduler_group_array #(  
    .NUM_CHANNELS          (8),  
    .ADDR_WIDTH             (64),  
    .DATA_WIDTH              (512),  
    .AXI_ID_WIDTH            (8)  
) u_scheduler_group_array (  
    .clk                    (clk),  
    .rst_n                  (rst_n),  
  
    // APB kick-off  
    .apb_valid                (apb_valid),  
    .apb_ready                (apb_ready),  
    .apb_addr                 (apb_addr),
```

```

// Configuration
.cfg_channel_enable      (cfg_channel_enable),
.cfg_channel_reset        (cfg_channel_reset),
.cfg_sched_enable          (cfg_sched_enable),
// ... other config

// Descriptor AXI master
.desc_axi_arvalid         (desc_axi_arvalid),
.desc_axi_arready          (desc_axi_arready),
// ... full AR/R interface

// Data read interface (to axi_read_engine)
.sched_rd_valid            (sched_rd_valid),
.sched_rd_addr              (sched_rd_addr),
.sched_rd_beats             (sched_rd_beats),
.sched_rd_done_strobe       (rd_done_strobe),
.sched_rd_beats_done        (rd_beats_done),

// Data write interface (to axi_write_engine)
.sched_wr_valid            (sched_wr_valid),
.sched_wr_ready             (sched_wr_ready),
.sched_wr_addr              (sched_wr_addr),
.sched_wr_beats             (sched_wr_beats),
.sched_wr_done_strobe       (wr_done_strobe),
.sched_wr_beats_done        (wr_beats_done),

// Monitor bus
.mon_valid                 (sga_mon_valid),
.mon_ready                  (sga_mon_ready),
.mon_packet                 (sga_mon_packet)
);

```

7.7 Related Documentation

- **Parent:** 01_stream_core.md - Top-level integration
- **Child:** 03_scheduler_group.md - Single channel scheduler + descriptor engine
- **Scheduler:** 04_scheduler.md - Channel state machine
- **Descriptor Engine:** 05_descriptor_engine.md - Descriptor fetch and parsing

- **AXI Read Engine:** 06_axi_read_engine.md - Read datapath (receives sched_rd_*)
 - **AXI Write Engine:** 12_axi_write_engine.md - Write datapath (receives sched_wr_*)
 - **MonBus Arbiter:** 16_monbus_axil_group.md - Monitor bus aggregation
-

7.8 Revision History

Scheduler Group Array Revision History

Version	Date	Author	Description
0.90	2025-11-22	seang	Initial block specification
0.91	2026-01-02	seang	Added table captions and figure numbers

Last Updated: 2026-01-02

8 Scheduler Group

Module: scheduler_group.sv **Location:** projects/components/stream/rtl/macro/

Category: MACRO (Channel Wrapper) **Parent:** scheduler_group_array.sv **Status:**

Implemented **Last Updated:** 2025-11-30

8.1 Overview

The scheduler_group module is a wrapper combining the scheduler and descriptor_engine for a single STREAM channel. It provides a simplified interface compared to RAPIDS by removing unused features while maintaining the core descriptor-based DMA functionality.

8.1.1 Key Features

- **Single Channel Wrapper:** Combines scheduler + descriptor_engine
- **Simplified from RAPIDS:** No program engine, no control engines, no alignment bus

- **MonBus Aggregation:** Combines 2 internal sources (scheduler, descriptor_engine)
- **Direct Data Interfaces:** Per-channel read/write interfaces to shared engines

8.1.2 Simplified from RAPIDS

Removed from RAPIDS scheduler_group: - Program engine - Control read/write engines - Alignment bus - Network interfaces (RDA, EOS completion) - Credit management

8.2 Architecture

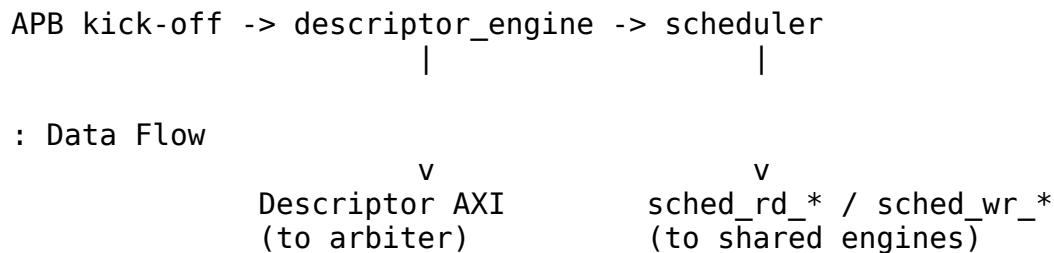
8.2.1 Component Hierarchy

```

scheduler_group
  scheduler          # Channel state machine
  descriptor_engine # Descriptor fetch and parsing
  arbiter_round_robin_monbus # 2-source MonBus arbiter

```

8.2.2 Data Flow



Descriptor Flow: 1. APB kick-off provides descriptor address 2. Descriptor engine fetches via shared AXI (arbitrated externally) 3. Parsed descriptor delivered to scheduler 4. Scheduler coordinates read/write phases

Data Path Flow: 1. Scheduler asserts sched_rd_valid with source address/beats 2. External read engine issues AXI reads, returns completion strobe 3. Scheduler asserts sched_wr_valid with destination address/beats 4. External write engine issues AXI writes, returns completion strobe 5. Scheduler advances to next descriptor or completes

8.3 Parameters

Parameters

Parameter	Type	Default	Description
CHANNEL_ID	int	0	Channel identifier (0-7)
NUM_CHANNELS	int	8	Total number of channels
CHAN_WIDTH	int	\$clog2(NUM_CHANNELS)	Channel ID width
ADDR_WIDTH	int	64	Address bus width
DATA_WIDTH	int	512	Data bus width
AXI_ID_WIDTH	int	8	AXI transaction ID width

8.3.1 Monitor Bus Agent IDs

Monitor Bus Agent IDs

Parameter	Type	Default	Description
DESC_MON_AGEN_T_ID	8-bit	0x10+ch	Descriptor engine agent ID
SCHED_MON_AGE_NT_ID	8-bit	0x30+ch	Scheduler agent ID
MON_UNIT_ID	4-bit	0x1	Unit ID
MON_CHANNEL_ID	6-bit	ch	Channel ID in packets

8.4 Port List

8.4.1 Clock and Reset

Clock and Reset

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

8.4.2 APB Programming Interface

APB Programming Interface

Signal	Direction	Width	Description
apb_valid	input	1	Descriptor address valid
apb_ready	output	1	Ready to accept descriptor
apb_addr	input	ADDR_WIDTH	Descriptor address

8.4.3 Configuration Interface

Configuration Interface

Signal	Direction	Width	Description
cfg_channel_enable	input	1	Enable this channel
cfg_channel_reset	input	1	Soft reset this channel

8.4.4 Scheduler Configuration

Scheduler Configuration

Signal	Direction	Width	Description
cfg_sched_timeout	input	16	Timeout threshold

Signal	Direction	Width	Description
cfg_sched_tim_eout_enable	input	1	Enable timeout detection
cfg_sched_err_enable	input	1	Enable error reporting
cfg_sched_compl_enable	input	1	Enable completion reporting
cfg_sched_perf_enable	input	1	Enable performance monitoring

8.4.5 Descriptor Engine Configuration

Descriptor Engine Configuration

Signal	Direction	Width	Description
cfg_desceng_prefetch	input	1	Enable descriptor prefetch
cfg_desceng_fifo_thresh	input	4	FIFO threshold for prefetch
cfg_desceng_addr0_base	input	ADDR_WIDT_H	Address range 0 base
cfg_desceng_addr0_limit	input	ADDR_WIDT_H	Address range 0 limit
cfg_desceng_addr1_base	input	ADDR_WIDT_H	Address range 1 base
cfg_desceng_addr1_limit	input	ADDR_WIDT_H	Address range 1 limit

8.4.6 Status Interface

Status Interface

Signal	Direction	Width	Description
descriptor_engine_idle	output	1	Descriptor engine idle
scheduler_idle	output	1	Scheduler idle

Signal	Direction	Width	Description
scheduler_state	output	7	IDLE state Scheduler FSM state (ONE-HOT)
sched_error	output	1	Scheduler error (sticky)

8.4.7 Descriptor AXI Interface (to external arbiter)

AR Channel:

Descriptor AXI Interface

Signal	Direction	Width	Description
desc_ar_valid	output	1	Address valid
desc_ar_ready	input	1	Address ready
desc_ar_addr	output	ADDR_WIDTH	Address
desc_ar_len	output	8	Burst length - 1
desc_ar_size	output	3	Burst size
desc_ar_burst	output	2	Burst type
desc_ar_id	output	AXI_ID_WIDTH	Transaction ID
desc_ar_lock	output	1	Lock type
desc_ar_cache	output	4	Cache attributes
desc_ar_prot	output	3	Protection attributes
desc_ar_qos	output	4	QoS value
desc_ar_region	output	4	Region identifier

R Channel (256-bit fixed):

Descriptor AXI Interface

Signal	Direction	Width	Description
desc_r_valid	input	1	Read data valid

Signal	Direction	Width	Description
desc_r_ready	output	1	Read data ready
desc_r_data	input	256	Read data (256-bit descriptor)
desc_r_resp	input	2	Response
desc_r_last	input	1	Last beat
desc_r_id	input	AXI_ID_WIDTH	Transaction ID

8.4.8 Data Read Interface (to shared read engine)

Data Read Interface

Signal	Direction	Width	Description
sched_rd_vali d	output	1	Read request valid
sched_rd_addr	output	ADDR_WIDTH	Source address
sched_rd_beat s	output	32	Beats to read
sched_rd_done _strobe	input	1	Read completion strobe
sched_rd_beat s_done	input	32	Beats completed
sched_rd_error	input	1	Read error

8.4.9 Data Write Interface (to shared write engine)

Data Write Interface

Signal	Direction	Width	Description
sched_wr_vali d	output	1	Write request valid
sched_wr_ready	input	1	Write ready
sched_wr_addr	output	ADDR_WIDTH	Destination address
sched_wr_beat s	output	32	Beats to write

Signal	Direction	Width	Description
sched_wr_done_strobe	input	1	Write completion strobe
sched_wr_beat_s_done	input	32	Beats completed
sched_wr_error	input	1	Write error

8.4.10 Monitor Bus Interface

Monitor Bus Interface

Signal	Direction	Width	Description
mon_valid	output	1	Monitor packet valid
mon_ready	input	1	Monitor packet ready
mon_packet	output	64	Monitor packet data

8.5 Operation

8.5.1 Transfer Sequence

1. **APB Kick-off:** apb_valid with descriptor address
2. **Descriptor Fetch:** descriptor_engine fetches via AXI
3. **Descriptor Parse:** 256-bit descriptor parsed into fields
4. **Read Phase:** scheduler asserts sched_rd_valid
5. **Read Completion:** engine returns sched_rd_done_strobe
6. **Write Phase:** scheduler asserts sched_wr_valid
7. **Write Completion:** engine returns sched_wr_done_strobe
8. **Chain Check:** if next_descriptor_ptr != 0, fetch next
9. **Complete:** scheduler returns to IDLE

8.5.2 MonBus Sources

The scheduler_group aggregates 2 MonBus sources: 1. **Descriptor Engine:** Fetch events, errors 2. **Scheduler:** State transitions, completions, errors

8.6 Integration Example

```
scheduler_group #(
    .CHANNEL_ID          (ch),
    .NUM_CHANNELS        (8),
    .ADDR_WIDTH          (64),
    .DATA_WIDTH          (512),
    .AXI_ID_WIDTH        (8),
    .DESC_MON_AGENT_ID   (8'h10 + ch),
    .SCHED_MON_AGENT_ID  (8'h30 + ch)
) u_scheduler_group (
    .clk                 (clk),
    .rst_n               (rst_n),

    // APB interface
    .apb_valid           (apb_valid[ch]),
    .apb_ready            (apb_ready[ch]),
    .apb_addr             (apb_addr[ch]),

    // Configuration
    .cfg_channel_enable   (cfg_channel_enable[ch]),
    .cfg_channel_reset    (cfg_channel_reset[ch]),
    // ... other config

    // Descriptor AXI (to arbiter)
    .desc_ar_valid        (desc_ar_valid[ch]),
    .desc_ar_ready         (desc_ar_ready[ch]),
    // ... full AR/R interface

    // Data interfaces (direct passthrough)
    .sched_rd_valid       (sched_rd_valid[ch]),
    .sched_rd_addr         (sched_rd_addr[ch]),
    .sched_rd_beats        (sched_rd_beats[ch]),
    .sched_rd_done_strobe  (sched_rd_done_strobe[ch]),
    .sched_rd_beats_done   (sched_rd_beats_done[ch]),

    .sched_wr_valid       (sched_wr_valid[ch]),
    .sched_wr_ready        (sched_wr_ready[ch]),
    .sched_wr_addr         (sched_wr_addr[ch]),
    .sched_wr_beats        (sched_wr_beats[ch]),
    .sched_wr_done_strobe  (sched_wr_done_strobe[ch]),
    .sched_wr_beats_done   (sched_wr_beats_done[ch]),

    // Monitor bus
```

```
.mon_valid          (mon_valid_ch[ch]),  
.mon_ready         (mon_ready_ch[ch]),  
.mon_packet        (mon_packet_ch[ch])  
);
```

8.7 Related Documentation

- **Parent:** 02_scheduler_group_array.md - Multi-channel array
 - **Scheduler:** 04_scheduler.md - Channel state machine details
 - **Descriptor Engine:** 05_descriptor_engine.md - Descriptor fetch logic
 - **AXI Read Engine:** 06_axi_read_engine.md - Shared read engine
 - **AXI Write Engine:** 12_axi_write_engine.md - Shared write engine
-

8.8 Revision History

Scheduler Group Revision History

Version	Date	Author	Description
0.90	2025-11-22	seang	Initial block specification
0.91	2026-01-02	seang	Added table captions and figure numbers

Last Updated: 2026-01-02

9 Scheduler Specification

Module: scheduler.sv **Location:** projects/components/stream/rtl/fub/ **Status:** Implemented **Last Updated:** 2025-11-30

9.1 Overview

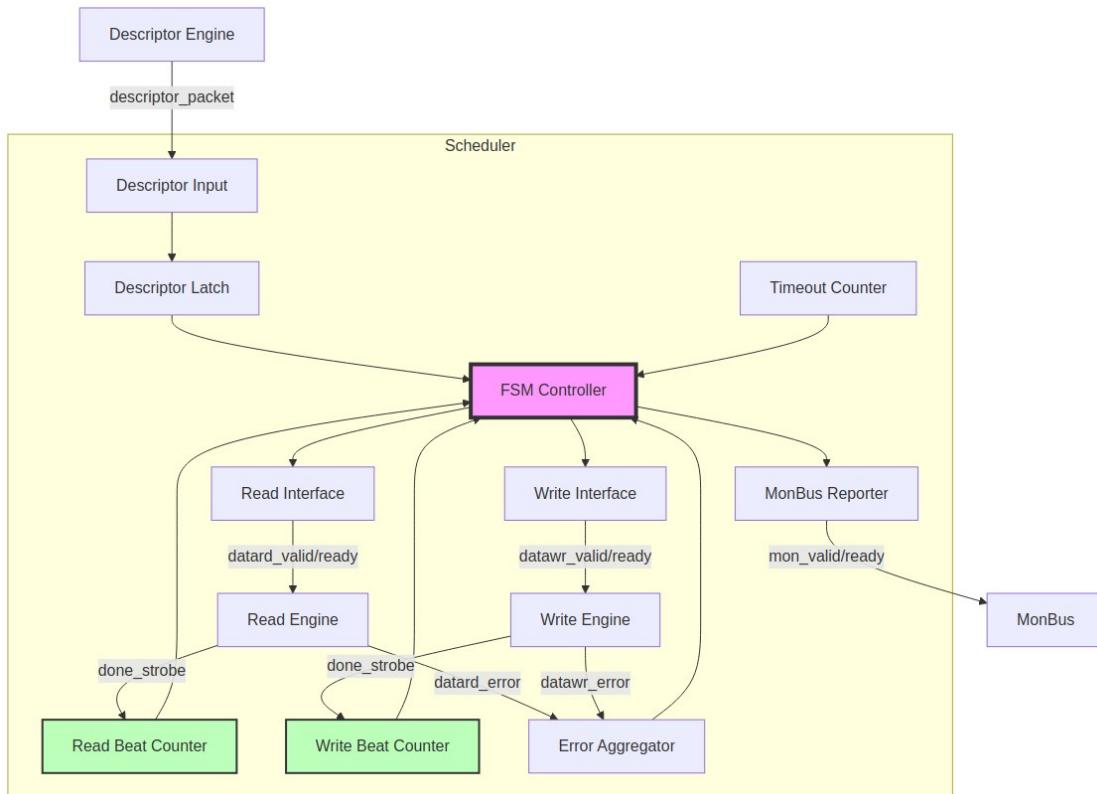
The Scheduler coordinates descriptor-based memory-to-memory DMA transfers for a single channel. It receives descriptors, manages concurrent read/write operations, and handles descriptor chaining.

9.1.1 Key Features

- **Concurrent read/write:** Read and write engines run simultaneously (prevents deadlock)
- **Beat-based tracking:** Length in data width units (STREAM simplification)
- **Aligned addresses:** No alignment fixup logic (must be pre-aligned)
- **Descriptor chaining:** Follows next_descriptor_ptr for multi-buffer transfers
- **Interrupt generation:** MonBus IRQ event when gen_irq flag set
- **Error handling:** Timeout detection, error aggregation from engines
- **MonBus integration:** State transition and IRQ event reporting

9.1.2 Block Diagram

9.1.3 Figure 2.4.1: Scheduler Block Diagram



Diagram

Source: [02_scheduler_block.mmd](#)

9.2 CRITICAL: Concurrent Read/Write Design

Why Concurrent Operation is Essential:

The scheduler runs read and write engines **CONCURRENTLY** in CH_XFER_DATA state. This prevents deadlock when transfer size exceeds SRAM buffer capacity:

Example: 100MB transfer with 2KB SRAM buffer

Sequential operation (WRONG):

1. Read 100MB → DEADLOCK at 2KB (SRAM full, can't complete read)

Concurrent operation (CORRECT):

1. Read starts filling SRAM → SRAM becomes full (2KB)
2. Read pauses (natural backpressure)
3. Write drains SRAM → SRAM has free space
4. Read resumes → Both continue until 100MB complete

Implementation: - Both sched_rd_valid and sched_wr_valid asserted in CH_XFER_DATA - Independent beat counters: r_read_beats_remaining, r_write_beats_remaining - Exit when **BOTH** counters reach zero

9.3 Parameters

```

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

// Monitor Bus Parameters
parameter logic [7:0] MON_AGENT_ID = 8'h40;          // STREAM Scheduler
Agent ID
parameter logic [3:0] MON_UNIT_ID = 4'h1;              // Unit identifier
parameter logic [5:0] MON_CHANNEL_ID = 6'h0;           // Base channel ID

// Descriptor Width (FIXED at 256-bit for STREAM)
parameter int DESC_WIDTH = 256;

```

Validation:

```

// Scheduler only supports 256-bit STREAM descriptors
if (DESC_WIDTH != 256)
    $fatal("DESC_WIDTH must be 256 for STREAM scheduler");

```

9.4 Port List

9.4.1 Clock and Reset

Clock and Reset

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

Signal	Direction	Width	Description
			asynchronous reset

9.4.2 Configuration Interface

Configuration Interface

Signal	Direction	Width	Description
cfg_channel_enable	input	1	Enable this channel
cfg_channel_reset	input	1	Channel soft reset (FSM → IDLE)
cfg_sched_timeout_cycles	input	16	Timeout threshold in clock cycles (runtime config)
cfg_sched_timeout_enable	input	1	Enable timeout detection

9.4.3 Status Interface

Status Interface

Signal	Direction	Width	Description
scheduler_idle	output	1	Scheduler idle flag
scheduler_state	output	7	Current FSM state (one-hot encoding)

9.4.4 Descriptor Engine Interface

Descriptor Engine Interface

Signal	Direction	Width	Description
descriptor_valid	input	1	Descriptor valid from descriptor engine
descriptor_ready	output	1	Scheduler ready to accept descriptor
descriptor_packet	input	256	256-bit STREAM descriptor (see format below)
descriptor_error	input	1	Error signal from

Signal	Direction	Width	Description
rror			descriptor engine

9.4.5 Data Read Interface

To AXI Read Engine:

Data Read Interface

Signal	Direction	Width	Description
sched_rd_val_id	output	1	Channel requests read
sched_rd_addr	output	ADDR_WIDTH_H	Source address (aligned, static during burst)
sched_rd_beats	output	32	Beats remaining to read

Completion from Read Engine:

Data Read Interface

Signal	Direction	Width	Description
sched_rd_done_strobe	input	1	Read burst completed (1-cycle pulse)
sched_rd_beats_done	input	32	Number of beats completed in burst
sched_rd_error	input	1	Read engine error (sticky)

9.4.6 Data Write Interface

To AXI Write Engine:

Data Write Interface

Signal	Direction	Width	Description
sched_wr_val_id	output	1	Channel requests write
sched_wr_ready	input	1	Engine ready for channel (completion handshake)
sched_wr_addr	output	ADDR_WIDTH_H	Destination address (aligned, static during

Signal	Direction	Width	Description
sched_wr_beats	output	32	Beats remaining to write burst)

Completion from Write Engine:

Data Write Interface

Signal	Direction	Width	Description
sched_wr_don_e_strobe	input	1	Write burst completed (1-cycle pulse)
sched_wr_beats_done	input	32	Number of beats completed in burst
sched_wr_err or	input	1	Write engine error (sticky)

9.4.7 Error Signals

Error Signals

Signal	Direction	Width	Description
sched_error	output	1	Scheduler error output (aggregates rd/wr errors, sticky)

9.4.8 Monitor Bus Interface

Monitor Bus Interface

Signal	Direction	Width	Description
mon_valid	output	1	Monitor packet valid
mon_ready	input	1	Monitor bus ready
mon_packet	output	64	64-bit monitor bus packet

9.5 Interface

9.5.1 Clock and Reset

```
input logic clk;
input logic rst_n; // Active-low
asynchronous reset
```

9.5.2 Configuration Interface

```
input logic cfg_channel_enable; //  
Enable this channel  
input logic cfg_channel_reset; //  
Channel reset (soft reset)  
input logic [15:0] cfg_sched_timeout_cycles; //  
Timeout threshold (runtime config)  
input logic cfg_sched_timeout_enable; //  
Enable timeout detection
```

Channel Reset Behavior: - cfg_channel_reset forces FSM to CH_IDLE immediately - Clears descriptor_loaded flag - Resets beat counters - Independent of global rst_n

Timeout Configuration: - Runtime configurable via cfg_sched_timeout_cycles (replaces compile-time TIMEOUT_CYCLES parameter) - Can be disabled/enabled dynamically via cfg_sched_timeout_enable

9.5.3 Status Interface

```
output logic scheduler_idle; //  
Scheduler in CH_IDLE  
output logic [6:0] scheduler_state; // Current  
state (ONE-HOT)
```

State Encoding (ONE-HOT): - [0] = CH_IDLE - [1] = CH_FETCH_DESC - [2] = CH_XFER_DATA - [3] = CH_COMPLETE - [4] = CH_NEXT_DESC - [5] = CH_ERROR - [6] = Reserved

9.5.4 Descriptor Engine Interface

```
input logic descriptor_valid;
output logic descriptor_ready;
input logic [DESC_WIDTH-1:0] descriptor_packet; // 256-bit
STREAM descriptor
input logic descriptor_error; // Error
from descriptor engine
```

Descriptor Handshake: - descriptor_ready asserted in CH_IDLE or CH_NEXT_DESC - Descriptor captured when valid && ready - Supports descriptor chaining (next_descriptor_ptr)

9.5.5 Data Read Interface (to AXI Read Engine)

Request:

```

output logic                               sched_rd_valid;           // Request
read access
output logic [ADDR_WIDTH-1:0]            sched_rd_addr;           // Source
address (static base)
output logic [31:0]                      sched_rd_beats;          // Total
beats to read

```

Completion:

```

input logic                               sched_rd_done_strobe;    // Read
engine completed beats
input logic [31:0]                      sched_rd_beats_done;     // Number
of beats completed

```

Error:

```

input logic                               sched_rd_error;           // Read
engine error

```

Address Management: - Scheduler provides **static base address** in `sched_rd_addr` - Read engine handles address increment internally - Scheduler does NOT update `sched_rd_addr` after each burst

Interface Timing Notes (November 2025 Updates): - Scheduler holds `sched_rd_valid` high while `sched_rd_beats > 0` - Engine reports completion via `sched_rd_done_strobe` (pulsed) - Scheduler decrements `r_read_beats_remaining` by `sched_rd_beats_done` - Process repeats until `r_read_beats_remaining == 0`

9.5.6 Data Write Interface (to AXI Write Engine)

Request:

```

output logic                               sched_wr_valid;           // Request
write access
input logic                               sched_wr_ready;           // Engine
grants access
output logic [ADDR_WIDTH-1:0]            sched_wr_addr;           // 
Destination address (static base)
output logic [31:0]                      sched_wr_beats;          // Total
beats to write

```

Completion:

```

input logic                               sched_wr_done_strobe;    // Write
engine completed beats
input logic [31:0]                      sched_wr_beats_done;     // Number
of beats completed

```

Error:

```

 sched_wr_error; // Write engine error

```

Interface Timing Notes (November 2025 Updates):

sched_wr_ready Timing: - REGISTERED OUTPUT from write engine - Not combinatorial from completion signals - Asserts 1 cycle after channel becomes ready to accept new request - Cleared 1 cycle after handshake (valid && ready) - **Do not expect immediate deassertion on handshake** - takes 1 cycle

Example Timing:

Cycle	sched_wr_valid	sched_wr_ready	sched_wr_done_strobe	Notes
0	0	0	0	Idle
1	1	0	0	Request (not ready yet)
2	1	1	0	Handshake!
3	1	0	0	Ready cleared (registered) ... (W-phase executes)
50	1	0	1	B response, done strobe
51	1	1	0	Ready re- asserts (next cycle)

9.5.7 Monitor Bus Interface

```

output logic mon_valid;
input logic mon_ready;
output logic [63:0] mon_packet;

```

MonBus Events Generated: - State transitions (IDLE → FETCH_DESC, etc.) - IRQ event (when descriptor.gen_irq set) - Error events

9.6 Descriptor Format

9.6.1 STREAM Descriptor (256-bit)

```

typedef struct packed {
    logic [47:0] reserved;           // [255:208] Reserved
    logic [7:0] desc_priority;       // [207:200] Transfer
priority
    logic [3:0] channel_id;          // [199:196] Channel ID
(informational)
    logic error;                   // [195] Error flag
    logic last;                     // [194] Last in chain flag

```

```

    logic      gen_irq;           // [193] Generate interrupt
on completion
    logic      valid;            // [192] Valid descriptor
    logic [31:0] next_descriptor_ptr; // [191:160] Next descriptor
address (0 = last)
    logic [31:0] length;        // [159:128] Length in BEATS
    logic [63:0] dst_addr;      // [127:64] Destination
address (aligned)
    logic [63:0] src_addr;      // [63:0] Source address
(aligned)
} descriptor_t;

```

Field Constraints: - src_addr / dst_addr: Must be aligned to DATA_WIDTH (e.g., 64-byte aligned for 512-bit data) - length: Transfer size in **BEATS** (not bytes or chunks) - next_descriptor_ptr: 0 or address of next descriptor - valid: Must be 1 for descriptor to be accepted - last: Terminates chain (overrides next_descriptor_ptr) - gen_irq: Generates STREAM_EVENT_IRQ via MonBus when transfer completes

Descriptor Bit Positions:

DESC_SRC_ADDR:	[63:0]
DESC_DST_ADDR:	[127:64]
DESC_LENGTH:	[159:128]
DESC_NEXT_PTR:	[191:160]
DESC_VALID_BIT:	[192]
DESC_GEN_IRQ:	[193]
DESC_LAST:	[194]

9.7 FSM Operation

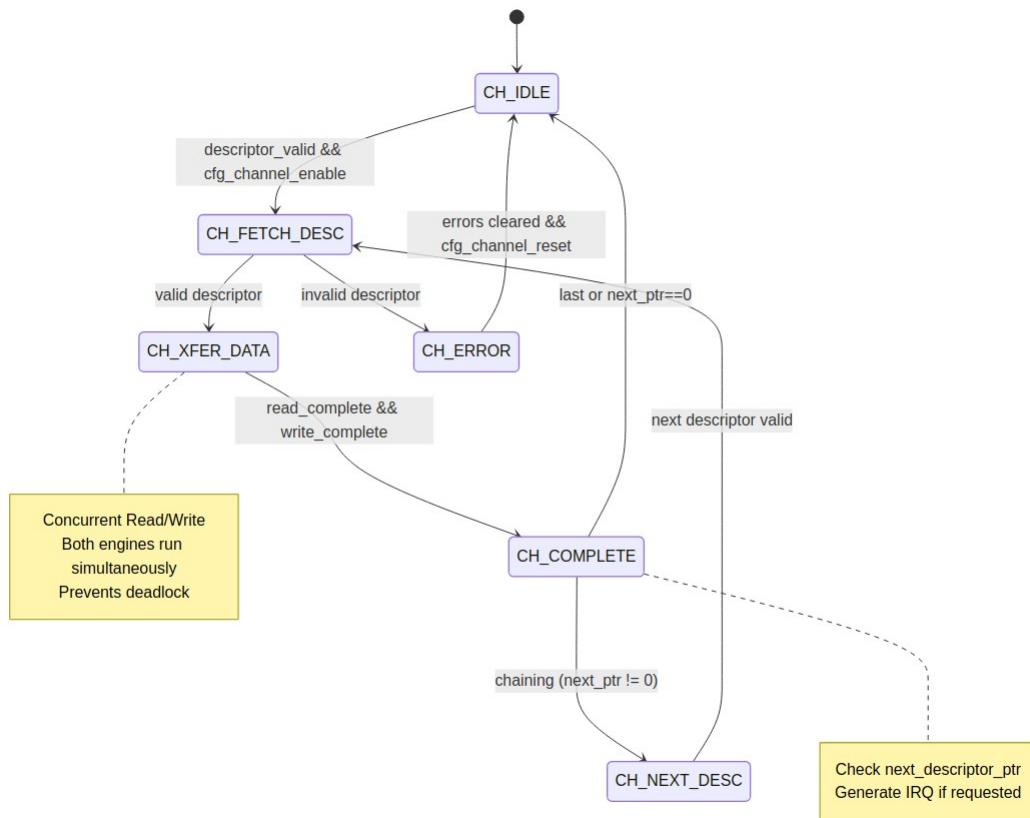
9.7.1 State Machine

States (ONE-HOT encoded):

CH_IDLE	- Waiting for descriptor
CH_FETCH_DESC	- Latch and validate descriptor
CH_XFER_DATA	- Concurrent read/write transfer
CH_COMPLETE	- Transfer done, check chaining
CH_NEXT_DESC	- Fetch next chained descriptor
CH_ERROR	- Error condition

FSM Flow:

9.7.2 Figure 2.4.2: Scheduler FSM



Diagram

Source: [02_scheduler_block.mmd](#)

9.7.3 State Transitions

CH_IDLE: - Wait for: `descriptor_valid && cfg_channel_enable` - Action: Assert `descriptor_ready` - Next: **CH_FETCH_DESC** (when handshake occurs)

CH_FETCH_DESC: - Action: - Latch descriptor fields into `r_descriptor` - Initialize working registers (`r_src_addr`, `r_dst_addr`, `r_*_beats_remaining`) - Validate `descriptor.valid` bit - Next: - **CH_XFER_DATA** (if valid) - **CH_ERROR** (if invalid)

CH_XFER_DATA: - Action: - Assert **BOTH** `sched_rd_valid` and `sched_wr_valid` (concurrent operation!) - Decrement `r_read_beats_remaining` on `sched_rd_done_strobe` - Decrement `r_write_beats_remaining` on `sched_wr_done_strobe` - Monitor timeout counter - Exit When: `r_read_beats_remaining == 0 && r_write_beats_remaining == 0` - Next: **CH_COMPLETE**

CH_COMPLETE: - Action: - Generate MonBus IRQ event (if `gen_irq` set) - Check `next_descriptor_ptr` and `last` flag - Clear `descriptor_loaded` flag - Next: - **CH_NEXT_DESC** (if chaining) - **CH_IDLE** (if last or no chain)

CH_NEXT_DESC: - **Wait for:** descriptor_valid (descriptor engine fetches next) - **Action:** Assert descriptor_ready - **Next:** CH_FETCH_DESC

CH_ERROR: - **STICKY STATE:** Once in error, stays here until channel reset - **Action:** Report error via MonBus - **Recovery:** Requires cfg_channel_reset assertion (or global reset) - **Note:** scheduler_idle asserts in this state (allows external monitoring)

9.8 Beat Tracking

9.8.1 Independent Counters

Initialization (CH_FETCH_DESC):

```
r_read_beats_remaining <= r_descriptor.length;  
r_write_beats_remaining <= r_descriptor.length;
```

Decrement (CH_XFER_DATA):

```
// Read progress (independent)  
if (sched_rd_done_strobe) begin  
    r_read_beats_remaining <= (r_read_beats_remaining >=  
    sched_rd_beats_done) ?  
        (r_read_beats_remaining -  
    sched_rd_beats_done) : 32'h0;  
end  
  
// Write progress (independent)  
if (sched_wr_done_strobe) begin  
    r_write_beats_remaining <= (r_write_beats_remaining >=  
    sched_wr_beats_done) ?  
        (r_write_beats_remaining -  
    sched_wr_beats_done) : 32'h0;  
end
```

Saturation: - Counters saturate at 0 (prevent underflow) - Safety check for engine misbehavior

9.8.2 Completion Detection

Combinational Flags:

```
w_read_complete      = (r_read_beats_remaining == 0);  
w_write_complete     = (r_write_beats_remaining == 0);  
w_transfer_complete = w_read_complete && w_write_complete;
```

State Exit:

```
// In CH_XFER_DATA:
if (w_transfer_complete) begin
    w_next_state = CH_COMPLETE;
end
```

9.8.3 Multiple Requests per Descriptor

For large transfers, scheduler issues multiple requests as engines complete work:

Descriptor: length = 256 beats (16KB @ 512-bit data)
 Engine burst size: 16 beats (configured via cfg_axi_wr_xfer_beats)

Request sequence:

Request 1: sched_wr_valid=1, sched_wr_beats=256, ready → handshake
 Engine executes 16-beat burst
 sched_wr_done_strobe=1, sched_wr_beats_done=16
 Scheduler updates: 256 - 16 = 240 beats remaining

Request 2: sched_wr_valid=1, sched_wr_beats=240, ready → handshake
 Engine executes 16-beat burst
 sched_wr_done_strobe=1, sched_wr_beats_done=16
 Scheduler updates: 240 - 16 = 224 beats remaining

... (continues for 16 requests total)

Request 16: sched_wr_valid=1, sched_wr_beats=16, ready → handshake
 Engine executes 16-beat burst
 sched_wr_done_strobe=1, sched_wr_beats_done=16
 Scheduler updates: 16 - 16 = 0 beats remaining →

COMPLETE

Key Point: Scheduler holds sched_wr_valid high and keeps reissuing requests as long as sched_wr_beats > 0. Engines handle each request independently.

9.9 Address Management

9.9.1 Static Base Address

Scheduler Provides:

```
assign sched_rd_addr = r_src_addr; // Static base, set in
CH_FETCH_DESC
assign sched_wr_addr = r_dst_addr; // Static base, set in
CH_FETCH_DESC
```

Scheduler Does NOT: - Increment addresses after each burst - Calculate byte offsets - Handle alignment

Engine Responsibility: - Read engine: `m_axi_araddr = sched_rd_addr + (beats_issued << AXSIZE)` - Write engine: `m_axi_awaddr = sched_wr_addr + (beats_issued << AXSIZE)`

9.10 Timeout Detection

9.10.1 Timeout Counter

Configuration: - Runtime configurable via `cfg_sched_timeout_cycles` (16-bit) - Can be enabled/disabled via `cfg_sched_timeout_enable` - Replaces compile-time `TIMOUT_CYCLES` parameter

Increment:

```
// In CH_XFER_DATA when waiting for engines
if (cfg_sched_timeout_enable && (!sched_rd_ready || !sched_wr_ready))
begin
    r_timeout_counter <= r_timeout_counter + 1;
end
```

Timeout Flag:

```
assign w_timeout_expired = (r_timeout_counter >=
cfg_sched_timeout_cycles);
```

Reset:

```
// Clear when state changes or engines respond
if (state_change || (sched_rd_ready && sched_wr_ready)) begin
    r_timeout_counter <= 0;
end
```

Action on Timeout: - FSM transitions to `CH_ERROR` - MonBus timeout event generated - Channel must be reset to recover

9.11 Error Handling

9.11.1 Error Sources

External: - `descriptor_error` - Descriptor engine reports error - `sched_rd_error` - Read engine error (AXI RRESP != OKAY, etc.) - `sched_wr_error` - Write engine error (AXI BRESP != OKAY, etc.)

Internal: - `w_timeout_expired` - Timeout counter exceeded threshold - !`r_descriptor.valid` - Invalid descriptor in CH_FETCH_DESC

9.11.2 Sticky Error Flags

```
logic r_read_error_sticky; // Set on sched_rd_error, cleared in  
CH_IDLE  
logic r_write_error_sticky; // Set on sched_wr_error, cleared in  
CH_IDLE  
logic r_descriptor_error; // Set on descriptor_error or validation  
failure
```

Set Condition:

```
if (sched_rd_error)  
    r_read_error_sticky <= 1'b1;  
  
if (sched_wr_error)  
    r_write_error_sticky <= 1'b1;
```

Clear Condition:

```
if (r_current_state == CH_IDLE)  
    r_*_error_sticky <= 1'b0;
```

9.11.3 Error Recovery

Error Transition:

```
// Any state with error condition  
if (descriptor_error || sched_rd_error || sched_wr_error ||  
    r_read_error_sticky || r_write_error_sticky || w_timeout_expired)  
begin  
    w_next_state = CH_ERROR;  
end
```

CH_ERROR is STICKY:

```
CH_ERROR: begin  
    // Error state - STICKY, stay here until reset  
    // Once in error, only way out is through reset  
    w_next_state = CH_ERROR;  
end
```

Recovery: - CH_ERROR is a **sticky state** - does NOT auto-recover - Software **must** assert `cfg_channel_reset` (or global `rst_n`) - On channel reset: FSM → CH_IDLE, sticky flags cleared

9.12 Interrupt Generation

9.12.1 IRQ via MonBus

Trigger: - Descriptor completes (CH_COMPLETE state) - r_descriptor.gen_irq flag set

MonBus Event:

```
// In CH_COMPLETE state with gen_irq set
mon_packet = {
    MON_AGENT_ID,           // [63:56] Agent ID (0x40 = STREAM
Scheduler)
    MON_UNIT_ID,            // [55:52] Unit ID
    MON_CHANNEL_ID,          // [51:46] Channel ID
    STREAM_EVENT_IRQ,        // [45:40] Event code (IRQ)
    descriptor_fields        // [39:0] Descriptor info
};
```

No Separate IRQ Signal: - IRQ communicated via MonBus only - Software monitors MonBus for IRQ events - Event includes channel ID for routing

9.13 Descriptor Chaining

9.13.1 Chain Detection

In CH_COMPLETE:

```
if (r_descriptor.next_descriptor_ptr != 32'h0 && !r_descriptor.last)
begin
    w_next_state = CH_NEXT_DESC; // Chain to next descriptor
end else begin
    w_next_state = CH_IDLE;     // Complete (last or no chain)
end
```

9.13.2 Chain Termination

Explicit Termination: - next_descriptor_ptr == 0 → Stop - last == 1 → Stop (overrides next_descriptor_ptr)

Example Chain:

Descriptor 0 @ 0x1000:
src_addr = 0x2000, dst_addr = 0x3000, length = 64
next_descriptor_ptr = 0x1040, last = 0
→ Chains to next

Descriptor 1 @ 0x1040:

```
src_addr = 0x2100, dst_addr = 0x3100, length = 32  
next_descriptor_ptr = 0x0000, last = 1  
→ Last in chain
```

9.14 MonBus Integration

9.14.1 Event Types

State Transitions: - IDLE → FETCH_DESC: Descriptor fetch start - FETCH_DESC → XFER_DATA: Transfer start - XFER_DATA → COMPLETE: Transfer complete - COMPLETE → NEXT_DESC: Chain fetch - Any → ERROR: Error occurred

Special Events: - STREAM_EVENT_IRQ: Interrupt generation (gen_irq flag) - STREAM_EVENT_TIMEOUT: Timeout expired - STREAM_EVENT_ERROR: Error condition

9.14.2 MonBus Packet Format

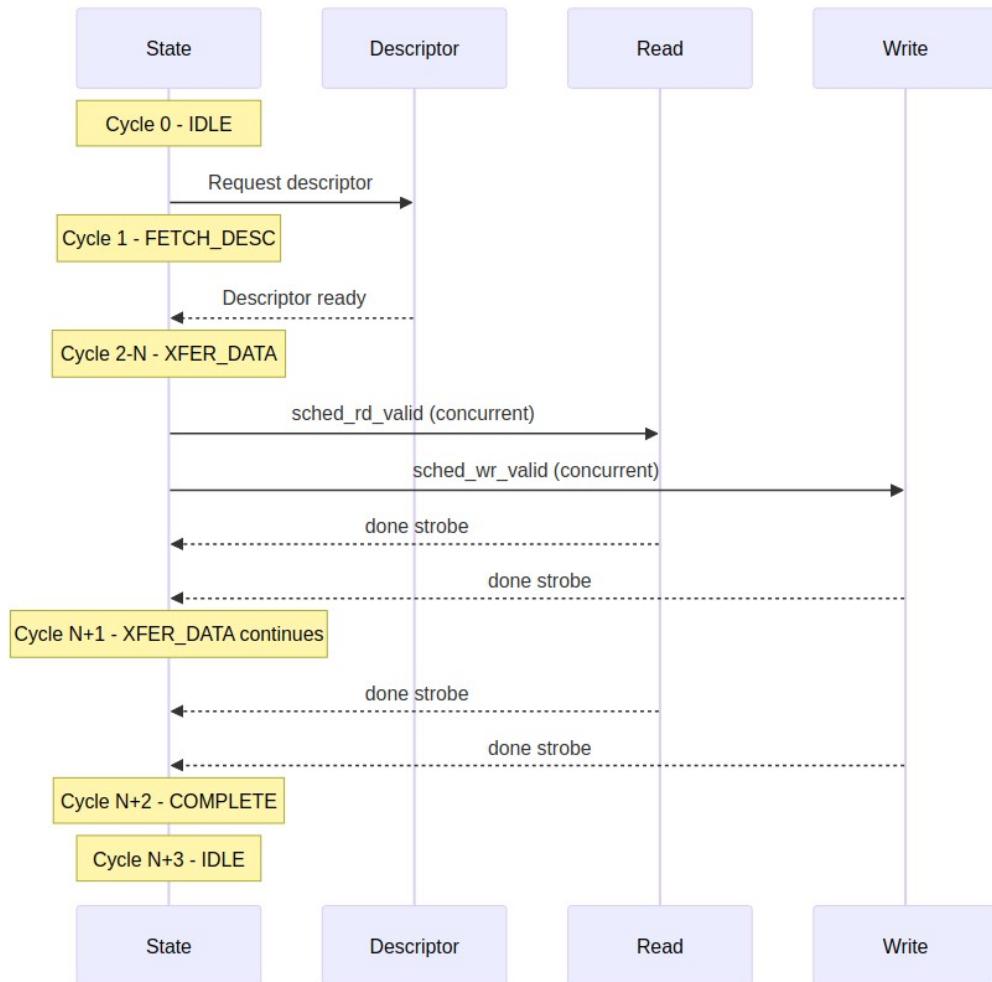
Generic Event:

[63:56]	- MON_AGENT_ID (0x40)
[55:52]	- MON_UNIT_ID
[51:46]	- MON_CHANNEL_ID + CHANNEL_ID
[45:40]	- Event code
[39:0]	- Event-specific data

9.15 Timing Diagrams

9.15.1 Normal Transfer (No Chaining)

9.15.2 Figure 2.4.3: Scheduler Normal Transfer Timing



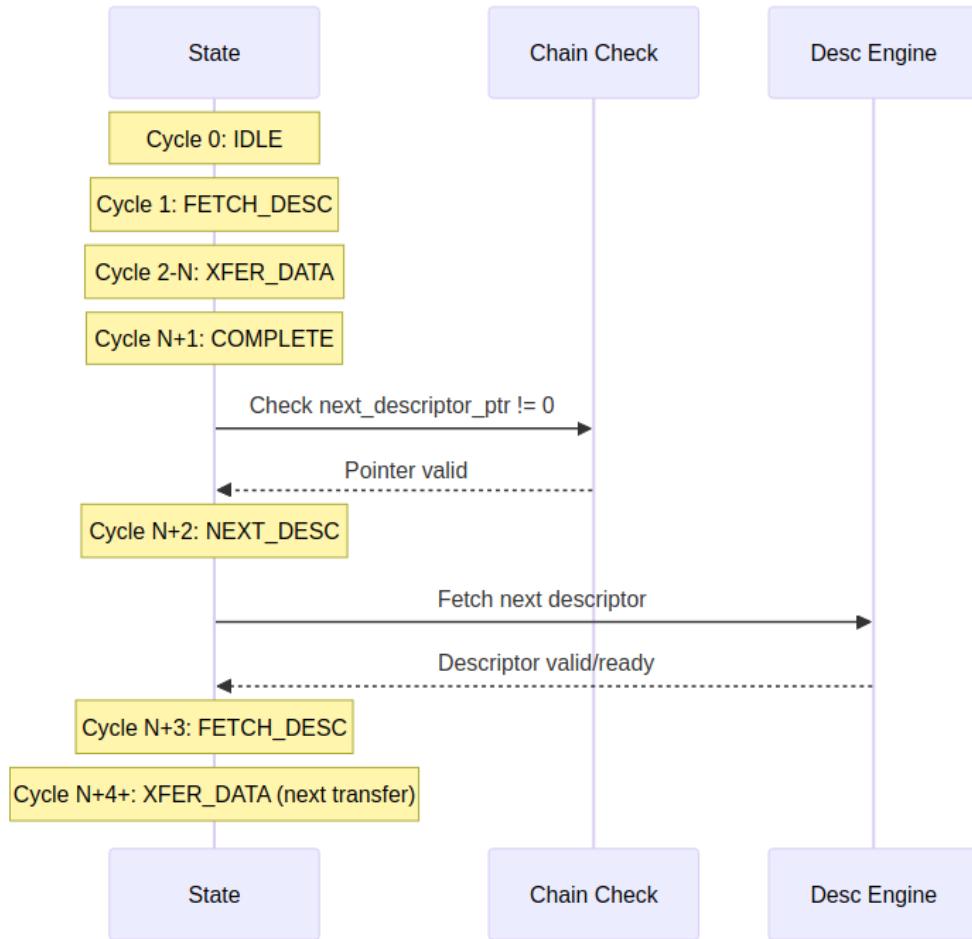
Scheduler Normal Transfer Timing

Source: [04_scheduler_normal_transfer.mmd](#)

Notes: - Both read and write run concurrently in XFER state - Independent done strobes decrement separate counters

9.15.3 Descriptor Chaining

9.15.4 Figure 2.4.4: Scheduler Descriptor Chaining Timing



Scheduler Descriptor Chaining Timing

Source: [04_scheduler_chaining.mmd](#)

9.15.5 Single-Descriptor Transfer (Detailed)

Cycle	State	sched_wr_valid	sched_wr_ready	sched_wr_beats
Notes				
-----	-----	-----	-----	-----
0	IDLE	0	0	0
Waiting				
1	FETCH_DESC	0	0	0
Request descriptor				
2	FETCH_DESC	0	0	0
(fetch latency)				

3	XFER_DATA	0	0	0
Descriptor received				
...	(read phase)			
50	XFER_DATA	1	0	256
Assert write request				
51	XFER_DATA	1	1	256
Engine ready, handshake!				
52	XFER_DATA	1	0	256
Ready cleared (registered)				
...	(engine executes burst)			
100	XFER_DATA	1	0	256
done_strobe=1, beats_done=16				
101	XFER_DATA	1	1	240
Ready re-asserts, new beats_remaining				
102	XFER_DATA	1	0	240
Handshake again				
...	(continues until beats_remaining == 0)			

9.15.6 Chained Descriptors (Detailed)

Descriptor chain:

```
Desc 0: length=128, next_ptr=0x1000_0100
Desc 1: length=64, next_ptr=0
```

Cycle	State	Descriptor	sched_wr_beats	Notes
0	IDLE	-	0	Start
1	FETCH_DESC	Req @0x1000 0	0	Fetch first
3	XFER_DATA	Desc 0	128	Transfer desc 0
...	(8 bursts × 16 beats)			
200	COMPLETE	Desc 0	0	Transfer done
201	NEXT_DESC	Check ptr	0	next_ptr = 0x1000_0100
202	FETCH_DESC	Req @0x1100 0	0	Fetch second
204	XFER_DATA	Desc 1	64	Transfer desc 1
...	(4 bursts × 16 beats)			
400	COMPLETE	Desc 1	0	Transfer done
401	NEXT_DESC	Check ptr	0	next_ptr = 0 →
terminate				
402	IDLE	-	0	Chain complete

9.16 Testing

Test Location: projects/components/stream/dv/tests/fub_tests/scheduler/

Key Test Scenarios:

1. **Single descriptor transfer** - Basic operation
 2. **Descriptor chaining** - 2-4 descriptors linked
 3. **Concurrent read/write** - Verify no deadlock with small SRAM
 4. **Large transfer (> SRAM)** - 100MB transfer with 2KB SRAM
 5. **IRQ generation** - gen_irq flag set
 6. **Error handling** - Descriptor, read, write errors
 7. **Timeout detection** - Engine stall scenarios
 8. **Channel reset** - cfg_channel_reset during transfer
 9. **Runtime timeout config** - Change cfg_sched_timeout_cycles dynamically
 10. **sched_rd_ready / sched_wr_ready timing validation** - Verify registered ready behavior
-

9.17 Performance Considerations

9.17.1 Concurrent Operation Benefit

Without Concurrency (Sequential): - Max transfer size = SRAM buffer size - Deadlock when transfer > buffer - Throughput = min(read_bw, write_bw)

With Concurrency: - No transfer size limit - Natural flow control via SRAM full/empty - Throughput = max(read_bw, write_bw) (pipeline overlap)

9.17.2 Example Performance

Configuration: - DATA_WIDTH = 512 bits (64 bytes/beat) - SRAM = 2KB (32 beats) - Transfer = 100MB (1,562,500 beats)

Sequential (hypothetical): - DEADLOCK at 2KB (can't complete read)

Concurrent: - Read fills SRAM (32 beats) - Write drains SRAM concurrently - Both engines sustain ~0.9 beats/cycle - Total time: ~1.7M cycles

9.18 Related Documentation

- **Descriptor Engine:** 05_descriptor_engine.md - Descriptor fetch
- **AXI Read Engine:** 06_axi_read_engine.md - Source data read
- **AXI Write Engine:** 12_axi_write_engine.md - Destination data write
- **SRAM Controller:** 08_sram_controller.md - Buffer management
- **Scheduler Group:** 03_scheduler_group.md - Single-channel integration

- **Scheduler Group Array:** 02_scheduler_group_array.md - Multi-channel integration
-

9.19 Revision History

Revision History

Date	Version	Changes
2025-10-17	1.0	Initial documentation with old signal names (datard_, datawr_)
2025-11-16	1.5	Enhanced documentation with detailed sections
2025-11-21	2.0	Merged documentation:- Updated all signal names (sched_rd_, sched_wr_)- Added runtime timeout configuration (cfg_sched_timeout_cycles/enable)- Registered ready signal timing clarification- Added multiple requests per descriptor section- Enhanced beat tracking and error handling details- Updated all code examples and timing diagrams- Added timing examples for chained descriptors- Combined best content from multiple documentation sources
2025-11-30	2.1	RTL Sync Update:- CH_ERROR is now STICKY (requires reset to recover)- scheduler_idle asserts in CH_ERROR state- Updated related documentation

Date	Version	Changes
		references

Last Updated: 2025-11-30 (matched to current RTL implementation)

10 Descriptor Engine

Module: descriptor_engine.sv **Location:** projects/components/stream/rtl/fub/
Category: FUB (Functional Unit Block) **Parent:** scheduler_group.sv **Status:** Implemented **Last Updated:** 2025-11-30

10.1 Overview

The descriptor_engine module is an autonomous descriptor fetch engine with chaining support. It fetches 256-bit STREAM descriptors from memory via AXI4 master interface and provides two operating modes for kick-off.

10.1.1 Key Features

- **Two Operating Modes:**
 - APB-initiated: Software writes descriptor address, engine fetches
 - Autonomous chaining: Engine automatically fetches next_descriptor_ptr
 - **Two-FIFO Architecture:**
 - Descriptor address FIFO: Holds addresses to fetch
 - Descriptor data FIFO: Buffers fetched descriptors for scheduler
 - **Address Range Validation:** Two configurable ranges for security
 - **MonBus Event Reporting:** Fetch complete and error events
-

10.2 Architecture

10.2.1 Operating Flow

APB Mode:

APB write -> skid buffer -> desc addr FIFO -> AXI fetch -> desc data FIFO -> scheduler

Chaining Mode:

```
Descriptor complete -> extract next_ptr -> validate -> desc addr FIFO  
->  
AXI fetch -> desc data FIFO -> scheduler (repeat until last=1 or  
next_ptr=0)
```

10.2.2 FSM States

```
typedef enum logic [2:0] {  
    RD_IDLE      = 3'b000, // Waiting for descriptor address  
    RD_ISSUE_ADDR = 3'b001, // Issue AXI AR transaction  
    RD_WAIT_DATA  = 3'b010, // Wait for AXI R response  
    RD_COMPLETE   = 3'b011, // Descriptor fetched, push to FIFO  
    RD_ERROR      = 3'b100 // Error occurred  
} read_engine_state_t;
```

10.2.3 Descriptor Format (256-bit STREAM)

Descriptor Format

Bits	Field	Description
[63:0]	src_addr	Source memory address (64-bit, must be aligned)
[127:64]	dst_addr	Destination memory address (64-bit, must be aligned)
[159:128]	length	Transfer length in BEATS (not bytes!)
[191:160]	next_descriptor_ptr	Address of next descriptor (32-bit, 0 = last)
[192]	valid	Descriptor valid flag
[193]	gen_irq	Generate interrupt on completion
[194]	last	Last descriptor in chain (explicit termination)
[195]	error	Descriptor error flag
[199:196]	channel_id	Channel identifier
[207:200]	priority	Descriptor priority
[255:208]	reserved	Reserved for future use

10.3 Parameters

Parameters

Parameter	Type	Default	Description
CHANNEL_ID	int	0	Channel identifier
NUM_CHANNELS	int	32	Total number of channels
CHAN_WIDTH	int	\$clog2(NUM_CHANNELS)	Channel ID width
ADDR_WIDTH	int	64	Address bus width
AXI_ID_WIDTH	int	8	AXI transaction ID width
FIFO_DEPTH	int	8	Descriptor data FIFO depth
DESC_ADDR_FIFO_DEPTH	int	2	Descriptor address FIFO depth
TIMEOUT_CYCLE_S	int	1000	AXI timeout threshold

10.3.1 Monitor Bus Parameters

Monitor Bus Parameters

Parameter	Type	Default	Description
MON_AGENT_ID	8-bit	0x10	Descriptor Engine Agent ID
MON_UNIT_ID	4-bit	0x1	Unit identifier
MON_CHANNEL_ID	6-bit	0x0	Base channel ID

10.4 Port List

10.4.1 Clock and Reset

Clock and Reset

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

10.4.2 APB Programming Interface

APB Programming Interface

Signal	Direction	Width	Description
apb_valid	input	1	Descriptor address valid
apb_ready	output	1	Ready to accept address
apb_addr	input	ADDR_WIDTH	Descriptor address

10.4.3 Scheduler Interface

Scheduler Interface

Signal	Direction	Width	Description
channel_idle	input	1	Scheduler idle (enables APB)
descriptor_va_lid	output	1	Descriptor available
descriptor_ready	input	1	Scheduler ready to accept
descriptor_packet	output	256	FIXED 256-bit descriptor
descriptor_error	output	1	Fetch error flag

10.4.4 Enhanced Control Outputs

Enhanced Control Outputs

Signal	Direction	Width	Description
descriptor_eos	output	1	End of Stream (future use)
descriptor_eol	output	1	End of Line (future use)
descriptor_eod	output	1	End of Data (future use)
descriptor_tyipe	output	2	Packet type (future use)

10.4.5 AXI4 AR Channel

AXI4 AR Channel

Signal	Direction	Width	Description
ar_valid	output	1	Address valid
ar_ready	input	1	Address ready
ar_addr	output	ADDR_WIDTH	Address
ar_len	output	8	Burst length - 1
ar_size	output	3	Burst size (log2 bytes)
ar_burst	output	2	Burst type (INCR)
ar_id	output	AXI_ID_WIDTH	Transaction ID
ar_lock	output	1	Lock type
ar_cache	output	4	Cache attributes
ar_prot	output	3	Protection attributes
ar_qos	output	4	QoS value
ar_region	output	4	Region identifier

10.4.6 AXI4 R Channel (FIXED 256-bit)

AXI4 R Channel

Signal	Direction	Width	Description
r_valid	input	1	Read data valid
r_ready	output	1	Read data ready
r_data	input	256	Read data (FIXED 256-bit)
r_resp	input	2	Response
r_last	input	1	Last beat
r_id	input	AXI_ID_WIDTH	Transaction ID

10.4.7 Configuration Interface

Configuration Interface

Signal	Direction	Width	Description
cfg_prefetch_enable	input	1	Enable descriptor prefetch
cfg_fifo_thre_shold	input	4	FIFO threshold for prefetch
cfg_addr0_base	input	ADDR_WIDTH	Address range 0 base
cfg_addr0_limit	input	ADDR_WIDTH	Address range 0 limit
cfg_addr1_base	input	ADDR_WIDTH	Address range 1 base
cfg_addr1_limit	input	ADDR_WIDTH	Address range 1 limit
cfg_channel_reset	input	1	Channel reset

10.4.8 Status Interface

Status Interface

Signal	Direction	Width	Description
descriptor_en_gine_idle	output	1	Engine idle

10.4.9 Monitor Bus Interface

Monitor Bus Interface

Signal	Direction	Width	Description
mon_valid	output	1	Monitor packet valid
mon_ready	input	1	Monitor packet ready
mon_packet	output	64	Monitor packet data

10.5 Operation

10.5.1 APB Acceptance Policy

APB writes are accepted ONLY when ALL conditions met: 1. !r_channel_reset_active - Not in channel reset 2. w_desc_addr_fifo_empty - No pending descriptor fetches 3. channel_idle - Scheduler completed all prior descriptors 4. !r_apb_ip - No APB transaction currently in progress 5. apb_addr != 0 - Address 0 is invalid

10.5.2 Autonomous Chaining Decision Tree

Level 1: Basic Eligibility (w_chain_condition) - next_descriptor_ptr != 0 (non-null pointer) - descriptor.last == 0 (not explicitly marked as last) - descriptor.valid == 1 (valid descriptor)

Level 2: Address Validation (w_next_addr_valid) - next_addr within cfg_addr0_base..cfg_addr0_limit OR - next_addr within cfg_addr1_base..cfg_addr1_limit

Level 3: Final Decision (w_should_chain) - Level 1 + Level 2 conditions met - !r_descriptor_error (no fetch error) - w_desc_fifo_wr_ready (descriptor FIFO has space)

10.5.3 AXI Transaction

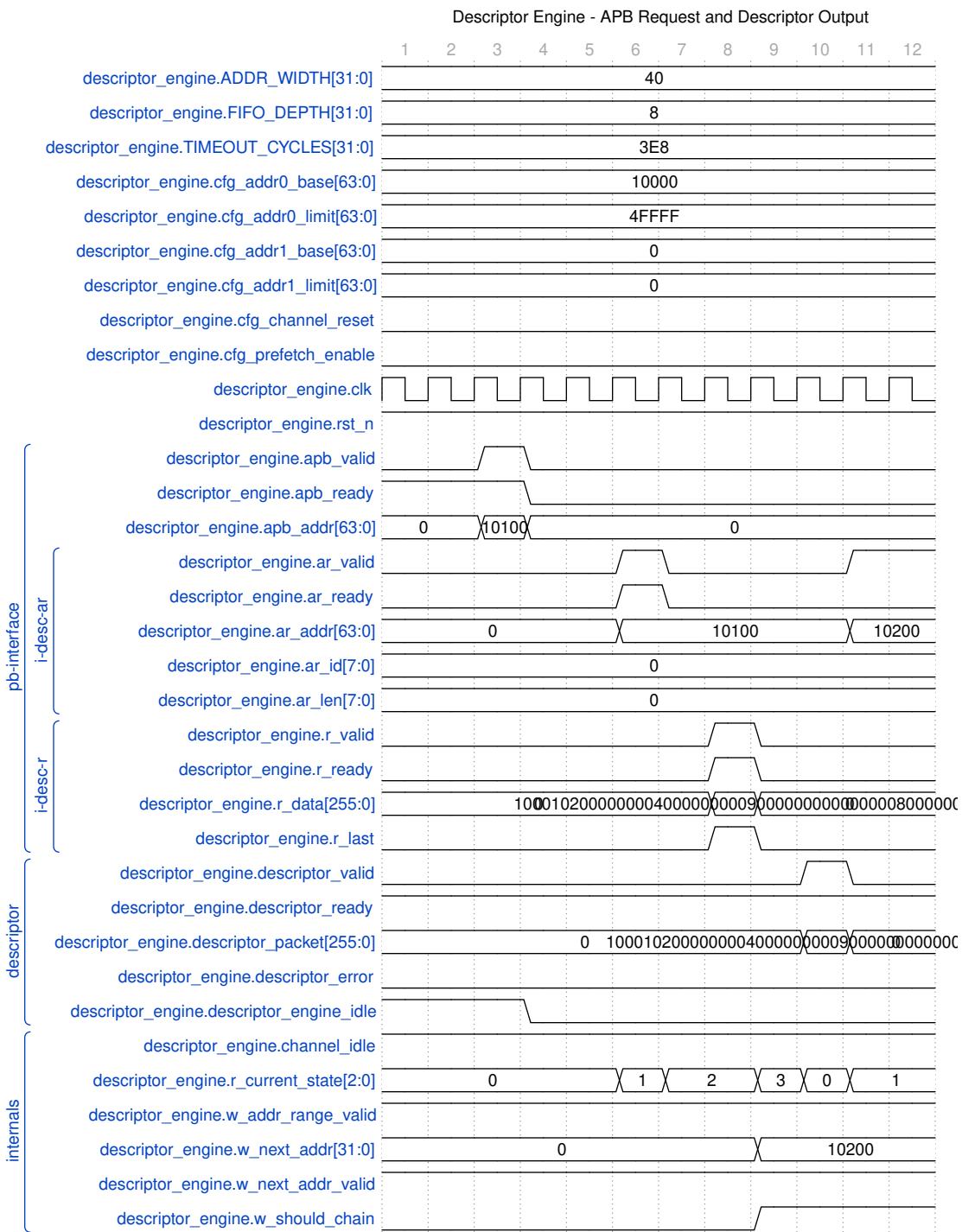
```
ar_len = 8'h00;           // Single beat transfer
ar_size = 3'b110;         // 64 bytes (512-bit for 256-bit descriptor)
ar_burst = 2'b01;         // INCR burst type
ar_id = {{padding}, CHANNEL_ID}; // Channel ID in lower bits
ar_cache = 4'b0010;       // Normal non-cacheable bufferable
ar_prot = 3'b000;         // Data, secure, unprivileged
```

10.6 Timing Diagrams

10.6.1 APB Kick-off Sequence

The following diagram shows the APB-initiated descriptor kick-off sequence:

10.6.1.1 Waveform 2.5.1: Descriptor Engine APB Basic Kick-off



Descriptor Engine APB Basic

Source: [descriptor_engine_apb_basic.json](#)

10.7 Integration Example

```
descriptor_engine #(
    .CHANNEL_ID          (ch),
    .NUM_CHANNELS        (8),
    .ADDR_WIDTH          (64),
    .AXI_ID_WIDTH        (8),
    .FIFO_DEPTH          (8),
    .MON_AGENT_ID        (8'h10 + ch)
) u_descriptor_engine (
    .clk                 (clk),
    .rst_n               (rst_n),

    // APB interface
    .apb_valid           (apb_valid),
    .apb_ready            (apb_ready),
    .apb_addr             (apb_addr),

    // Scheduler interface
    .channel_idle         (scheduler_idle),
    .descriptor_valid     (desc_valid),
    .descriptor_ready     (desc_ready),
    .descriptor_packet    (desc_packet),
    .descriptor_error     (desc_error),

    // AXI interface
    .ar_valid              (ar_valid),
    .ar_ready              (ar_ready),
    .ar_addr               (ar_addr),
    // ... full AR/R interface

    // Configuration
    .cfg_prefetch_enable   (cfg_desceng_prefetch),
    .cfg_addr0_base        (cfg_desceng_addr0_base),
    .cfg_addr0_limit        (cfg_desceng_addr0_limit),
    .cfg_addr1_base        (cfg_desceng_addr1_base),
    .cfg_addr1_limit        (cfg_desceng_addr1_limit),
    .cfg_channel_reset      (cfg_channel_reset),

    // Status
    .descriptor_engine_idle (desc_engine_idle),

    // Monitor bus
    .mon_valid              (desceng_mon_valid),
```

```
.mon_ready          (desceng_mon_ready),  
.mon_packet        (desceng_mon_packet)

---


```

10.8 Common Issues

10.8.1 Issue 1: APB Write Not Accepted

Symptom: apb_ready never asserts

Root Causes: 1. Channel not idle (channel_idle = 0) 2. Previous APB transaction still in progress (r_apb_ip = 1) 3. Descriptor address FIFO not empty 4. Channel reset active

Solution: Wait for channel to complete current work before next kick-off.

10.8.2 Issue 2: Descriptor Chain Stops Early

Symptom: Chaining stops before expected

Root Causes: 1. next_descriptor_ptr is 0 or out of valid range 2. last flag set in descriptor 3. AXI error during fetch

Debug: Check descriptor memory contents and address range configuration.

10.9 Related Documentation

- **Parent:** 03_scheduler_group.md - Channel wrapper
- **Consumer:** 04_scheduler.md - Scheduler that receives descriptors
- **Arbiter:** 02_scheduler_group_array.md - Descriptor AXI arbitration
- **APB Interface: 13_apbtodescr.md - APB to descriptor kick-off**

10.10 Revision History

Descriptor Engine Revision History

Version	Date	Author	Description
0.90	2025-11-22	seang	Initial block specification
0.91	2026-01-02	seang	Added table captions and

Version	Date	Author	Description
			figure numbers

Last Updated: 2026-01-02

11 AXI Read Engine

Module: axi_read_engine.sv **Location:** projects/components/stream/rtl/fub/ **Category:** FUB (Functional Unit Block) **Parent:** stream_core.sv **Status:** Implemented **Last Updated:** 2025-11-30

11.1 Overview

The axi_read_engine module is a high-performance multi-channel AXI4 read engine with space-aware arbitration. It serves all 8 STREAM channels through a single AXI master interface with intelligent flow control.

11.1.1 Key Features

- **Round-Robin Arbitration:** Fair scheduling across channels
 - **Space-Aware Masking:** Only arbitrate channels with sufficient SRAM space
 - **Pre-Allocation Handshake:** Reserve SRAM space before data arrives
 - **Streaming Data Path:** Direct passthrough to SRAM controller
 - **Channel ID in AXI ID:** Enables per-channel response routing
 - **Pipelined/Non-Pipelined Modes:** Configurable outstanding transaction depth
-

11.2 Architecture

11.2.1 Operation Flow

1. Scheduler Interface: Each channel can request read bursts
2. Space Checking: Mask channels without sufficient SRAM space
3. Arbitration: Round-robin arbiter selects next channel to service
4. AXI AR Issue: Issue read command to AXI, assert rd_alloc to SRAM controller
5. AXI R Response: Stream read data directly to SRAM controller

11.2.2 Key Design Decisions

Combinational AR Outputs: AR outputs are driven组合性地 from the arbiter to avoid 1-cycle delay. When `axi_rd_alloc_space_free` goes to 0, `arvalid` drops in the same cycle.

No Internal Buffering: The engine is a streaming pipeline with no internal data storage. Data flows directly from AXI R channel to SRAM controller.

11.2.3 System Idle Behavior

When ALL channels complete their work (`w_arb_request == 0`), the engine becomes idle until new requests arrive. This is NOT a bubble - it's legitimate system idle.

Why This Matters for Testing: - With few active channels and short bursts, there are brief periods when all channels are in their WRITE phase - During these periods, `m_axi_arvalid = 0` (no R channel activity) - This is EXPECTED behavior, not a performance bug - The `dbg_arb_request` signal exposes `w_arb_request` to help testbenches distinguish: - **TRUE BUBBLE:** `arvalid=0` while `arb_request!=0` (channels waiting, arbiter stalled) - **SYSTEM IDLE:** `arvalid=0` while `arb_request==0` (no channels need service)

11.3 Parameters

Parameters

Parameter	Type	Default	Description
NUM_CHANNELS	int	8	Number of channels
ADDR_WIDTH	int	64	AXI address width
DATA_WIDTH	int	512	AXI data width
ID_WIDTH	int	8	AXI ID width
SEG_COUNT_WIDTH	int	8	Width of space/count signals
PIPELINE	int	0	0: non-pipelined, 1: pipelined
AR_MAX_OUTSTANDING	int	8	Maximum outstanding AR requests (PIPELINE=1)
STROBE_EVERY_BEAT	int	0	0: strobe on last beat, 1: strobe every beat

11.3.1 Derived Parameters

Derived Parameters

Parameter	Derivation	Description
NC	NUM_CHANNELS	Short alias
AW	ADDR_WIDTH	Short alias
DW	DATA_WIDTH	Short alias
IW	ID_WIDTH	Short alias
SCW	SEG_COUNT_WIDTH	Segment count width
CIW	\$clog2(NC)	Channel ID width (min 1 bit)

11.4 Port List

11.4.1 Clock and Reset

Clock and Reset

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

11.4.2 Configuration Interface

Configuration Interface

Signal	Direction	Width	Description
cfg_axi_rd_xfer_beats	input	8	Transfer size in beats (all channels)

11.4.3 Scheduler Interface (Per-Channel)

Scheduler Interface

Signal	Direction	Width	Description
sched_rd_vali_d[ch]	input	NC	Channel requests read
sched_rd_addr	input	NC x AW	Source

Signal	Direction	Width	Description
[ch]			addresses
sched_rd_beat s[ch]	input	NC x 32	Beats remaining to read

11.4.4 Completion Interface (Per-Channel)

Completion Interface

Signal	Direction	Width	Description
sched_rd_don_e_strobe[ch]	output	NC	Burst completed (1 cycle pulse)
sched_rd_beats_done[ch]	output	NC x 32	Number of beats completed

11.4.5 SRAM Allocation Interface

SRAM Allocation Interface

Signal	Direction	Width	Description
axi_rd_alloc_req	output	1	Request space allocation
axi_rd_alloc_size	output	8	Beats to reserve
axi_rd_alloc_id	output	IW	Transaction ID (channel)
axi_rd_alloc_space_free[ch]	input	NC x SCW	Free space per channel

11.4.6 SRAM Write Interface

SRAM Write Interface

Signal	Direction	Width	Description
axi_rd_sram_valid	output	1	Read data valid
axi_rd_sram_ready	input	1	Ready to accept data
axi_rd_sram_id	output	IW	Transaction ID (channel)

Signal	Direction	Width	Description
axi_rd_sram_d ata	output	DW	Read data payload

11.4.7 AXI4 AR Channel

AXI4 AR Channel

Signal	Direction	Width	Description
m_axi_arvalid	output	1	Address valid
m_axi_arready	input	1	Address ready
m_axi_arid	output	IW	Transaction ID
m_axi_araddr	output	AW	Address
m_axi_arlen	output	8	Burst length - 1
m_axi_arsize	output	3	Burst size (log2 bytes)
m_axi_arburst	output	2	Burst type (INCR)

11.4.8 AXI4 R Channel

AXI4 R Channel

Signal	Direction	Width	Description
m_axi_rvalid	input	1	Read data valid
m_axi_rready	output	1	Read data ready
m_axi_rid	input	IW	Transaction ID
m_axi_rdata	input	DW	Read data
m_axi_rrresp	input	2	Response
m_axi_rlast	input	1	Last beat of burst

11.4.9 Error Interface

Error Interface

Signal	Direction	Width	Description
sched_rd_error	output	NC	Sticky error

Signal	Direction	Width	Description
r[ch]			flag per channel

11.4.10 Debug Interface

Debug Interface

Signal	Direction	Width	Description
dbg_rd_all_complete[ch]	output	NC	All reads complete
dbg_r_beats_rcvd	output	32	Total R beats received
dbg_sram_writes	output	32	Total writes to SRAM
dbg_arb_request[ch]	output	NC	Arbiter request vector

11.5 Operation

11.5.1 Space-Aware Request Masking

```
// Only request arbitration if:
// 1. Scheduler is requesting (sched_rd_valid)
// 2. Sufficient SRAM space available (w_space_ok)
// 3. Below outstanding limit (w_below_outstanding_limit)
w_arb_request[i] = sched_rd_valid[i] && w_space_ok[i] &&
w_below_outstanding_limit[i];
```

11.5.2 Transfer Size Calculation

```
// Calculate actual transfer size for this channel
// Min of remaining beats or configured max
w_transfer_size[i] = (sched_rd_beats[i] <= cfg_axi_rd_xfer_beats +
1) ?
(sched_rd_beats[i] - 1) : cfg_axi_rd_xfer_beats;
```

11.5.3 Outstanding Transaction Tracking

PIPELINE=0 (Non-Pipelined): - Binary flag per channel (0 or 1 outstanding) - Set when AR issues for channel - Clear when R last arrives for channel

PIPELINE=1 (Pipelined): - Counter per channel (0 to AR_MAX_OUTSTANDING) - Increment on AR handshake - Decrement on R last handshake

11.5.4 Completion Strobe

Strobes fire when AR transaction is accepted (not when R completes):

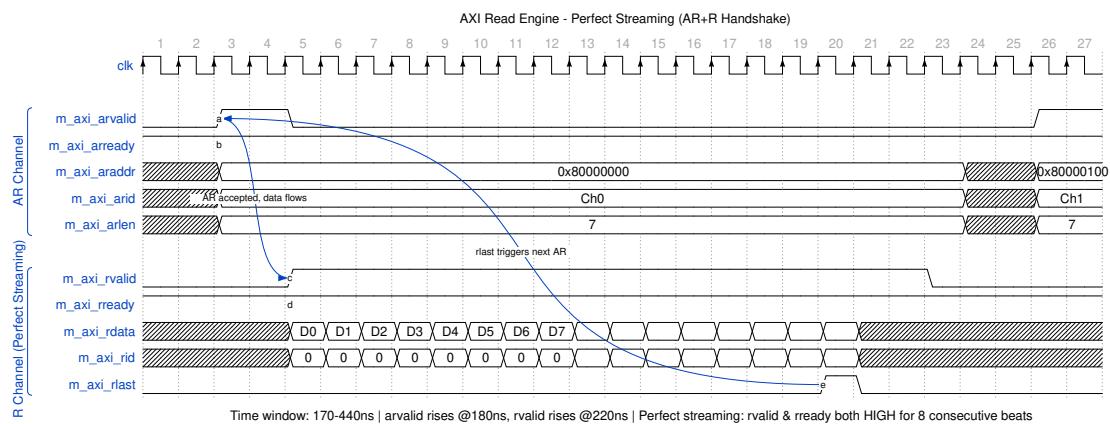
```
// Pulse when AR transaction is accepted (arvalid && arready)
// This tells scheduler that request was issued to AXI bus
if (m_axi_arvalid && m_axi_arready) begin
    r_done_strobe[w_arb_grant_id] <= 1'b1;
    r_beats_done[w_arb_grant_id] <= w_transfer_size[w_arb_grant_id] +
1;
end
```

11.6 Timing Diagrams

11.6.1 Perfect Streaming - AXI Read Transaction

The following timing diagram shows the AXI read engine operating at maximum throughput with **perfect streaming** - where both rvalid and rready remain HIGH for consecutive clock cycles, achieving one data beat per cycle.

11.6.1.1 Waveform 2.6.1: AXI Read Engine - Perfect Streaming



AXI Read Engine - Perfect Streaming

Transaction Flow:

1. AR Channel Handshake (Address Phase)

- m_axi_arvalid rises to initiate a read request
- m_axi_arready is already HIGH (slave ready)
- Address (araddr), ID (arid), and length (arlen=7 for 8 beats) are captured

- Handshake completes in a single cycle when both valid and ready are HIGH

2. R Channel Streaming (Data Phase)

- After AR acceptance, the AXI slave begins returning data
- `m_axi_rvalid` rises and **stays HIGH** for all 8 beats
- `m_axi_rready` remains HIGH throughout (no backpressure from SRAM controller)
- **Perfect streaming:** One data beat transferred every clock cycle
- `m_axi_rlast` rises on the final beat (beat 7) to mark burst completion

3. Next Transaction

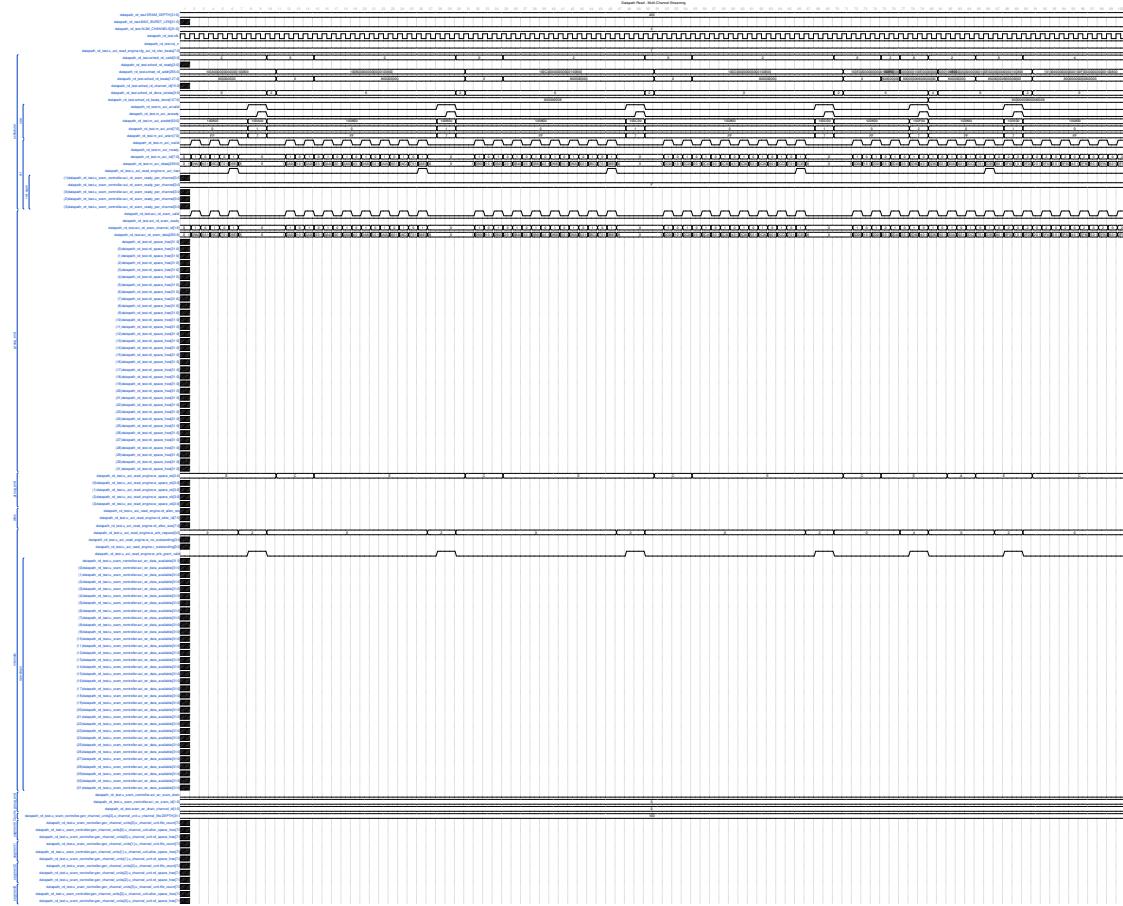
- After `rlast`, the engine can immediately issue the next AR request
- The cycle repeats for the next channel or next burst

Key Performance Indicators: - **No bubbles:** `rvalid` and `rready` both HIGH during data phase - **Full bandwidth:** Data width (256/512 bits) x clock frequency - **Zero-wait states:** SRAM controller accepts data at line rate

11.6.2 Multi-Channel Streaming

For multi-channel operation showing channel switching while maintaining streaming performance, see:

11.6.2.1 Waveform 2.6.2: Datapath Read - Multi-Channel



Datapath Read - Multi-Channel

This diagram shows how the engine arbitrates between channels while maintaining high throughput.

11.7 Integration Example

```
axi_read_engine #(
    .NUM_CHANNELS      (8),
    .ADDR_WIDTH        (64),
    .DATA_WIDTH        (512),
    .ID_WIDTH          (8),
    .SEG_COUNT_WIDTH   (10),
    .PIPELINE           (0),
    .AR_MAX_OUTSTANDING (8)
) u_axi_read_engine (
    .clk
```

```

.rst_n          (rst_n),
// Configuration
.cfg_axi_rd_xfer_beats (cfg_axi_rd_xfer_beats),
// Scheduler interface
.sched_rd_valid      (sched_rd_valid),
.sched_rd_addr       (sched_rd_addr),
.sched_rd_beats     (sched_rd_beats),
// Completion interface
.sched_rd_done_strobe (sched_rd_done_strobe),
.sched_rd_beats_done (sched_rd_beats_done),
// SRAM allocation interface
.axi_rd_alloc_req    (axi_rd_alloc_req),
.axi_rd_alloc_size   (axi_rd_alloc_size),
.axi_rd_alloc_id     (axi_rd_alloc_id),
.axi_rd_alloc_space_free(axi_rd_alloc_space_free),
// SRAM write interface
.axi_rd_sram_valid   (axi_rd_sram_valid),
.axi_rd_sram_ready   (axi_rd_sram_ready),
.axi_rd_sram_id      (axi_rd_sram_id),
.axi_rd_sram_data    (axi_rd_sram_data),
// AXI master
.m_axi_arvalid      (m_axi_rd_arvalid),
.m_axi_arready      (m_axi_rd_arready),
.m_axi_arid         (m_axi_rd_arid),
.m_axi_araddr       (m_axi_rd_araddr),
.m_axi_arlen        (m_axi_rd_arlen),
.m_axi_arsize       (m_axi_rd_arsize),
.m_axi_arburst      (m_axi_rd_arburst),
.m_axi_rvalid       (m_axi_rd_rvalid),
.m_axi_rrready      (m_axi_rd_rrready),
.m_axi_rid          (m_axi_rd_rid),
.m_axi_rdata        (m_axi_rd_rdata),
.m_axi_rresp        (m_axi_rd_rresp),
.m_axi_rlast        (m_axi_rd_rlast),
// Error and debug
.sched_rd_error      (sched_rd_error),
.dbg_rd_all_complete (dbg_rd_all_complete),

```

```
    .dbg_arb_request          (dbg_arb_request)
);
```

11.8 Related Documentation

- **Parent:** 01_stream_core.md - Top-level integration
 - **Scheduler Array:** 02_scheduler_group_array.md - Provides sched_rd_* signals
 - **Allocation Controller:** 07_stream_alloc_ctrl.md - Space tracking for read engine
 - **SRAM Controller:** 08_sram_controller.md - Receives read data
 - **Write Engine:** [12_axi_write_engine.md - Complementary write datapath](#)
-

11.9 Revision History

AXI Read Engine Revision History

Version	Date	Author	Description
0.90	2025-11-22	seang	Initial block specification
0.91	2026-01-02	seang	Added table captions and figure numbers

Last Updated: 2026-01-02

12 Stream Allocation Controller

Module: stream_alloc_ctrl.sv **Location:** projects/components/stream/rtl/fub/
Category: FUB (Functional Unit Block) **Parent:** sram_controller_unit.sv **Status:** Implemented
Last Updated: 2025-11-30

12.1 Overview

The `stream_alloc_ctrl` module is a **virtual FIFO without data storage** that tracks space allocation and availability using FIFO pointer logic. It provides pre-allocation support for AXI read engines to prevent race conditions between burst requests and data arrival.

12.1.1 What Makes This a “Virtual FIFO”

Virtual FIFO: - Has write pointer (allocation pointer) - Has read pointer (actual write pointer) - Calculates full/empty/space_free - **NO data storage** - only pointer arithmetic

Use Case: Reserve FIFO space BEFORE AXI read data arrives

12.2 The Allocation Problem

12.2.1 Without Allocation Controller

Problem: Race condition between space check and data arrival

Cycle 0: Read engine checks FIFO space
space_free = 100 beats → OK to issue 16-beat burst

Cycle 5: AR handshake completes, AXI read starts

Cycle 10: Meanwhile, write engine drains 90 beats from FIFO
space_free = 10 beats (NOT ENOUGH!)

Cycle 15: AXI read data starts arriving (16 beats)
→ OVERFLOW! Only 10 beats of space available

12.2.2 With Allocation Controller

Solution: Reserve space when issuing AR, not when data arrives

Cycle 0: Read engine checks space_free = 100 beats
Allocate 16 beats:
rd_alloc_req = 1, rd_alloc_size = 16
→ space_free = 84 beats (reserved!)

Cycle 5: AR handshake completes

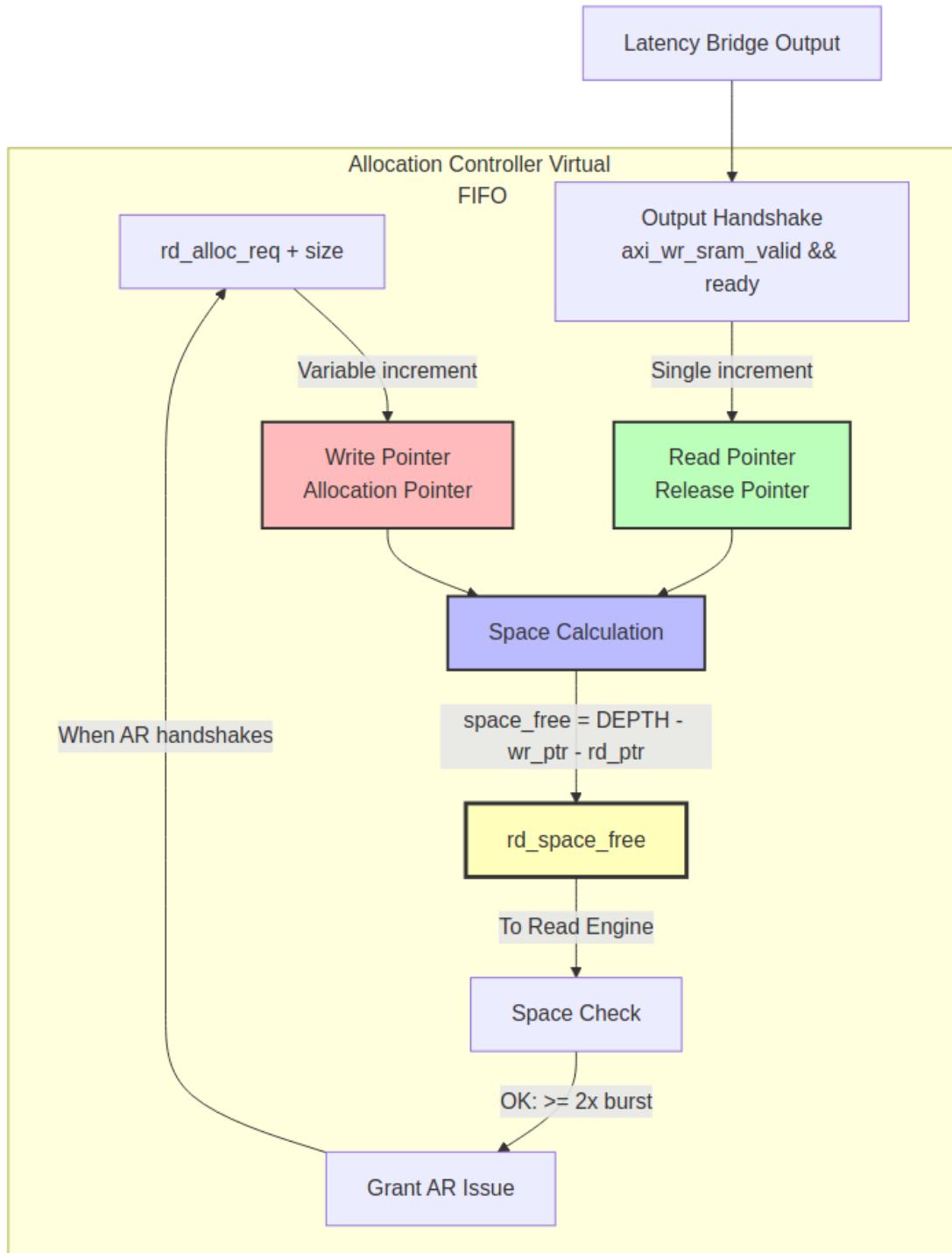
Cycle 10: Write engine checks space_free = 84 beats
→ Sees reduced space, won't overdrain

Cycle 15: AXI read data arrives, enters FIFO
→ Space was reserved, guaranteed to fit

12.3 Architecture

12.3.1 Two-Pointer System

12.3.2 Figure 2.7.1: Stream Allocation Controller Block Diagram



Diagram

→

Write Pointer (Allocation Pointer): - Advances when space is **allocated** (burst request) - Variable-size increment (rd_alloc_size) - Represents “reserved space”

Read Pointer (Actual Write Pointer): - Advances when data **exits the controller** (output handshake) - Single-beat increment - Represents “released space”

Space Calculation:

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

12.3.3 CRITICAL: Confusing Naming Convention

WARNING: Allocation controller uses OPPOSITE naming from normal FIFO!

Normal FIFO: | Signal | Meaning | |——|——| | wr_* | Write data → Consumes space → space_free decreases | | rd_* | Read data → Frees space → space_free increases |

: CRITICAL: Confusing Naming Convention

Allocation Controller: | Signal | Meaning | |——|——| | wr_* | **ALLOCATE** space → Reserves space → space_free **decreases** | | rd_* | **RELEASE** space → Data exits controller → space_free **increases** |

: CRITICAL: Confusing Naming Convention

Why This Matters:

```
// "Write" side = ALLOCATE (reserve space for upcoming burst)
// Read engine says "I need 16 beats"
.wr_valid (rd_alloc_req),
.wr_size (rd_alloc_size), // Variable size (16 in this example)

// "Read" side = RELEASE (data exits controller, free space)
// Must monitor OUTPUT handshake (after latency bridge!)
.rd_valid (axi_wr_sram_valid && axi_wr_sram_ready), // Output side!
```

Key Insight: The “read” side of allocation controller connects to the OUTPUT of the FIFO + latency bridge, NOT the FIFO input!

12.4 Parameters

Parameters

Parameter	Type	Default	Description
DEPTH	int	512	Virtual FIFO depth (must match physical FIFO)

Parameter	Type	Default	Description
ALMOST_WR_MAR_GIN	int	1	Almost full threshold
ALMOST_RD_MAR_GIN	int	1	Almost empty threshold
REGISTERED	int	1	Register outputs for timing

12.5 Port List

12.5.1 Clock and Reset

Clock and Reset

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

12.5.2 Write Interface (Allocation Requests)

Write Interface

Signal	Direction	Width	Description
wr_valid	input	1	Allocate space request
wr_size	input	8	Number of entries to allocate
wr_ready	output	1	Space available (!wr_full)

12.5.3 Read Interface (Actual Data Written)

Read Interface

Signal	Direction	Width	Description
rd_valid	input	1	Data exits controller (release space)
rd_ready	output	1	Not empty (! rd_empty)

12.5.4 Status Outputs

Status Outputs

Signal	Direction	Width	Description
space_free	output	AW+1	Available space (beats)
wr_full	output	1	Full flag (no space available)
wr_almost_full	output	1	Almost full flag
rd_empty	output	1	Empty flag (no allocations)
rd_almost_empty	output	1	Almost empty flag

12.6 Interfaces

12.6.1 Write Interface (Allocation Requests)

Write Interface

Signal	Direction	Width	Description
wr_valid	Input	1	Allocate space
wr_size	Input	8	Number of entries to allocate
wr_ready	Output	1	Space

Signal	Direction	Width	Description
			available (!wr_full)

Usage:

```
// Read engine allocates space before issuing AR
rd_alloc_req = (space_check_ok && !ar_pending);
rd_alloc_size = cfg_axi_rd_xfer_beats;
```

12.6.2 Read Interface (Actual Data Written)

Read Interface

Signal	Direction	Width	Description
rd_valid	Input	1	Data exits controller
rd_ready	Output	1	Not empty (!rd_empty)

Usage:

```
// Connect to OUTPUT handshake (after latency bridge)
assign rd_valid = (axi_wr_sram_valid && axi_wr_sram_ready);
```

12.6.3 Status Outputs

Status Outputs

Signal	Direction	Width	Description
space_free	Output	AW+1	Available unreserved space
wr_full	Output	1	No space available
wr_almost_full	Output	1	Almost full
rd_empty	Output	1	No allocations pending
rd_almost_empty	Output	1	Almost empty

Note: space_free is the most important output - used by read engine for space checking.

12.7 Operation

12.7.1 Allocation Flow

Step 1: Check Space

```
// Read engine checks space availability
if (space_free >= (cfg_axi_rd_xfer_beats << 1)) begin // 2x margin
    // Space OK, proceed to allocation
end
```

Step 2: Allocate Space

```
rd_alloc_req = 1'b1;
rd_alloc_size = 8'd16; // Reserve 16 beats
```

```
// Next cycle: wr_ptr advances
r_wr_ptr_bin <= r_wr_ptr_bin + 16;
space_free decreases by 16
```

Step 3: AXI Read Executes

```
// Some cycles later, AXI AR handshake completes
m_axi_arvalid = 1, m_axi_arready = 1
// AXI read starts
```

Step 4: Data Arrives at FIFO

```
// Many cycles later, AXI read data arrives
axi_rd_sram_valid = 1, axi_rd_sram_ready = 1
// Data enters FIFO (allocation controller doesn't see this directly)
```

Step 5: Data Exits Controller (After Latency Bridge)

```
// Eventually, data traverses FIFO + latency bridge
axi_wr_sram_valid = 1, axi_wr_sram_ready = 1
```

```
// THIS triggers allocation controller release!
rd_valid = 1
r_rd_ptr_bin <= r_rd_ptr_bin + 1
space_free increases by 1
```

12.7.2 Pointer Arithmetic

Write Pointer (Allocation):

```
// Variable-size increment
if (w_write && !r_wr_full) begin
    r_wr_ptr_bin <= r_wr_ptr_bin + (AW+1)'(wr_size);
end
```

Read Pointer (Release):

```
// Single-beat increment (uses counter_bin utility)
counter_bin #(
    .WIDTH (AW + 1),
    .MAX   (D)
) read_pointer_inst (
    .clk      (axi_aclk),
    .rst_n    (axi_aresetn),
    .enable   (w_read && !r_rd_empty),
    .counter_bin_curr (r_rd_ptr_bin),
    .counter_bin_next (w_rd_ptr_bin_next)
);
```

Space Calculation:

```
// Occupancy = wr_ptr - rd_ptr
w_count = w_wr_ptr_bin_next - w_rd_ptr_bin_next;

// Space free = total depth - occupancy
space_free = (AW+1)'(D) - w_count;
```

12.8 Timing Behavior

12.8.1 Allocation Latency

Allocation is IMMEDIATE (combinational + 1 cycle):

```
Cycle N: rd_alloc_req = 1, rd_alloc_size = 16
Cycle N+1: r_wr_ptr_bin = old_value + 16
            space_free = old_value - 16
```

Read engine sees updated space_free on next cycle.

12.8.2 Release Latency

Release is IMMEDIATE (combinational + 1 cycle):

```
Cycle N: axi_wr_sram_valid = 1, axi_wr_sram_ready = 1
          → rd_valid = 1
Cycle N+1: r_rd_ptr_bin = old_value + 1
            space_free = old_value + 1
```

Write engine sees updated space_free on next cycle.

12.9 Integration Example

12.9.1 In sram_controller_unit.sv

```
stream_alloc_ctrl #(
    .DEPTH(SD),
    .REGISTERED(1)
) u_alloc_ctrl (
    .axi_aclk          (clk),
    .axi_aresetn       (rst_n),
    // ALLOCATE (reserve space for upcoming burst)
    .wr_valid          (rd_alloc_req),           // From read engine
    .wr_size           (rd_alloc_size),         // Burst size
    .wr_ready          (),                      // Unused (space check
uses space_free)

    // RELEASE (data exits controller - OUTPUT handshake!)
    .rd_valid          (axi_wr_sram_valid && axi_wr_sram_ready), // After latency bridge!
    .rd_ready          (),                      // Unused

    // Space tracking
    .space_free        (alloc_space_free),   // To read engine (via register)

    // Unused status
    .wr_full           (),
    .wr_almost_full   (),
    .rd_empty          (),
    .rd_almost_empty  ()
);

// Register output to break long paths
`ALWAYS_FF_RST(clk, rst_n,
    if (^RST_ASSERTED(rst_n)) begin
        rd_space_free <= SCW'(SD); // Full space on reset
    end else begin
        rd_space_free <= alloc_space_free;
    end
)
```

12.10 Debug Support

12.10.1 Display Statements

Allocation:

```

$display("ALLOC @ %t: allocated %0d beats, wr_ptr: %0d -> %0d,
space_free will be %0d",
        $time, wr_size, r_wr_ptr_bin, r_wr_ptr_bin + wr_size,
        D - (r_wr_ptr_bin + wr_size - r_rd_ptr_bin));

```

Release (Drain):

```

$display("DRAIN @ %t: drained 1 beat, rd_ptr: %0d -> %0d, space_free
will be %0d",
        $time, r_rd_ptr_bin, w_rd_ptr_bin_next,
        D - (r_wr_ptr_bin - w_rd_ptr_bin_next));

```

12.10.2 Waveform Analysis

Key Signals to Monitor: - r_wr_ptr_bin - Allocation pointer - r_rd_ptr_bin - Release pointer - space_free - Available space - rd_alloc_req, rd_alloc_size - Allocation requests - axi_wr_sram_valid, axi_wr_sram_ready - Release handshake

12.11 Common Issues

12.11.1 Issue 1: Space Not Released

Symptom: space_free decreases but never increases

Root Cause: rd_valid not connected to output handshake

Wrong:

```
.rd_valid (axi_rd_sram_valid && axi_rd_sram_ready) // FIFO input -
WRONG!
```

Correct:

```
.rd_valid (axi_wr_sram_valid && axi_wr_sram_ready) // After latency
bridge - CORRECT!
```

12.11.2 Issue 2: Overflow Despite Allocation

Symptom: FIFO overflows even with allocation controller

Root Cause: Read engine uses wrong space value

Wrong:

```
if (space_free >= cfg_axi_rd_xfer_beats) begin // Exact match -
WRONG!
    allocate();
end
```

Correct:

```

if (space_free >= (cfg_axi_rd_xfer_beats << 1)) begin // 2x margin -
CORRECT!
    allocate();
end

```

Why 2x: Accounts for in-flight allocations and pipeline delays.

12.11.3 Issue 3: Allocation Pointer Overflow

Symptom: r_wr_ptr_bin wraps at unexpected value

Root Cause: Pointer width insufficient

Check:

```

parameter int AW = $clog2(D); // Address width
logic [AW:0] r_wr_ptr_bin; // AW+1 bits (for full detection)

```

Extra bit allows distinguishing full ($\text{wr_ptr} = \text{rd_ptr} + \text{DEPTH}$) from empty ($\text{wr_ptr} = \text{rd_ptr}$).

12.12 Comparison with Drain Controller

Comparison with Drain Controller

Aspect	Allocation Controller	Drain Controller
Purpose	Reserve FIFO space	Reserve FIFO data
User	AXI read engine	AXI write engine
Write side	Allocation request (reserve)	Data enters FIFO (increment)
Read side	Data exits controller (release)	Drain request (reserve)
Naming	OPPOSITE of normal FIFO	SAME as normal FIFO
Output	space_free	data_available

12.13 Resource Utilization

Per Instance: - $2 \times (\text{AW}+1)$ -bit counters (wr_ptr , rd_ptr) - $1 \times (\text{AW}+1)$ -bit space_free calculation - FIFO control block (full/empty logic) - ~50-100 flip-flops total

Example (DEPTH=512, AW=9): - 2×10 -bit counters = 20 FFs - Control logic = ~30 FFs - **Total:** ~50 FFs

Very lightweight - pointer logic only, no data storage.

12.14 Related Modules

- **Counterpart:** stream_drain_ctrl.sv - Drain-side flow control
- **Parent:** sram_controller_unit.sv - Instantiates allocation controller
- **User:** axi_read_engine.sv - Checks rd_space_free before issuing AR

12.15 Related Documentation

- **Drain Controller:** 11_stream_drain_ctrl.md
 - **SRAM Controller Unit:** 09_sram_controller_unit.md
 - **SRAM Controller:** 08_sram_controller.md
 - **AXI Read Engine:** 06_axi_read_engine.md
-

12.16 Revision History

Stream Allocation Controller Revision History

Version	Date	Author	Description
0.90	2025-11-22	seang	Initial block specification
0.91	2026-01-02	seang	Added table captions and figure numbers

Last Updated: 2026-01-02

13 SRAM Controller Specification

Module: sram_controller.sv **Location:** projects/components/stream/rtl/fub/ **Status:** Implemented **Last Updated:** 2025-11-30

13.1 Overview

The SRAM Controller provides per-channel buffering between AXI read and write engines using independent FIFO structures. Each channel has its own FIFO with dedicated allocation controller (write side) and drain controller (read side).

13.1.1 Key Features

- **Per-channel FIFOs:** Independent gaxi_fifo_sync per channel (no segmentation complexity)
- **ID-based routing:** Transaction ID selects channel for write/read operations
- **Allocation controller:** Reserves space before AXI read data arrives
- **Drain controller:** Manages data availability for AXI write engine
- **Latency bridge:** Aligns FIFO read latency (registered output)
- **Saturating counters:** 8-bit space/count reporting per channel

13.1.2 Design Rationale

Why Per-Channel FIFOs Instead of Segmented SRAM?

The implementation uses **independent FIFOs** rather than a monolithic SRAM divided into segments:

Advantages of Per-Channel FIFOs: - **Simpler logic** - FIFOs are standard, well-tested components
- **Better isolation** - Channel failures don't affect others - **Easier timing** - No cross-channel paths or arbitration - **Modular** - Easy to add/remove channels - **No pointer arithmetic** - FIFO handles internally

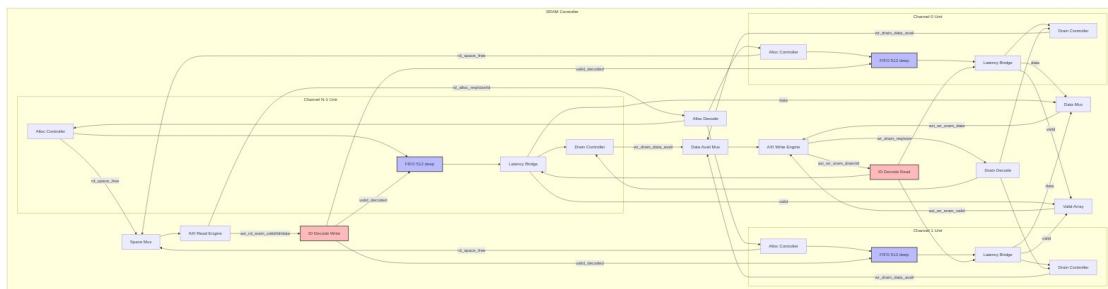
Trade-offs: - More SRAM resources ($NC \times DEPTH$ vs. shared pool) - Potential waste if channels idle (unused FIFO space)

Decision: Simplicity and isolation outweigh SRAM efficiency for STREAM's tutorial focus.

13.2 Architecture

13.2.1 Block Diagram

13.2.2 Figure 2.8.1: SRAM Controller Block Diagram



Diagram

Source: 05_sram_controller_block.mmd

13.2.3 Per-Channel Architecture

Each channel contains three components (in sram_controller_unit):

1. **Allocation Controller (stream_alloc_ctrl):**
 - Receives rd_alloc_req from read engine
 - Tracks reserved vs. committed space
 - Provides rd_space_free to read engine
 2. **FIFO (gaxi_fifo_sync):**
 - Stores data between read and write engines
 - Depth = SRAM_DEPTH parameter
 - Standard valid/ready handshaking
 3. **Drain Controller + Latency Bridge (stream_drain_ctrl):**
 - Receives wr_drain_req from write engine
 - Provides wr_drain_data_avail to write engine
 - Latency bridge aligns FIFO read latency
-

13.3 Parameters

```
parameter int NUM_CHANNELS = 8;                                // Number of
independent channels
parameter int DATA_WIDTH = 512;                                 // Data width in bits
parameter int SRAM_DEPTH = 512;                                // Depth per channel
FIFO
parameter int SEG_COUNT_WIDTH = $clog2(SRAM_DEPTH) + 1;    // Width of
count signals

// Short aliases (internal use)
parameter int NC = NUM_CHANNELS;
parameter int DW = DATA_WIDTH;
parameter int SD = SRAM_DEPTH;
parameter int SCW = SEG_COUNT_WIDTH;                          // FIFO depth counter
width
parameter int CIW = (NC > 1) ? $clog2(NC) : 1;      // Channel ID width
(min 1 bit)
```

Note: “SEG_COUNT_WIDTH” refers to the FIFO depth counter width (historical name from segmented design consideration).

13.4 Port List

13.4.1 Clock and Reset

Clock and Reset

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

13.4.2 Allocation Interface

AXI Read Engine Flow Control (Space Reservation):

Allocation Interface

Signal	Direction	Width	Description
axi_rd_alloc_req	input	1	Allocation request (single request with ID)
axi_rd_alloc_size	input	8	Number of beats to allocate
axi_rd_alloc_id	input	CIW	Channel ID for allocation
axi_rd_alloc_space_free[ch]	output	NUM_CHAN NELS × SEG_COUNT _WIDTH	Free space per channel FIFO

13.4.3 Write Interface

AXI Read Engine → FIFO (ID-based routing):

Write Interface

Signal	Direction	Width	Description
axi_rd_sram_valid	input	1	Write data valid (single valid with ID)
axi_rd_sram_ready	output	1	Ready (muxed from selected channel)
axi_rd_sram_id	input	CIW	Channel ID select for

Signal	Direction	Width	Description
id			write
axi_rd_sram_data	input	DATA_WIDTH	Write data (common bus)

13.4.4 Drain Interface

Write Engine Flow Control (Data Availability):

Drain Interface

Signal	Direction	Width	Description
axi_wr_drain_data_avail[ch]	output	NUM_CHAN NELS × SEG_COUNT _WIDTH	Available data after reservations per channel
axi_wr_drain_req[ch]	input	NUM_CHAN NELS	Per-channel drain request
axi_wr_drain_size[ch]	input	NUM_CHAN NELS × 8	Per-channel drain size (beats to reserve)

13.4.5 Read Interface

FIFO → AXI Write Engine (ID-based routing):

Read Interface

Signal	Direction	Width	Description
axi_wr_sram_valid[ch]	output	NUM_CHAN NELS	Per-channel valid (data available)
axi_wr_sram_drain	input	1	Drain request (consumer ready)
axi_wr_sram_id	input	CIW	Channel ID select for read
axi_wr_sram_data	output	DATA_WIDTH	Read data from selected channel (muxed)

13.4.6 Debug Interface

Debug Interface

Signal	Direction	Width	Description
dbg_bridge_pending[ch]	output	NUM_CHAN NELS	Latency bridge pending per channel
dbg_bridge_output_valid[ch]	output	NUM_CHAN NELS	Latency bridge output valid per channel

13.5 Interface

13.5.1 Clock and Reset

```
input logic clk;
input logic rst_n; // Active-low asynchronous reset
```

13.5.2 Allocation Interface (AXI Read Engine Flow Control)

Space Reservation:

```
input logic axi_rd_alloc_req; // Single allocation request
input logic [7:0] axi_rd_alloc_size; // Beats to allocate
input logic [CIW-1:0] axi_rd_alloc_id; // Channel ID for allocation
output logic [NC-1:0][SCW-1:0] axi_rd_alloc_space_free; // Free space per channel
```

Allocation Protocol: 1. Read engine issues AR transaction 2. **Before R data arrives:** Engine asserts `axi_rd_alloc_req` with size and ID 3. Allocation controller reserves space in selected channel 4. Engine tracks reserved space, prevents over-issuing AR commands 5. Space commits when actual data arrives (FIFO write)

Why Allocation is Critical:

Without pre-allocation, multiple AR commands could be issued before data arrives, causing FIFO overflow:

Problem without allocation:

```
Cycle 0: Issue AR (16 beats) - FIFO has 32 free
Cycle 1: Issue AR (16 beats) - FIFO still shows 32 free (data hasn't arrived!)
Cycle 2: Issue AR (16 beats) - FIFO still shows 32 free
Cycle 10: R data starts arriving (48 beats total)
```

→ OVERFLOW! Only 32 beats of space

Solution with allocation:

Cycle 0: Issue AR (16 beats), allocate 16 - FIFO shows 16 free (reserved)

Cycle 1: Issue AR (16 beats), allocate 16 - FIFO shows 0 free (reserved)

Cycle 2: Cannot issue AR - no space available

Cycle 10: R data arrives - space is guaranteed

13.5.3 Write Interface (AXI Read Engine → FIFO)

ID-Based Write:

```
input logic                                axi_rd_sram_valid;      // Single
valid for all channels
input logic [CIW-1:0]                      axi_rd_sram_id;        // Channel ID
select
output logic                               axi_rd_sram_ready;     // Ready
(muxed from selected channel)
input logic [DW-1:0]                        axi_rd_sram_data;      // Shared data
bus
```

Write Protocol: 1. Read engine asserts axi_rd_sram_valid with data 2. axi_rd_sram_id selects which channel FIFO receives data 3. Controller decodes ID to per-channel valid_decoded[id] 4. Selected channel's FIFO ready muxed to axi_rd_sram_ready

13.5.4 Drain Interface (AXI Write Engine Flow Control)

Data Availability:

```
input logic [NC-1:0]                      axi_wr_drain_req;      //
Per-channel drain request
input logic [NC-1:0][7:0]                   axi_wr_drain_size;      //
Per-channel drain size
output logic [NC-1:0][SCW-1:0]              axi_wr_drain_data_avail;
// Available data after reservations
```

Drain Protocol: 1. Write engine checks axi_wr_drain_data_avail[ch] for available data 2. Engine asserts axi_wr_drain_req[ch] to reserve data for W burst 3. Drain controller updates available count (subtracts reserved) 4. Data commits when actually read from FIFO

Why Drain Reservation is Critical:

Similar to allocation, drain prevents under-reporting data availability:

Problem without drain reservation:

Cycle 0: FIFO has 32 beats available

Cycle 1: Issue AW (16 beats) - FIFO still shows 32 available

Cycle 2: Issue AW (16 beats) - FIFO still shows 32 available

Cycle 3: Issue AW (16 beats) - FIFO still shows 32 available
 Cycle 10: W bursts start draining (48 beats expected)
 → UNDERFLOW! Only 32 beats available

Solution with drain reservation:

Cycle 0: FIFO has 32 beats available
 Cycle 1: Issue AW (16 beats), drain 16 - shows 16 available (reserved)
 Cycle 2: Issue AW (16 beats), drain 16 - shows 0 available (reserved)
 Cycle 3: Cannot issue AW - no data available
 Cycle 10: W bursts drain - data is guaranteed

13.5.5 Read Interface (FIFO → AXI Write Engine)

ID-Based Read:

```
output logic [NC-1:0]           axi_wr_sram_valid;      // Per-channel
valid (data available)        input logic          axi_wr_sram_drain;    // Single
input logic          drain (consumer requests data) input logic [CIW-1:0]   axi_wr_sram_id;       // Channel ID
select                   output logic [DW-1:0]   axi_wr_sram_data;     // Data from
selected channel
```

Read Protocol: 1. Write engine checks `axi_wr_sram_valid[ch]` (per-channel) 2. Engine asserts `axi_wr_sram_drain` with `axi_wr_sram_id` 3. Controller decodes ID to per-channel `drain_decoded[id]` 4. Selected channel's data muxed to `axi_wr_sram_data`

13.5.6 Debug Interface

```
output logic [NC-1:0]           dbg_bridge_pending;    // Latency
bridge pending per channel    output logic [NC-1:0]   dbg_bridge_out_valid; // Latency
bridge output valid per channel
```

Purpose: Monitor latency bridge state to catch bugs in read timing.

13.6 ID Decode Logic

13.6.1 Write Valid Decode

Decode `axi_rd_sram_id` to per-channel valid:

```
always_comb begin
  axi_rd_sram_valid_decoded = '0;
```

```

    if (axi_rd_sram_valid && axi_rd_sram_id < NC) begin
        axi_rd_sram_valid_decoded[axi_rd_sram_id] = 1'b1;
    end
end

```

Mux ready from selected channel:

```

always_comb begin
    if (axi_rd_sram_id < NC) begin
        axi_rd_sram_ready =
    axi_rd_sram_ready_per_channel[axi_rd_sram_id];
    end else begin
        axi_rd_sram_ready = 1'b0; // Invalid ID → not ready
    end
end

```

13.6.2 Read/Drain Decode

Decode axi_wr_sram_id to per-channel drain:

```

always_comb begin
    axi_wr_sram_drain_decoded = '0;
    if (axi_wr_sram_drain && axi_wr_sram_id < NC) begin
        axi_wr_sram_drain_decoded[axi_wr_sram_id] = 1'b1;
    end
end

```

Mux data from selected channel:

```

always_comb begin
    if (axi_wr_sram_id < NC) begin
        axi_wr_sram_data =
    axi_wr_sram_data_per_channel[axi_wr_sram_id];
    end else begin
        axi_wr_sram_data = '0; // Invalid ID → zero data
    end
end

```

13.6.3 Allocation Decode

Decode axi_rd_alloc_id to per-channel allocation:

```

always_comb begin
    axi_rd_alloc_req_decoded = '0;
    if (axi_rd_alloc_req && axi_rd_alloc_id < NC) begin
        axi_rd_alloc_req_decoded[axi_rd_alloc_id] = 1'b1;
    end
end

```

13.7 Per-Channel Unit

Each channel instantiates sram_controller_unit (separate module):

```
sram_controller_unit #(
    .DATA_WIDTH(DW),
    .SRAM_DEPTH(SRAM_DEPTH),
    .SEG_COUNT_WIDTH(SEG_COUNT_WIDTH)
) u_channel_unit (
    .clk                  (clk),
    .rst_n                (rst_n),

    // Write interface (decoded valid from ID)
    .axi_rd_sram_valid   (axi_rd_sram_valid_decoded[i]),
    .axi_rd_sram_ready   (axi_rd_sram_ready_per_channel[i]),
    .axi_rd_sram_data    (axi_rd_sram_data),           // SHARED

    // Read interface (decoded drain)
    .axi_wr_sram_valid   (axi_wr_sram_valid[i]),
    .axi_wr_sram_ready   (axi_wr_sram_drain_decoded[i]),
    .axi_wr_sram_data    (axi_wr_sram_data_per_channel[i]),

    // Allocation interface (decoded req from ID)
    .rd_alloc_req         (axi_rd_alloc_req_decoded[i]),
    .rd_alloc_size         (axi_rd_alloc_size),          // SHARED
    .rd_space_free         (axi_rd_alloc_space_free[i]),

    // Drain interface (per-channel)
    .wr_drain_req         (axi_wr_drain_req[i]),
    .wr_drain_size         (axi_wr_drain_size[i]),
    .wr_drain_data_avail (axi_wr_drain_data_avail[i]),

    // Debug
    .dbg_bridge_pending   (dbg_bridge_pending[i]),
    .dbg_bridge_out_valid (dbg_bridge_out_valid[i])
);
```

Key Points: - axi_rd_sram_data is **shared** - all units see same data bus - Only unit with valid_decoded[i] = 1 writes to its FIFO - rd_alloc_size is **shared** - all see same size value - Only unit with alloc_req_decoded[i] = 1 reserves space

13.8 Operation Flows

13.8.1 Write Flow (AXI Read Data → FIFO)

Allocation Phase (before data arrives):

1. Read engine issues AR command (araddr, arlen, arid)
2. Engine extracts channel ID from arid[CIW-1:0]
3. Engine asserts:
 $\text{axi_rd_alloc_req} = 1$
 $\text{axi_rd_alloc_id} = \text{channel_id}$
 $\text{axi_rd_alloc_size} = \text{arlen} + 1$ (burst size in beats)
4. Allocation controller (in sram_controller_unit):
 - Checks $\text{rd_space_free}[\text{channel_id}] \geq \text{alloc_size}$
 - Reserves space (doesn't commit yet)
5. Engine proceeds with AR transaction

Data Arrival Phase:

1. AXI R data arrives:
 $\text{m_axi_rvalid} = 1$
 $\text{m_axi_rid} = \text{transaction_id}$ (contains channel_id in lower bits)
 $\text{m_axi_rdata} = \text{data}$
2. Read engine forwards to SRAM controller:
 $\text{axi_rd_sram_valid} = 1$
 $\text{axi_rd_sram_id} = \text{rid}[\text{CIW}-1:0]$ (extract channel ID)
 $\text{axi_rd_sram_data} = \text{rdata}$
3. SRAM controller decodes:
 $\text{axi_rd_sram_valid_decoded}[\text{channel_id}] = 1$
4. Selected channel FIFO:
 - Writes data if ready
 - Commits 1 beat of reserved space
 - Decrement reserved counter
5. Ready mux:
 $\text{axi_rd_sram_ready} = \text{fifo_ready}[\text{channel_id}]$

13.8.2 Read Flow (FIFO → AXI Write Data)

Drain Reservation Phase (before W data drains):

1. Write engine checks $\text{drain_data_avail}[\text{channel_id}]$
2. If sufficient data available:
 $\text{axi_wr_drain_req}[\text{channel_id}] = 1$
 $\text{axi_wr_drain_size}[\text{channel_id}] = \text{burst_size}$

3. Drain controller:
 - Reserves data (decrements available count)
 - Prevents other channels from seeing this data

Data Drain Phase:

1. Write engine ready to consume data:

```
axi_wr_sram_drain = 1
axi_wr_sram_id = channel_id
```

2. SRAM controller decodes:

```
axi_wr_sram_drain_decoded[channel_id] = 1
```

3. Selected channel FIFO:

- Reads data (1-cycle latency through bridge)
- Commits 1 beat of reserved data
- Decrements drain reservation

4. Data mux:

```
axi_wr_sram_data = fifo_data[channel_id]
```

5. Valid output:

```
axi_wr_sram_valid[channel_id] = bridge_valid
```

13.9 Allocation vs. Drain Controllers

13.9.1 Why Separate Controllers?

Allocation Controller (Write Side): - Problem: AXI read AR issued before R data arrives - Solution: Pre-allocate space when AR issues, commit when R arrives - Prevents: Over-issuing AR commands when FIFO full

Drain Controller (Read Side): - Problem: AXI write AW issued before W data drains from FIFO - Solution: Reserve data when AW issues, commit when W drains - Prevents: Under-reporting available data when burst in progress

13.9.2 Flow Control Comparison

Read Engine (uses allocation):

```
// Check space BEFORE issuing AR
if (rd_space_free[ch] >= burst_size) begin
    issue_ar_command();
    assert_rd_alloc_req(); // Reserve space
end
// Later: R data arrives → commits reservation
```

Write Engine (uses drain):

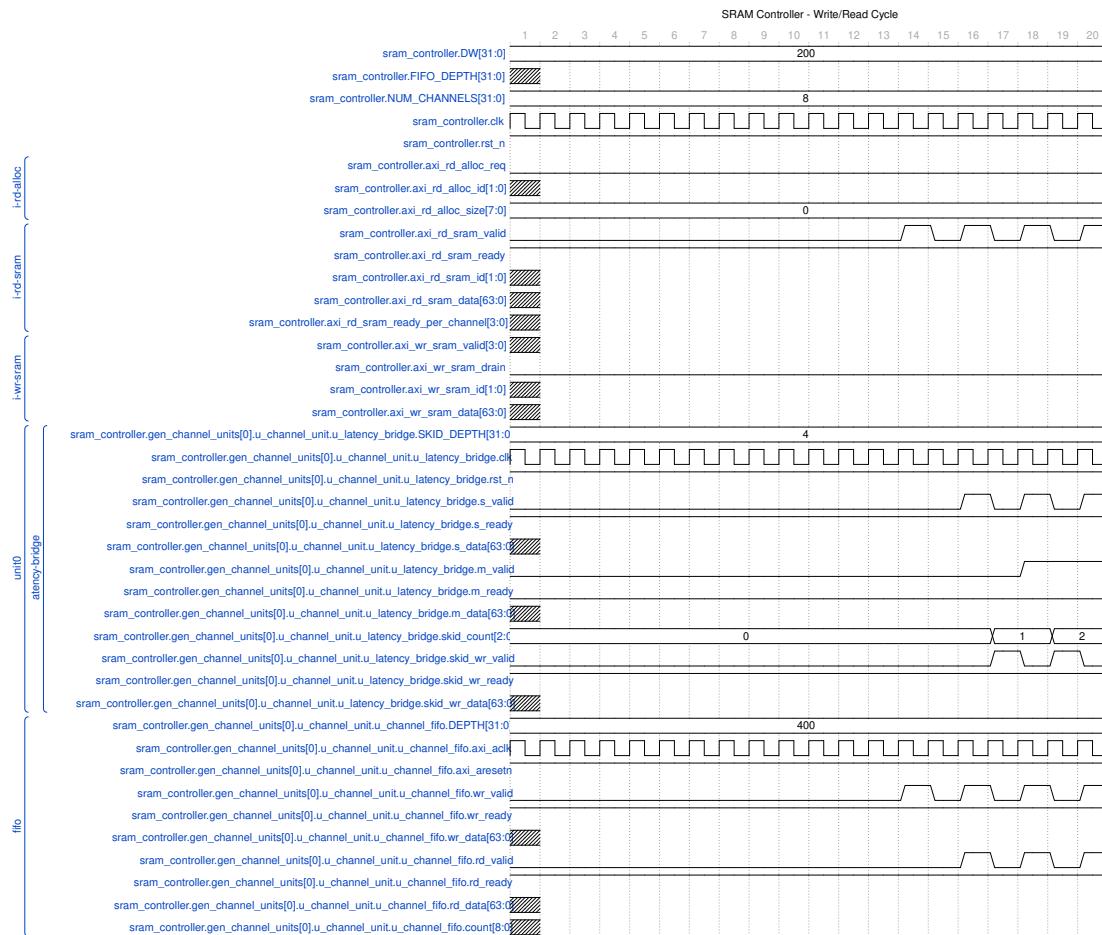
```
// Check data availability BEFORE issuing AW
if (wr_drain_data_avail[ch] >= burst_size) begin
    issue_aw_command();
    assert_wr_drain_req(); // Reserve data
end
// Later: W beats drain → commits reservation
```

13.10 Timing Diagrams

13.10.1 Combined Write and Read Timing

The following diagram shows the complete SRAM controller write and read flow:

13.10.1.1 Waveform 2.8.1: SRAM Controller Write/Read Operation



SRAM Controller Write/Read

Source: [sram_controller_wr_rd.json](#)

13.10.2 Write Path (R Data → FIFO)

Cycle: 0 1 2 3 4 5
| | | | | |

: Write Path
Alloc: [REQ][---grant---]
(ch2, size=4)

R Data: [V][V][V][V]
R ID: [2][2][2][2]
| | | |

: Write Path
Decode: [1][1][1][1] (channel 2)
FIFO: [W][W][W][W]
Ready: [1][1][1][1][1][1][1]...

Notes:

- Allocation at cycle 0 reserves 4 beats
- R data arrives cycles 3-6
- Each beat commits 1 reserved entry

13.10.3 Read Path (FIFO → W Data)

Cycle: 0 1 2 3 4 5 6
| | | | | | |

: Read Path
Drain: [REQ][---reserve---]
(ch2, size=2)

Drain: [D][D]
Drain ID: [2][2]
| | |

: Read Path
Decode: [1][1] (channel 2)
Bridge: [-][V][V] (1-cycle latency)
W Data: [D0][D1]

Notes:

- Drain request at cycle 0 reserves 2 beats
- Actual drain at cycles 3-4
- Latency bridge adds 1 cycle (data at 4-5)

13.11 Error Conditions

13.11.1 Invalid Channel ID

Condition: `axi_rd_sram_id >= NUM_CHANNELS` or `axi_wr_sram_id >= NUM_CHANNELS`

Handling: - Write: `axi_rd_sram_ready = 0` (not ready, data dropped) - Read: `axi_wr_sram_data = 0` (zero data output) - **No error signal** - engines should not generate invalid IDs

13.11.2 FIFO Overflow

Condition: Write when FIFO full

Prevention: - Allocation controller tracks reserved + committed space - Read engine checks `rd_space_free` before issuing AR - Only issues AR when sufficient space available

Should never happen if read engine follows protocol!

13.11.3 FIFO Underflow

Condition: Read when FIFO empty

Prevention: - Drain controller tracks available - reserved data - Write engine checks `wr_drain_data_avail` before issuing AW - Only issues AW when sufficient data available

Should never happen if write engine follows protocol!

13.12 Performance Considerations

13.12.1 Per-Channel FIFO Benefits

Isolation: - Channel failures don't affect others - No cross-channel contention - Simplified debug (each channel independent)

Timing: - No shared arbitration delays - Predictable latency per channel - Easier timing closure

Scalability: - Easy to add/remove channels - Parameterizable depth per channel - No global SRAM redesign needed

13.12.2 SRAM Resource Usage

Total SRAM: $\text{NUM CHANNELS} \times \text{SRAM DEPTH} \times \text{DATA WIDTH}$

Example (8 channels, 512 depth, 512-bit data): - Per channel: $512 \times 512 \text{ bits} = 32\text{KB}$ - Total: $8 \times 32\text{KB} = 256\text{KB}$

Comparison to Segmented Approach: - Segmented: $1 \times (8 \times 512) \times 512 \text{ bits} = 256\text{KB}$ (same total) - But segmented requires complex pointer arithmetic and arbitration - Per-channel FIFOs trade complexity for slight area increase (FIFO overhead)

13.13 Testing

Test Location: `projects/components/stream/dv/tests/fub_tests/sram_controller/`

Key Test Scenarios:

1. **Single channel fill/drain** - Basic FIFO operation
 2. **All channels concurrent** - Independent operation
 3. **ID decode** - Correct channel selection
 4. **Allocation before data** - Space reserved correctly
 5. **Drain before read** - Data reserved correctly
 6. **Latency bridge** - 1-cycle delay verified
 7. **Invalid ID** - Graceful handling
 8. **Back-to-back** - No bubbles between bursts
-

13.14 Related Documentation

- **SRAM Controller Unit:** `09_sram_controller_unit.md` - Per-channel implementation
 - **Allocation Controller:** `07_stream_alloc_ctrl.md` - Write-side flow control
 - **Drain Controller:** `11_stream_drain_ctrl.md` - Read-side flow control
 - **AXI Read Engine:** `06_axi_read_engine.md` - Write interface usage
 - **AXI Write Engine:** `12_axi_write_engine.md` - Read interface usage
 - **Stream Core:** `01_stream_core.md` - Top-level integration
-

13.15 Revision History

Revision History

Date	Version	Changes
2025-10-18	0.5	Initial draft documenting segmented SRAM approach
2025-11-16	1.0	Updated to document per-channel FIFO implementation (actual RTL)
2025-11-21	1.5	Merged documentation:- Consolidated duplicate files- Added design rationale section- Enhanced allocation/drain explanations with examples- Clarified per-channel FIFO architecture- Added performance comparison- Verified all content matches current RTL implementation
2025-11-30	1.6	Updated related documentation references

Last Updated: 2025-11-30 (verified against RTL implementation)

14 SRAM Controller Unit

Module: sram_controller_unit.sv **Location:** projects/components/stream/rtl/fub/
Category: FUB (Functional Unit Block) **Parent:** sram_controller.sv **Status:** Implemented **Last Updated:** 2025-11-30

14.1 Overview

The sram_controller_unit module is a single-channel SRAM controller unit containing allocation controller, FIFO buffer, and latency bridge. It handles one channel's data flow from AXI read engine through buffering to AXI write engine with proper flow control.

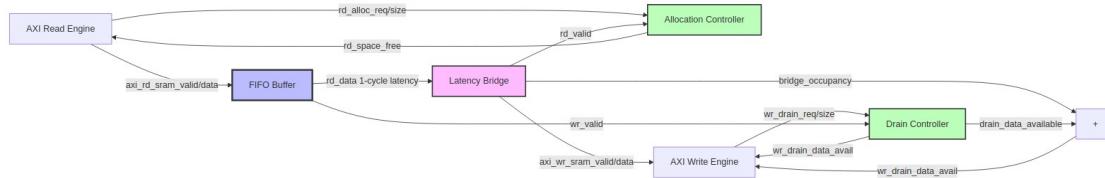
14.1.1 Key Features

- **Three-Component Architecture:**
 - Allocation controller (stream_alloc_ctrl) - Space tracking for reads
 - FIFO buffer (gaxi_fifo_sync) - Physical data storage
 - Latency bridge (stream_latency_bridge) - Timing compensation
 - **Drain Controller:** Tracks data availability for write engine
 - **Virtual FIFO Pattern:** Pointer arithmetic without data storage for flow control
 - **Registered Outputs:** Breaks combinatorial paths for timing closure
-

14.2 Architecture

14.2.1 Block Diagram

14.2.2 Figure 1: SRAM Controller Unit Block Diagram



SRAM Controller Unit Block Diagram

Source: [06_sram_controller_unit_block.mmd](#)

14.2.3 Component Hierarchy

```
sram_controller_unit
  stream_alloc_ctrl          # Allocation tracking (space availability)
  stream_drain_ctrl          # Drain tracking (data availability)
  gaxi_fifo_sync              # Physical data storage FIFO
  stream_latency_bridge       # 1-cycle latency compensation
```

14.2.4 Data Flow

AXI Read Engine → FIFO Write Port → FIFO Storage → FIFO Read Port →
Latency Bridge → AXI Write Engine

Allocation Controller (space tracking)

Drain Controller (data
tracking)

14.2.5 Controller Naming Convention (CRITICAL)

Allocation Controller Perspective: - wr side = ALLOCATE (reserve space, advance wr_ptr) - rd side = FULFILL (data arrives, advance rd_ptr, FREE space)

Drain Controller Perspective: - wr side = DATA WRITTEN (increment occupancy) - rd side = DRAIN REQUEST (reserve data for write burst)

This is OPPOSITE of normal FIFO naming conventions!

14.3 Parameters

Parameters

Parameter	Type	Default	Description
DATA_WIDTH	int	512	Data width in bits
SRAM_DEPTH	int	512	FIFO depth in entries
SEG_COUNT_WIDTH	int	\$clog2(SRAM_DEPTH)+1	Width for count signals

14.3.1 Derived Parameters

Derived Parameters

Parameter	Derivation	Description
DW	DATA_WIDTH	Short alias
SD	SRAM_DEPTH	Short alias
SCW	SEG_COUNT_WIDTH	Segment count width
ADDR_WIDTH	\$clog2(SD)	FIFO address width

14.4 Port List

14.4.1 Clock and Reset

Clock and Reset

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

Signal	Direction	Width	Description
			reset

14.4.2 Allocation Interface (Read Engine Flow Control)

Allocation Interface

Signal	Direction	Width	Description
axi_rd_alloc_req	input	1	Request space allocation
axi_rd_alloc_size	input	8	Beats to reserve
axi_rd_alloc_space_free	output	SCW	Free space available

14.4.3 Write Interface (AXI Read Engine to FIFO)

Write Interface

Signal	Direction	Width	Description
axi_rd_sram_valid	input	1	Write data valid
axi_rd_sram_ready	output	1	Ready to accept data
axi_rd_sram_data	input	DW	Write data payload

14.4.4 Drain Interface (Write Engine Flow Control)

Drain Interface

Signal	Direction	Width	Description
axi_wr_drain_data_avail	output	SCW	Data available for drain
axi_wr_drain_req	input	1	Request to drain data
axi_wr_drain_size	input	8	Beats to drain

14.4.5 Read Interface (FIFO to AXI Write Engine)

Read Interface

Signal	Direction	Width	Description
axi_wr_sram_v alid	output	1	Read data valid
axi_wr_sram_r eady	input	1	Ready to accept data
axi_wr_sram_d ata	output	DW	Read data payload

14.4.6 Debug Interface

Debug Interface

Signal	Direction	Width	Description
dbg_bridge_pe nding	output	1	Data in flight in bridge
dbg_bridge_out t_valid	output	1	Bridge output valid

14.5 Operation

14.5.1 Allocation Flow

1. **Space Check:** Read engine checks `axi_rd_alloc_space_free`
2. **Reservation:** Read engine asserts `axi_rd_alloc_req` with burst size
3. **Space Decrement:** Allocation controller decrements `space_free`
4. **Data Arrival:** Data enters FIFO via `axi_rd_sram_*` interface
5. **Space Release:** When data exits (output handshake), `space_free` increments

14.5.2 Drain Flow

1. **Data Check:** Write engine checks `axi_wr_drain_data_avail`
2. **Reservation:** Write engine asserts `axi_wr_drain_req` with burst size
3. **Data Decrement:** Drain controller decrements `data_available`
4. **Data Consumption:** Write engine reads via `axi_wr_sram_*` interface

14.5.3 Data Available Calculation

```
// Total data available = drain controller data + latency bridge
occupancy
assign axi_wr_drain_data_avail = drain_data_available +
SCW'(bridge_occupancy);
```

The latency bridge can hold 1 beat in flight plus up to 4 beats in its skid buffer, which must be accounted for in the data availability count.

14.6 Integration Example

```
sram_controller_unit #(
    .DATA_WIDTH      (512),
    .SRAM_DEPTH     (512),
    .SEG_COUNT_WIDTH(10)
) u_sram_controller_unit (
    .clk              (clk),
    .rst_n            (rst_n),

    // Allocation interface
    .axi_rd_alloc_req      (axi_rd_alloc_req[ch]),
    .axi_rd_alloc_size     (axi_rd_alloc_size),
    .axi_rd_alloc_space_free(axi_rd_alloc_space_free[ch]),

    // Write interface (from AXI Read Engine)
    .axi_rd_sram_valid     (axi_rd_sram_valid[ch]),
    .axi_rd_sram_ready     (axi_rd_sram_ready[ch]),
    .axi_rd_sram_data      (axi_rd_sram_data),

    // Drain interface
    .axi_wr_drain_data_avail(axi_wr_drain_data_avail[ch]),
    .axi_wr_drain_req       (axi_wr_drain_req[ch]),
    .axi_wr_drain_size      (axi_wr_drain_size[ch]),

    // Read interface (to AXI Write Engine)
    .axi_wr_sram_valid      (axi_wr_sram_valid[ch]),
    .axi_wr_sram_ready      (axi_wr_sram_ready[ch]),
    .axi_wr_sram_data        (axi_wr_sram_data[ch]),

    // Debug
    .dbg_bridge_pending     (dbg_bridge_pending[ch]),
    .dbg_bridge_out_valid   (dbg_bridge_out_valid[ch])
);
```

14.7 Common Issues

14.7.1 Issue 1: Space Accounting Mismatch

Symptom: Read engine sees space but FIFO overflows

Root Causes: 1. Allocation controller not receiving fulfillment signals 2. Output handshake not connected to allocation controller rd_valid

Solution: Ensure axi_wr_sram_valid && axi_wr_sram_ready connects to allocation controller.

14.7.2 Issue 2: Data Available Undercount

Symptom: Write engine stalls despite data in FIFO

Root Causes: 1. Bridge occupancy not included in data_available calculation 2. Drain controller not receiving write handshakes

Solution: Verify axi_wr_drain_data_avail = drain_data_available + bridge_occupancy.

14.8 Related Documentation

- **Parent:** 08_sram_controller.md - Multi-channel SRAM controller
 - **Allocation Controller:** 07_stream_alloc_ctrl.md - Space tracking details
 - **Drain Controller:** 11_stream_drain_ctrl.md - Data tracking details
 - **Latency Bridge:** 10_stream_latency_bridge.md - Timing compensation
 - **Read Engine:** 06_axi_read_engine.md - Data producer
 - **Write Engine:** [12_axi_write_engine.md - Data consumer](#)
-

14.9 Revision History

SRAM Controller Unit Revision History

Version	Date	Author	Description
0.90	2025-11-22	seang	Initial block specification
0.91	2026-01-02	seang	Added table

Version	Date	Author	Description
			captions and figure numbers

Last Updated: 2026-01-02

15 Stream Latency Bridge

Module: stream_latency_bridge.sv **Location:** projects/components/stream/rtl/fub/
Category: FUB (Functional Unit Block) **Parent:** sram_controller_unit.sv **Status:** Implemented
Last Updated: 2025-11-30

15.1 Overview

The stream_latency_bridge module is a simple latency-1 bridge that compensates for registered FIFO output latency. It uses glue logic plus a skid buffer to handle backpressure from downstream consumers.

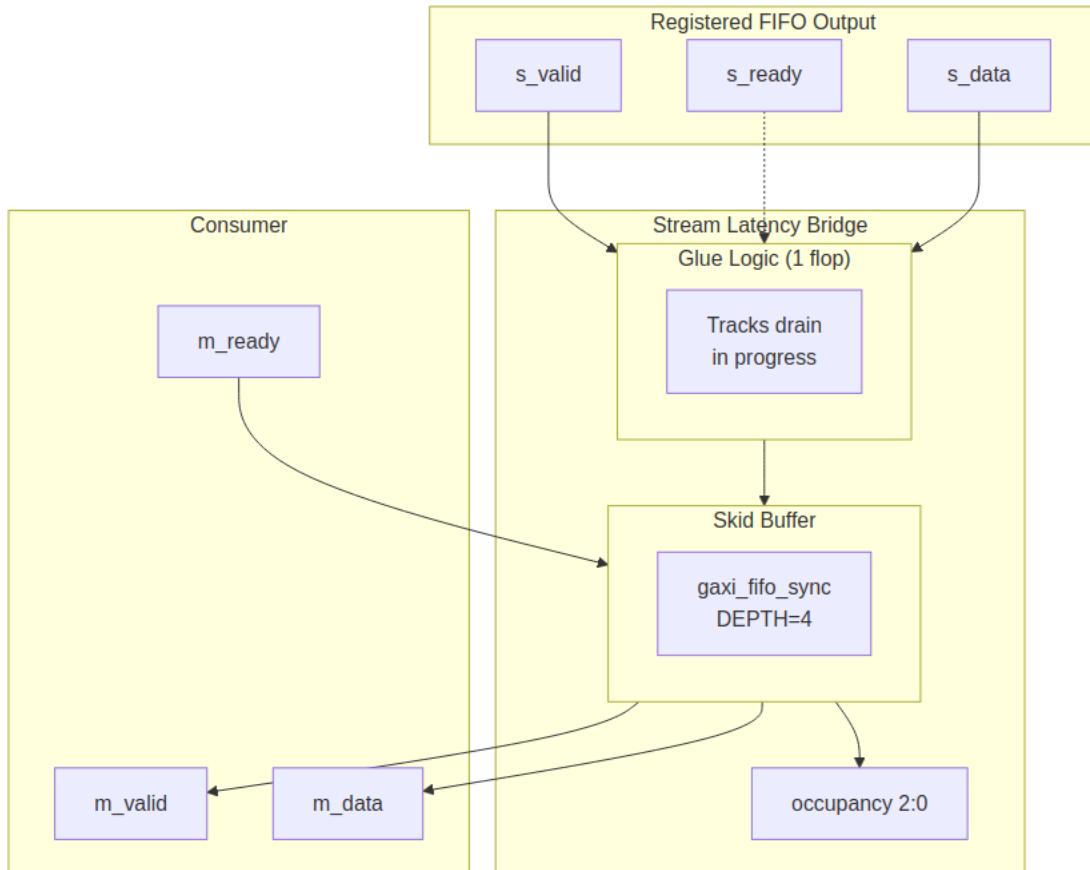
15.1.1 Key Features

- **1-Cycle Latency Compensation:** Bridges registered FIFO output to consumer
 - **Skid Buffer Architecture:** 4-deep FIFO absorbs consumer backpressure
 - **Full Throughput:** Maintains back-to-back transfers when consumer is ready
 - **Simple Design:** No complex ready calculations - skid buffer handles it all
 - **Occupancy Tracking:** Reports total beats held for flow control
-

15.2 Architecture

15.2.1 Block Diagram

15.2.2 Figure 2.10.1: Stream Latency Bridge Block Diagram



Stream Latency Bridge Block Diagram

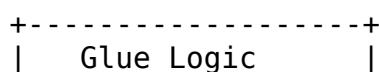
Source: [06_stream_latency_bridge_block.mmd](#)

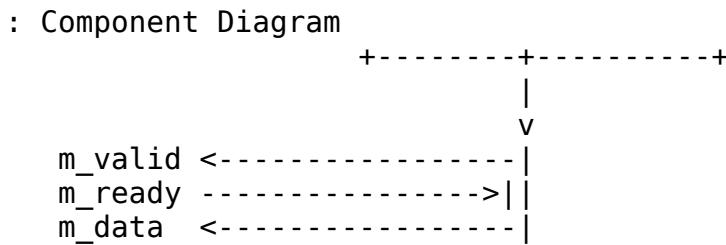
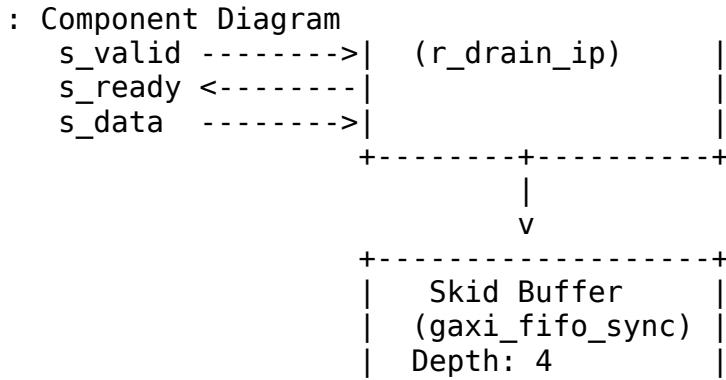
15.2.3 Operation Flow

Cycle 0: Drain FIFO (`s_valid && skid_ready`) \Rightarrow `r_drain_ip = 1`
Cycle 1: Data arrives from FIFO \Rightarrow Push to skid (`skid_valid = r_drain_ip`)

Cycle N: Consumer drains skid buffer at its own pace

15.2.4 Component Diagram





15.2.5 Backpressure Logic

The bridge uses a conservative backpressure approach:

- Track stalled writes (`wr_valid && !wr_ready`)
- Calculate pending count = `skid_count + stalled`
- Allow accept when pending < `SKID_DEPTH` OR consumer is draining

Truth Table (SKID_DEPTH=4):

count	wr_valid	wr_ready	stalled	pending	room	s_ready	
0	0	1	0	0	1	1	?
Empty							
0	1	1	0	0	1	1	?
Writing, completes							
3	0	1	0	3	1	1	?
for 1							
3	1	1	0	3	1	1	?
Writing, will be 4							
3	1	0	1	4	0	0	?
Write stalled							
4	0	0	0	4	0	0	?
Full							

15.3 Parameters

Parameters

Parameter	Type	Default	Description
DATA_WIDTH	int	64	Data width in bits
SKID_DEPTH	int	4	Skid buffer depth (2-4 recommended)

15.3.1 Derived Parameters

Derived Parameters

Parameter	Derivation	Description
DW	DATA_WIDTH	Short alias

15.4 Port List

15.4.1 Clock and Reset

Clock and Reset

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

15.4.2 Upstream Interface (from registered FIFO)

Upstream Interface

Signal	Direction	Width	Description
s_valid	input	1	Upstream valid (FIFO not empty)
s_ready	output	1	Ready to accept from FIFO
s_data	input	DW	Data from FIFO (valid 1 cycle after handshake)

15.4.3 Downstream Interface (to consumer)

Downstream Interface

Signal	Direction	Width	Description
m_valid	output	1	Data valid to consumer
m_ready	input	1	Consumer ready
m_data	output	DW	Data to consumer

15.4.4 Status Interface

Status Interface

Signal	Direction	Width	Description
occupancy	output	3	Beats in bridge (0-5: 1 in flight + 4 in skid)

15.4.5 Debug Interface

Debug Interface

Signal	Direction	Width	Description
dbg_r_pending	output	1	Data in flight (r_drain_ip)
dbg_r_out_val_id	output	1	Output valid (m_valid)

15.5 Operation

15.5.1 Glue Logic

The glue logic consists of a single flop (r_drain_ip) that tracks when a FIFO drain is in progress:

```
// Drain FIFO when upstream has data AND we can accept
wire w_drain_fifo = s_valid && s_ready;

// Flop the drain signal to track data in flight
always_ff @(posedge clk or negedge rst_n) begin
    if (!rst_n) begin
        r_drain_ip <= 1'b0;
```

```

    end else begin
        r_drain_ip <= w_drain_fifo;
    end
end

// When r_drain_ip asserted, data arrived from FIFO & push to skid
assign skid_wr_valid = r_drain_ip;
assign skid_wr_data = s_data;

```

15.5.2 Skid Buffer

A 4-deep gaxi_fifo_sync instance with:

- Non-registered mode (REGISTERED=0) for minimum latency
- Auto memory style selection

15.5.3 Occupancy Calculation

```

// Total occupancy = data in skid buffer
// Note: r_drain_ip not counted separately (conservative)
assign occupancy = skid_count;

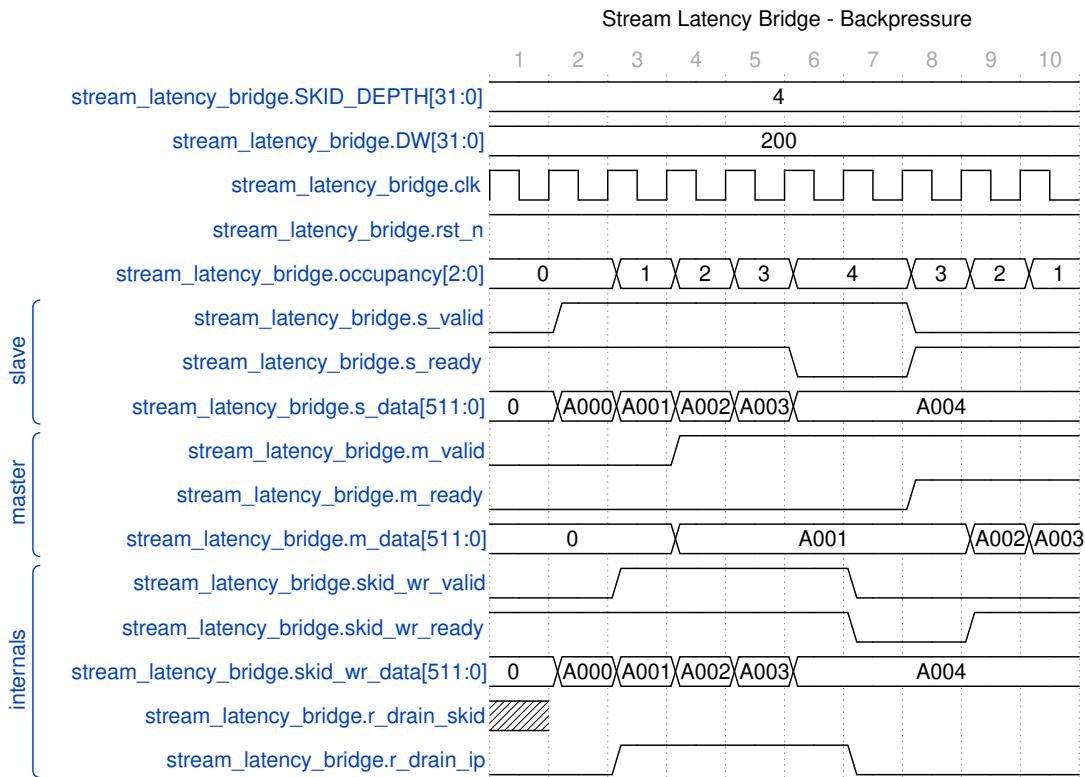
```

15.6 Timing Diagrams

15.6.1 Backpressure Handling

The following diagram shows how the latency bridge handles downstream backpressure:

15.6.1.1 Waveform 2.10.1: Stream Latency Bridge Backpressure



Latency Bridge Backpressure

Source: [latency_bridge_backpressure.json](#)

15.7 Integration Example

```
stream_latency_bridge #(
    .DATA_WIDTH (512),
    .SKID_DEPTH (4)
) u_latency_bridge (
    .clk          (clk),
    .rst_n        (rst_n),
    // Slave (from FIFO)
    .s_data       (fifo_rd_data),
    .s_valid      (fifo_rd_valid),
    .s_ready      (fifo_rd_ready),
    // Master (to AXI Write Engine)
    .m_data       (axi_wr_sram_data),
```

```

.m_valid      (axi_wr_sram_valid),
.m_ready     (axi_wr_sram_ready),

// Status
.occupancy   (bridge_occupancy),

// Debug
.dbg_r_pending (dbg_bridge_pending),
.dbg_r_out_valid(dbg_bridge_out_valid)
);

```

15.8 Design Rationale

15.8.1 Why Use a Skid Buffer?

Previous designs attempted complex ready signal calculations to handle backpressure. This led to:

- Timing issues from deep combinatorial paths
- Edge cases with multi-cycle backpressure
- Difficult debugging of flow control bugs

The skid buffer approach:

- Simple glue logic (single flop)
- FIFO handles all backpressure complexity
- Proven, reusable component
- Easy to verify and debug

15.8.2 Why 4-Deep?

- **2-deep:** Minimum for back-to-back (1 in flight + 1 buffered)
- **4-deep:** Allows 2-3 cycles of consumer stall without upstream backpressure
- **8-deep:** Overkill for most use cases

4-deep provides good balance between:

- Latency tolerance (handles brief stalls)
- Resource usage (small FIFO)
- Throughput (back-to-back capable)

15.9 Related Documentation

- **Parent:** 09_sram_controller_unit.md - Single-channel SRAM controller
- **Consumer:** 12_axi_write_engine.md - Receives bridge output

- [**Producer: 06_axi_read_engine.md - Feeds upstream FIFO**](#)
-

15.10 Revision History

Stream Latency Bridge Revision History

Version	Date	Author	Description
0.90	2025-11-22	seang	Initial block specification
0.91	2026-01-02	seang	Added table captions and figure numbers

Last Updated: 2026-01-02

16 Stream Drain Controller

Module: stream_ctrl.sv **Location:** projects/components/stream/rtl/fub/
Category: FUB (Functional Unit Block) **Parent:** sram_controller_unit.sv **Status:** Implemented
Last Updated: 2025-11-30

16.1 Overview

The stream_ctrl module is a virtual FIFO for write engine flow control. It tracks data availability using FIFO pointer logic without storing any data. This allows the write engine to pre-reserve data before issuing AXI write commands.

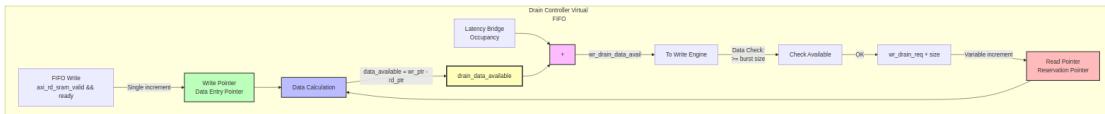
16.1.1 Key Features

- **Virtual FIFO Pattern:** Pointer arithmetic only, no data storage
 - **Pre-Reservation:** Write engine reserves data before AXI AW command
 - **Variable Size Drains:** Supports burst-sized reservations (not single beat)
 - **Registered Outputs:** Optional output registration for timing
 - **Underflow Prevention:** Ensures data exists before write engine commits
-

16.2 Architecture

16.2.1 Block Diagram

16.2.2 Figure 1: Stream Drain Controller Block Diagram



Stream Drain Controller Block Diagram

Source: [09_stream_drain_ctrl_block.mmd](#)

16.2.3 Virtual FIFO Concept

Unlike a real FIFO that stores data, the drain controller only tracks pointers:

Real FIFO: Virtual FIFO (Drain Controller):

Data[0]	wr_ptr: 5	Tracks entries written
Data[1]	rd_ptr: 2	Tracks entries drained
Data[2]	available:3	Difference
...		

Actual data stored in gaxi_fifo_sync (separate component)

16.2.4 Pointer Operations

Write Side (Data Entry):

When FIFO receives data:

```
wr_ptr += 1  
data_available += 1
```

Read Side (Drain Request):

When write engine requests drain:

```
rd_ptr += rd_size (variable burst size!)  
data_available -= rd_size
```

16.3 Parameters

Parameters

Parameter	Type	Default	Description
DEPTH	int	512	Virtual FIFO depth
ALMOST_WR_MAR_GIN	int	1	Almost full margin
ALMOST_RD_MAR_GIN	int	1	Almost empty margin
REGISTERED	int	1	Registered outputs

16.3.1 Derived Parameters

Derived Parameters

Parameter	Derivation	Description
D	DEPTH	Short alias
AW	\$clog2(D)	Address width

16.4 Port List

16.4.1 Clock and Reset

Clock and Reset

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

16.4.2 Write Interface (Data Entry)

Write Interface

Signal	Direction	Width	Description
wr_valid	input	1	Data written to FIFO

Signal	Direction	Width	Description
wr_ready	output	1	Not full (can accept)

16.4.3 Read Interface (Drain Requests)

Read Interface

Signal	Direction	Width	Description
rd_valid	input	1	Request to drain data
rd_size	input	8	Number of entries to drain
rd_ready	output	1	Data available (not empty)

16.4.4 Status Outputs

Status Outputs

Signal	Direction	Width	Description
data_available	output	AW+1	Available data count
wr_full	output	1	Full (no space)
wr_almost_full	output	1	Almost full
rd_empty	output	1	Empty (no data)
rd_almost_empty	output	1	Almost empty

16.5 Operation

16.5.1 Data Entry (Write Side)

Data entry is single-beat increments, tracking each beat written to the physical FIFO:

```
// Write pointer uses counter_bin for single-beat increments
counter_bin #(
    .WIDTH (AW + 1),
```

```

    .MAX      (D)
) write_pointer_inst (
    .clk          (axi_aclk),
    .rst_n        (axi_aresetn),
    .enable       (w_write && !r_wr_full),
    .counter_bin_curr (r_wr_ptr_bin),
    .counter_bin_next (w_wr_ptr_bin_next)
);

```

16.5.2 Drain Requests (Read Side)

Drain requests support variable-size bursts:

```

// Read pointer advances by rd_size on each drain request
always_ff @(posedge axi_aclk or negedge axi_aresetn) begin
    if (!axi_aresetn) begin
        r_rd_ptr_bin <= '0;
    end else begin
        if (w_read && !r_rd_empty) begin
            r_rd_ptr_bin <= r_rd_ptr_bin + (AW+1)'(rd_size);
        end
    end
end

```

16.5.3 Status Generation

Uses fifo_control block for full/empty/almost flags:

```

fifo_control #(
    .DEPTH           (D),
    .ADDR_WIDTH     (AW),
    .ALMOST_RD_MARGIN (ALMOST_RD_MARGIN),
    .ALMOST_WR_MARGIN (ALMOST_WR_MARGIN),
    .REGISTERED     (REGISTERED)
) fifo_control_inst (
    // ... pointer inputs
    .count          (w_count),
    .wr_full        (r_wr_full),
    .wr_almost_full (r_wr_almost_full),
    .rd_empty       (r_rd_empty),
    .rd_almost_empty (r_rd_almost_empty)
);

```

16.6 Comparison with stream_alloc_ctrl

Comparison with stream_alloc_ctrl

Aspect	stream_alloc_ctrl	stream_drain_ctrl
Purpose	Track space for reads	Track data for writes
Write Side	Reservation (burst)	Data entry (single beat)
Read Side	Fulfillment (single beat)	Drain request (burst)
Consumer	AXI Read Engine	AXI Write Engine

Both use virtual FIFO pattern but with opposite semantics for read/write operations.

16.7 Integration Example

```
stream_drain_ctrl #(
    .DEPTH          (512),
    .REGISTERED    (1)
) u_drain_ctrl (
    .axi_aclk      (clk),
    .axi_arstn     (rst_n),

    // Write interface: Connected to FIFO write handshake
    .wr_valid      (fifo_wr_valid && fifo_wr_ready),
    .wr_ready      (), // Not used (tracks, doesn't control)

    // Read interface: Connected to write engine drain requests
    .rd_valid      (axi_wr_drain_req),
    .rd_size       (axi_wr_drain_size),
    .rd_ready      (), // Not used (polling interface)

    // Status
    .data_available (drain_data_available),
    .wr_full       (),
    .wr_almost_full(),
    .rd_empty      (),
    .rd_almost_empty()
);
```

16.8 Common Issues

16.8.1 Issue 1: Data Available Undercount

Symptom: Write engine sees 0 available when FIFO has data

Root Causes: 1. wr_valid not connected to FIFO write handshake 2. Bridge occupancy not added to data_available

Solution: Ensure connection: `wr_valid = fifo_wr_valid && fifo_wr_ready`

16.8.2 Issue 2: Drain Request Overflow

Symptom: rd_ptr advances past wr_ptr

Root Causes: 1. Write engine requesting more than available 2. Multiple simultaneous drain requests

Solution: Write engine must check `data_available >= requested_size` before draining.

16.9 Related Documentation

- **Parent:** 09_sram_controller_unit.md - Integration context
 - **Counterpart:** 07_stream_alloc_ctrl.md - Space tracking (read side)
 - **Consumer:** 12_axi_write_engine.md - Uses drain interface
 - **Control Block: See fifo_control in rtl/amba/gaxi/**
-

16.10 Revision History

Stream Drain Controller Revision History

Version	Date	Author	Description
0.90	2025-11-22	seang	Initial block specification
0.91	2026-01-02	seang	Added table captions and figure numbers

Last Updated: 2026-01-02

17 AXI Write Engine

Module: axi_write_engine.sv **Location:** projects/components/stream/rtl/fub/
Category: FUB (Functional Unit Block) **Parent:** stream_core.sv **Status:** Implemented **Last Updated:** 2025-11-30

17.1 Overview

The `axi_write_engine` module is a high-performance multi-channel AXI4 write engine with data-aware arbitration. It serves all 8 STREAM channels through a single AXI master interface with intelligent flow control.

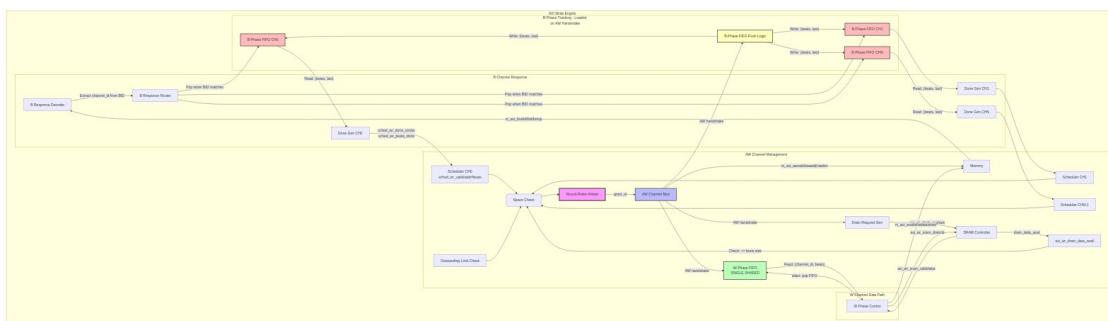
17.1.1 Key Features

- **Round-Robin Arbitration:** Fair scheduling across channels
- **Data-Aware Masking:** Only arbitrate channels with sufficient SRAM data
- **Pre-Drain Handshake:** Reserve SRAM data before AW command issues
- **FIFO-Based W Tracking:** Single shared FIFO for W-phase order preservation
- **Per-Channel B Tracking:** Separate FIFOs for out-of-order B responses
- **Channel ID in AXI ID:** Enables per-channel response routing
- **Pipelined/Non-Pipelined Modes:** Configurable outstanding transaction depth

17.2 Architecture

17.2.1 Block Diagram

17.2.2 Figure 2.12.1: AXI Write Engine Block Diagram



AXI Write Engine Block Diagram

Source: [10_axi_write_engine_block.mmd](#)

17.2.3 Operation Flow

1. Scheduler Interface: Each channel can request write bursts
2. Data Checking: Mask channels without sufficient SRAM data
3. Arbitration: Round-robin arbiter selects next channel to service
4. AXI AW Issue: Issue write command to AXI, assert wr_drain to reserve data
5. W-Phase FIFO: Push transaction to shared FIFO (preserves AW order)
6. AXI W Stream: Stream write data from SRAM controller (ID-based mux)
7. AXI B Response: Pop per-channel B-phase FIFO, complete transaction

17.2.4 Key Design Decisions

FIFO-Based W Tracking (No FSM): Instead of a state machine, uses a single shared FIFO to track pending W-phase transactions. This preserves the AW command order which is critical for W-phase correctness.

Separate W and B Phase FIFOs: - **W-Phase FIFO:** Single shared FIFO (in-order with AW) - **B-Phase FIFOs:** Per-channel FIFOs (out-of-order responses allowed by AXI4)

ID-Based SRAM Interface: The write engine drives `axi_wr_sram_id` to select which channel's data to drain. The SRAM controller muxes the appropriate channel's data to `axi_wr_sram_data`.

17.3 Parameters

Parameters

Parameter	Type	Default	Description
NUM_CHANNELS	int	8	Number of channels
ADDR_WIDTH	int	64	AXI address width
DATA_WIDTH	int	512	AXI data width
ID_WIDTH	int	8	AXI ID width
USER_WIDTH	int	8	AXI USER width
SEG_COUNT_WIDTH	int	8	Width of space/count signals
PIPELINE	int	0	0: non-pipelined, 1: pipelined
AW_MAX_OUTSTANDING	int	8	Maximum outstanding AW requests

Parameter	Type	Default	Description
			(PIPELINE=1)
W_PHASE_FIFO_DEPTH	int	64	W-phase transaction FIFO depth
B_PHASE_FIFO_DEPTH	int	16	B-phase transaction FIFO depth per channel

17.3.1 Derived Parameters

Derived Parameters

Parameter	Derivation	Description
NC	NUM_CHANNELS	Short alias
AW	ADDR_WIDTH	Short alias
DW	DATA_WIDTH	Short alias
IW	ID_WIDTH	Short alias
UW	USER_WIDTH	Short alias
SCW	SEG_COUNT_WIDTH	Segment count width
CIW	\$clog2(NC)	Channel ID width (min 1 bit)
AXSIZE	\$clog2(DW/8)	AXI burst size

17.4 Port List

17.4.1 Clock and Reset

Clock and Reset

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

17.4.2 Configuration Interface

Configuration Interface

Signal	Direction	Width	Description
cfg_axi_wr_xfer_beats	input	8	Transfer size in beats (all channels)

17.4.3 Scheduler Interface (Per-Channel)

Scheduler Interface

Signal	Direction	Width	Description
sched_wr_val_id[ch]	input	NC	Channel requests write
sched_wr_ready[ch]	output	NC	Engine ready (descriptor complete)
sched_wr_addr[ch]	input	NC x AW	Destination addresses
sched_wr_beats[ch]	input	NC x 32	Beats remaining to write
sched_wr_burst_len[ch]	input	NC x 8	Requested burst length

17.4.4 Completion Interface (Per-Channel)

Completion Interface

Signal	Direction	Width	Description
sched_wr_done_strobe[ch]	output	NC	Burst completed (1 cycle pulse)
sched_wr_beats_done[ch]	output	NC x 32	Number of beats completed

17.4.5 SRAM Drain Interface

SRAM Drain Interface

Signal	Direction	Width	Description
axi_wr_drain_req[ch]	output	NC	Request to reserve data
axi_wr_drain_size[ch]	output	NC x 8	Beats to reserve
axi_wr_drain_data_avail	input	NC x SCW	Data available per

Signal	Direction	Width	Description
ch]			channel

17.4.6 SRAM Read Interface

SRAM Read Interface

Signal	Direction	Width	Description
axi_wr_sram_v alid[ch]	input	NC	Per-channel valid
axi_wr_sram_d rain	output	1	Drain request
axi_wr_sram_i d	output	CIW	Channel ID select
axi_wr_sram_d ata	input	DW	Muxed data from selected channel

17.4.7 AXI4 AW Channel

AXI4 AW Channel

Signal	Direction	Width	Description
m_axi_awvalid	output	1	Address valid
m_axi_awready	input	1	Address ready
m_axi_awid	output	IW	Transaction ID
m_axi_awaddr	output	AW	Address
m_axi_awlen	output	8	Burst length - 1
m_axi_awsize	output	3	Burst size (log2 bytes)
m_axi_awburst	output	2	Burst type (INCR)

17.4.8 AXI4 W Channel

AXI4 W Channel

Signal	Direction	Width	Description
m_axi_wvalid	output	1	Write data valid

Signal	Direction	Width	Description
m_axi_wready	input	1	Write data ready
m_axi_wdata	output	DW	Write data
m_axi_wstrb	output	DW/8	Write strobes
m_axi_wlast	output	1	Last beat of burst
m_axi_wuser	output	UW	Channel ID for tracking

17.4.9 AXI4 B Channel

AXI4 B Channel

Signal	Direction	Width	Description
m_axi_bvalid	input	1	Response valid
m_axi_bready	output	1	Response ready
m_axi_bid	input	IW	Transaction ID
m_axi_bresp	input	2	Response

17.4.10 Error Interface

Error Interface

Signal	Direction	Width	Description
sched_wr_error[r[ch]]	output	NC	Sticky error flag per channel

17.4.11 Debug Interface

Debug Interface

Signal	Direction	Width	Description
dbg_wr_all_com plete[ch]	output	NC	All writes complete
dbg_aw_transactions	output	32	Total AW transactions issued

Signal	Direction	Width	Description
dbg_w_beats	output	32	Total W beats written to AXI

17.5 Operation

17.5.1 Data-Aware Request Masking

```
// Only request arbitration if:
// 1. Scheduler is requesting (sched_wr_valid)
// 2. Sufficient SRAM data available (w_data_ok)
// 3. Below outstanding limit (w_no_outstanding)
w_arb_request[i] = sched_wr_valid[i] && w_data_ok[i] &&
w_no_outstanding[i];

// Data check includes final burst handling
w_data_ok[i] = w_has_data[i] || w_final_burst[i];
```

17.5.2 W-Phase Transaction FIFO

Single shared FIFO preserves AW command order:

```
typedef struct packed {
    logic [7:0]    beats;          // Number of beats for this W
transaction
    logic [CIW-1:0] channel_id;   // Channel ID for this transaction
} w_phase_txn_t;

// Push on AW handshake (preserves order)
if (m_axi_awvalid && m_axi_awready) begin
    w_phase_txn_fifo_wr = 1'b1;
    w_phase_txn_fifo_din.beats = m_axi_awlen + 8'd1;
    w_phase_txn_fifo_din.channel_id = r_aw_channel_id;
end
```

17.5.3 B-Phase Transaction FIFOs

Per-channel FIFOs track beats and last flag for completion:

```
typedef struct packed {
    logic [7:0]    beats;          // Number of beats in this transaction
    logic          last;           // Is this the last transfer for
descriptor?
} b_phase_txn_t;

// last flag determines when sched_wr_ready asserts
```

```

if (m_axi_bvalid && m_axi_bready) begin
    if (b_phase_txn_fifo_dout[ch_id].last) begin
        r_sched_ready[ch_id] <= 1'b1;
    end
end

```

17.5.4 Outstanding Transaction Tracking

PIPELINE=0 (Non-Pipelined): - Boolean flag per channel (0 or 1 outstanding) - Set when AW issues, clear when B arrives

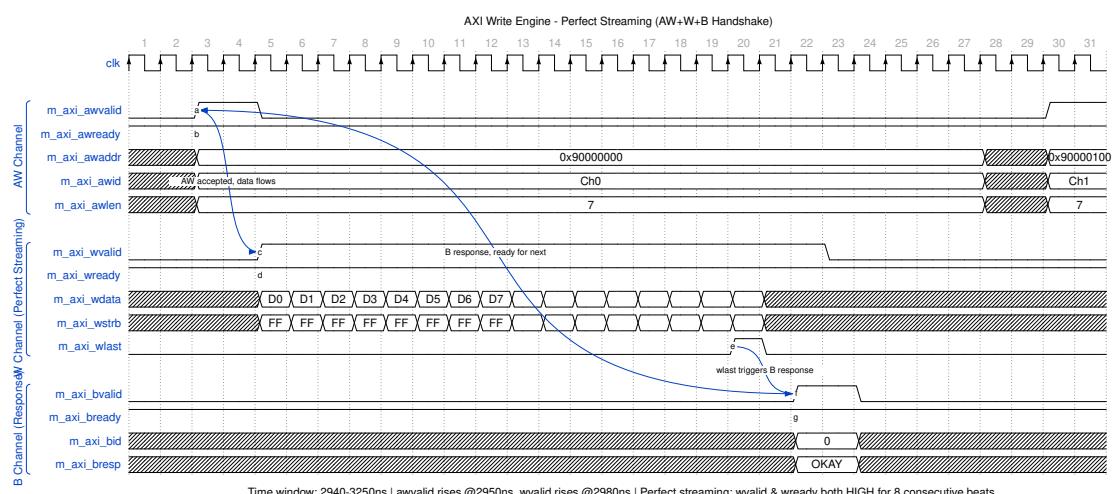
PIPELINE=1 (Pipelined): - Counter per channel (0 to AW_MAX_OUTSTANDING) - Increment on AW, decrement on B

17.6 Timing Diagrams

17.6.1 Perfect Streaming - AXI Write Transaction

The following timing diagram shows the AXI write engine operating at maximum throughput with **perfect streaming** - where both wvalid and wready remain HIGH for consecutive clock cycles, achieving one data beat per cycle.

17.6.1.1 Waveform 2.12.1: AXI Write Engine - Perfect Streaming



AXI Write Engine - Perfect Streaming

Transaction Flow:

1. AW Channel Handshake (Address Phase)

- m_axi_awvalid rises to initiate a write request

- `m_axi_awready` is already HIGH (slave ready)
- Address (`awaddr`), ID (`awid`), and length (`awlen=7` for 8 beats) are captured
- Handshake completes in a single cycle when both valid and ready are HIGH
- Simultaneously, `axi_wr_drain_req` reserves data from SRAM controller

2. W Channel Streaming (Data Phase)

- After AW acceptance, the engine begins streaming data from SRAM
- `m_axi_wvalid` rises and **stays HIGH** for all 8 beats
- `m_axi_wready` remains HIGH throughout (no backpressure from AXI slave)
- **Perfect streaming:** One data beat transferred every clock cycle
- `m_axi_wlast` rises on the final beat (beat 7) to mark burst completion
- `wstrb` is all-ones (0xFF) indicating full data width valid

3. B Channel Response (Completion Phase)

- After `wlast`, the AXI slave returns a write response
- `m_axi_bvalid` rises with `bid` matching the transaction ID
- `m_axi_bready` is HIGH (engine always ready for responses)
- `bresp=OKAY` indicates successful write
- Response triggers `sched_wr_done_strobe` to notify scheduler

4. Next Transaction

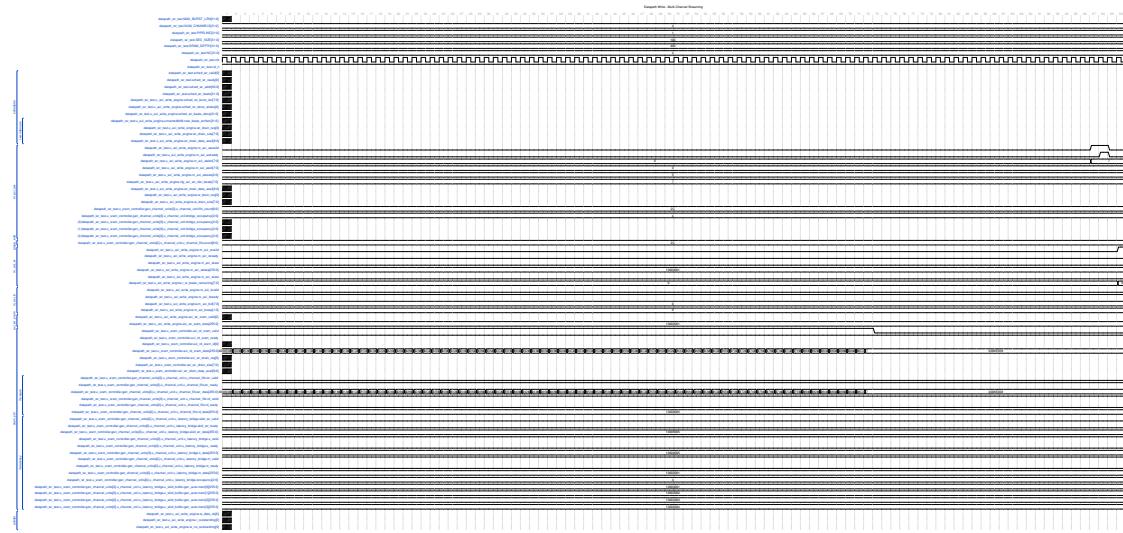
- After B response, the engine can immediately issue the next AW request
- The cycle repeats for the next channel or next burst

Key Performance Indicators: - **No bubbles:** `wvalid` and `wready` both HIGH during data phase - **Full bandwidth:** Data width (256/512 bits) x clock frequency - **Zero-wait states:** SRAM controller provides data at line rate - **Minimal latency:** B response arrives shortly after `wlast`

17.6.2 Multi-Channel Streaming

For multi-channel operation showing channel switching while maintaining streaming performance, see:

17.6.2.1 Waveform 2.12.2: Datapath Write - Multi-Channel



Datapath Write - Multi-Channel

This diagram shows how the engine arbitrates between channels while maintaining high throughput. Note how:

- Different channel IDs (awid) appear in sequence
- W-phase FIFO preserves AW order for correct data association
- B responses can arrive out-of-order (per AXI4 spec)

17.7 Integration Example

```
axi_write_engine #(
    .NUM_CHANNELS      (8),
    .ADDR_WIDTH        (64),
    .DATA_WIDTH        (512),
    .ID_WIDTH          (8),
    .SEG_COUNT_WIDTH   (10),
    .PIPELINE           (0),
    .AW_MAX_OUTSTANDING (8),
    .W_PHASE_FIFO_DEPTH (64),
    .B_PHASE_FIFO_DEPTH (16)
) u_axi_write_engine (
    .clk                (clk),
    .rst_n              (rst_n),
    // Configuration
    .cfg_axi_wr_xfer_beats (cfg_axi_wr_xfer_beats),
    // Scheduler interface
    .sched_wr_valid     (sched_wr_valid),
```

```

.sched_wr_ready          (sched_wr_ready),
.sched_wr_addr           (sched_wr_addr),
.sched_wr_beats          (sched_wr_beats),
.sched_wr_burst_len      (sched_wr_burst_len),

// Completion interface
.sched_wr_done_strobe    (sched_wr_done_strobe),
.sched_wr_beats_done     (sched_wr_beats_done),

// SRAM drain interface
.axi_wr_drain_req        (axi_wr_drain_req),
.axi_wr_drain_size       (axi_wr_drain_size),
.axi_wr_drain_data_avail(axi_wr_drain_data_avail),

// SRAM read interface
.axi_wr_sram_valid        (axi_wr_sram_valid),
.axi_wr_sram_drain        (axi_wr_sram_drain),
.axi_wr_sram_id           (axi_wr_sram_id),
.axi_wr_sram_data         (axi_wr_sram_data),

// AXI master
.m_axi_awvalid            (m_axi_wr_awvalid),
.m_axi_awready            (m_axi_wr_awready),
.m_axi_awid               (m_axi_wr_awid),
.m_axi_awaddr              (m_axi_wr_awaddr),
.m_axi_awlen               (m_axi_wr_awlen),
.m_axi_awsize              (m_axi_wr_awsize),
.m_axi_awburst             (m_axi_wr_awburst),

.m_axi_wvalid              (m_axi_wr_wvalid),
.m_axi_wready              (m_axi_wr_wready),
.m_axi_wdata               (m_axi_wr_wdata),
.m_axi_wstrb                (m_axi_wr_wstrb),
.m_axi_wlast                (m_axi_wr_wlast),
.m_axi_wuser                (m_axi_wr_wuser),

.m_axi_bvalid              (m_axi_wr_bvalid),
.m_axi_bready              (m_axi_wr_bready),
.m_axi_bid                 (m_axi_wr_bid),
.m_axi_bresp                (m_axi_wr_bresp),

// Error and debug
.sched_wr_error            (sched_wr_error),
.dbg_wr_all_complete       (dbg_wr_all_complete),
.dbg_aw_transactions        (dbg_aw_transactions),

```

```
    .dbg_w_beats          (dbg_w_beats)
);
```

17.8 Related Documentation

- **Parent:** 01_stream_core.md - Top-level integration
 - **Scheduler Array:** 02_scheduler_group_array.md - Provides sched_wr_* signals
 - **SRAM Controller:** 08_sram_controller.md - Data source
 - **Drain Controller:** 11_stream_drain_ctrl.md - Data availability tracking
 - **Read Engine:** 06_axi_read_engine.md - Complementary read datapath
-

17.9 Revision History

AXI Write Engine Revision History

Version	Date	Author	Description
0.90	2025-11-22	seang	Initial block specification
0.91	2026-01-02	seang	Added table captions and figure numbers

Last Updated: 2026-01-02

18 APB to Descriptor Router

Module: apbtodescr.sv **Location:** projects/components/stream/rtl/fub/ **Category:** FUB (Functional Unit Block) **Parent:** stream_top.sv **Status:** Implemented **Last Updated:** 2025-11-30

18.1 Overview

The apbtodescr module is an APB-to-descriptor engine router that handles address-based routing of APB writes to descriptor engine kick-off ports. It decodes the APB address to determine the target channel and routes the 64-bit descriptor address using a two-write sequence.

18.1.1 Key Features

- **Address-Based Routing:** APB address bits select target channel
- **64-bit Address Assembly:** Two 32-bit APB writes combine into 64-bit descriptor address
- **Backpressure Handling:** Delays APB response if descriptor engine busy
- **Address Range Checking:** Error on out-of-range addresses
- **Integration Signal:** apb_descriptor_kickoff_hit for response muxing

18.2 Architecture

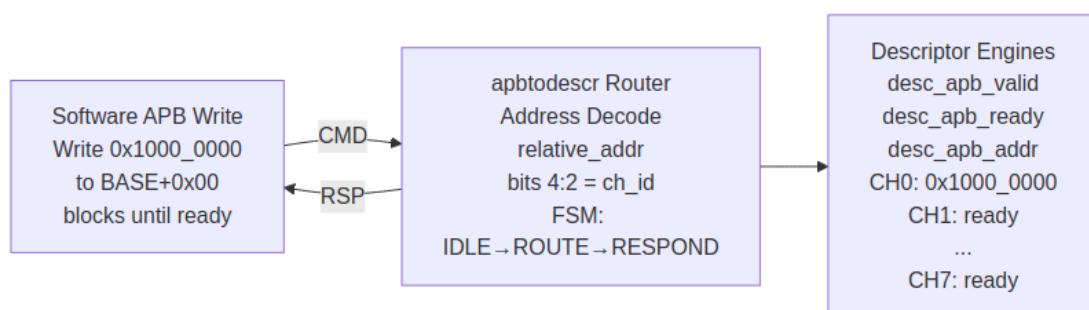
18.2.1 Address Map

Relative to BASE_ADDR:

BASE + 0x00: Channel 0 descriptor address LOW [31:0]
BASE + 0x04: Channel 0 descriptor address HIGH [63:32]
BASE + 0x08: Channel 1 descriptor address LOW [31:0]
BASE + 0x0C: Channel 1 descriptor address HIGH [63:32]
...
BASE + 0x38: Channel 7 descriptor address LOW [31:0]
BASE + 0x3C: Channel 7 descriptor address HIGH [63:32]

18.2.2 Block Diagram

18.2.3 Figure 2.13.1: APB to Descriptor Block Diagram

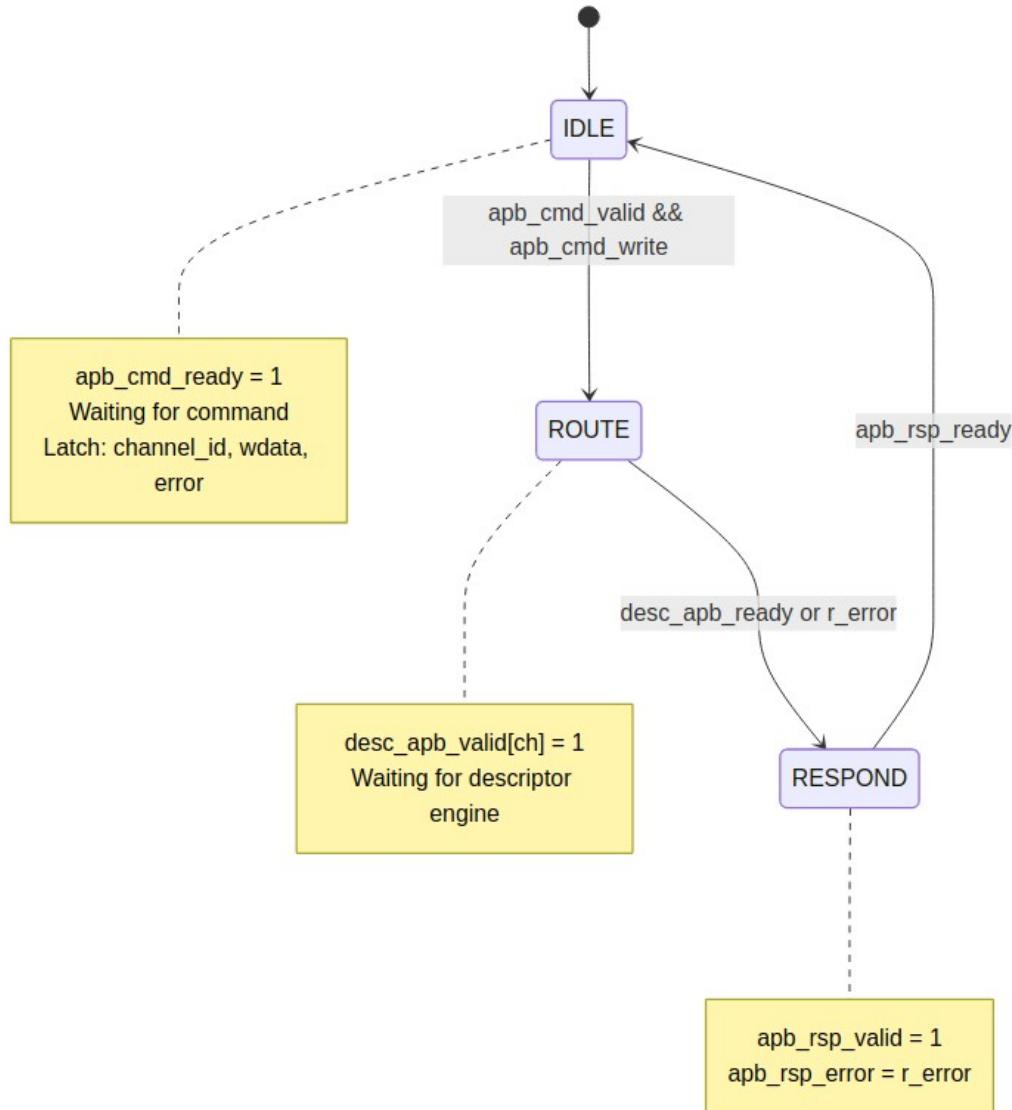


APB to Descriptor Block Diagram

Source: [02_apbtodescr_block.mmd](#)

18.2.4 FSM Diagram

18.2.5 Figure 2.13.2: APB to Descriptor FSM



APB to Descriptor FSM

Source: [02_apbtodescr_fsm.mmd](#)

18.2.6 Write Flow

1. Software writes LOW word to CHx_CTRL_LOW register
2. PeakRDL APB slave presents write on cmd/rsp interface
3. Module captures LOW word, responds with OKAY
4. Software writes HIGH word to CHx_CTRL_HIGH register

5. Module assembles 64-bit address from LOW + HIGH
6. Module asserts desc_apb_valid[channel_id]
7. Module drives desc_apb_addr[channel_id]
8. Waits for desc_apb_ready[channel_id]
9. Completes APB transaction (asserts apb_rsp_valid)

18.2.7 FSM States

```
typedef enum logic [2:0] {
    IDLE          = 3'b000,      // Waiting for APB command (LOW write)
    RESPOND_LOW   = 3'b001,      // Sending response after LOW write
    WAIT_HIGH    = 3'b010,      // Waiting for HIGH write
    ROUTE         = 3'b011,      // Routing to descriptor engine
    RESPOND_HIGH  = 3'b100      // Sending final response after HIGH
write
} state_t;
```

18.3 Parameters

Parameters

Parameter	Type	Default	Description
ADDR_WIDTH	int	32	APB address width
DATA_WIDTH	int	32	APB data width
NUM_CHANNELS	int	8	Number of DMA channels

18.4 Port List

18.4.1 Clock and Reset

Clock and Reset

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

18.4.2 APB Slave CMD Interface

APB Slave CMD Interface

Signal	Direction	Width	Description
apb_cmd_valid	input	1	Command valid
apb_cmd_ready	output	1	Ready to accept command
apb_cmd_addr	input	ADDR_WIDTH	Command address
apb_cmd_wdata	input	DATA_WIDTH	Write data
apb_cmd_write	input	1	Write enable (1=write, 0=read)

18.4.3 APB Slave RSP Interface

APB Slave RSP Interface

Signal	Direction	Width	Description
apb_rsp_valid	output	1	Response valid
apb_rsp_ready	input	1	Response ready
apb_rsp_rdata	output	DATA_WIDTH	Read data (always 0)
apb_rsp_error	output	1	Error flag

18.4.4 Descriptor Engine APB Ports

Descriptor Engine APB Ports

Signal	Direction	Width	Description
desc_apb_val_id[ch]	output	NUM_CHAN NELS	Per-channel valid
desc_apb_ready[ch]	input	NUM_CHAN NELS	Per-channel ready
desc_apb_addr[ch]	output	NUM_CHAN NELS x 64	64-bit descriptor addresses

18.4.5 Integration Control

Integration Control

Signal	Direction	Width	Description
apb_descript or_kickoff_h it	output	1	Indicates kick-off in progress

18.5 Operation

18.5.1 Address Decode

```
// Extract channel ID from address (dword-aligned: addr[5:3])
// Address bits [1:0] are ignored (word-aligned)
// Address bit [2] selects LOW (0) or HIGH (1) register
// Address bits [5:3] select channel 0-7
assign channel_id = apb_cmd_addr[5:3];
assign r_is_high_write = apb_cmd_addr[2];

// Valid range: 0x00 to 0x3F (8 channels x 8 bytes)
assign addr_in_range = ({20'h0, apb_cmd_addr[11:0]} < (NUM_CHANNELS * 8));
```

18.5.2 64-bit Address Assembly

```
// Low write captured in IDLE state
if (r_state == IDLE && apb_cmd_valid) begin
    r_wdata_low <= apb_cmd_wdata;
end

// HIGH write captured in WAIT_HIGH state
if (r_state == WAIT_HIGH && apb_cmd_valid) begin
    r_wdata_high <= apb_cmd_wdata;
end

// Assembled 64-bit address
desc_apb_addr[ch] = {r_wdata_high, r_wdata_low};
```

18.5.3 Error Conditions

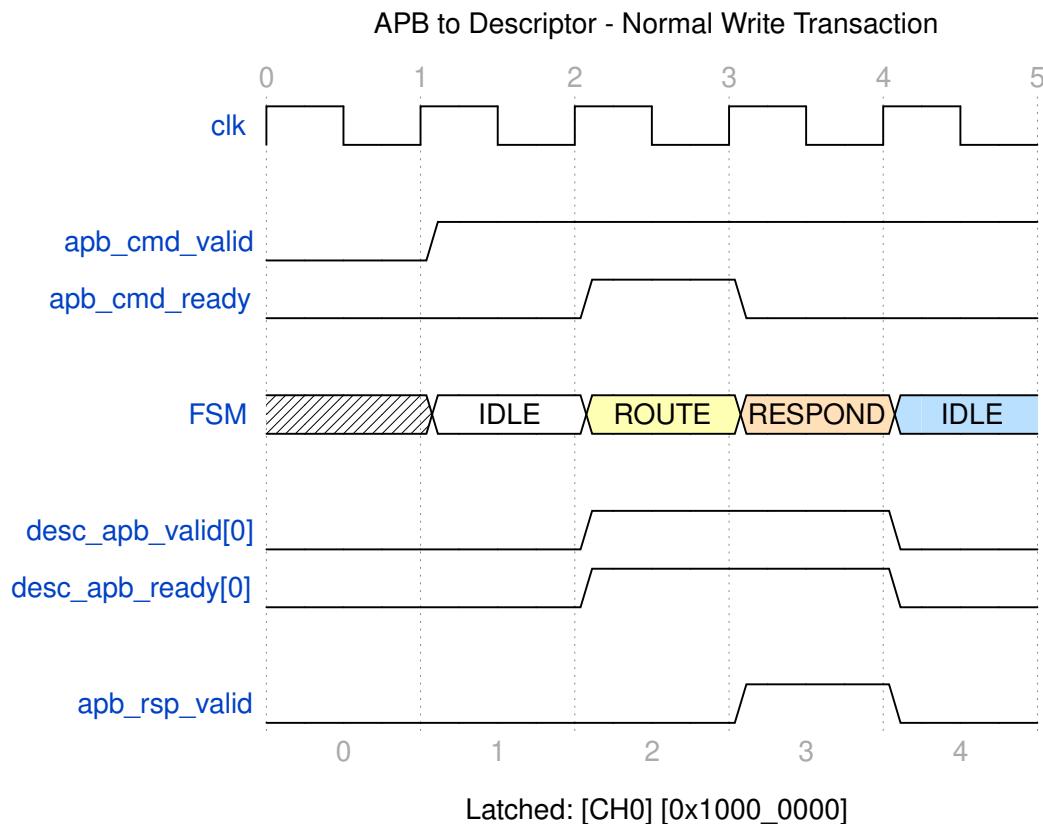
1. **Read Operation:** APB reads not supported
 2. **Address Out of Range:** Beyond channel address space
 3. **HIGH Before LOW:** HIGH write without preceding LOW write
 4. **Wrong Channel:** HIGH write to different channel than LOW
-

18.6 Timing Diagrams

18.6.1 Normal Write Sequence

The following diagram shows the normal write flow where a LOW word followed by HIGH word kicks off a channel:

18.6.1.1 *Waveform 2.13.1: APB Normal Write Sequence*



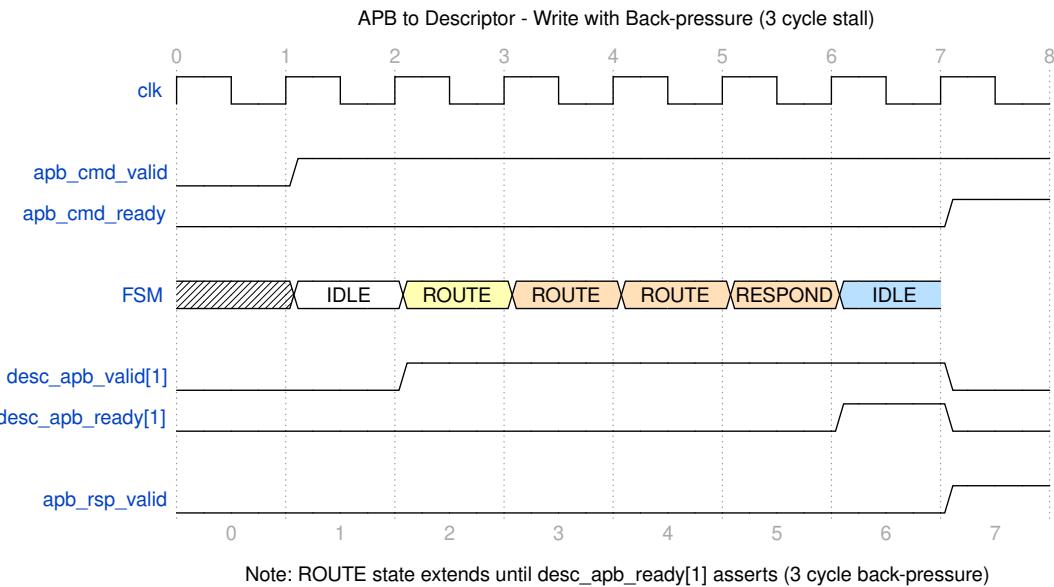
APB Normal Write

Source: [apbtodescr_normal_write.json](#)

18.6.2 Backpressure Handling

When the descriptor engine is busy, the APB response is delayed until the engine accepts the kick-off:

18.6.2.1 *Waveform 2.13.2: APB Write with Backpressure*

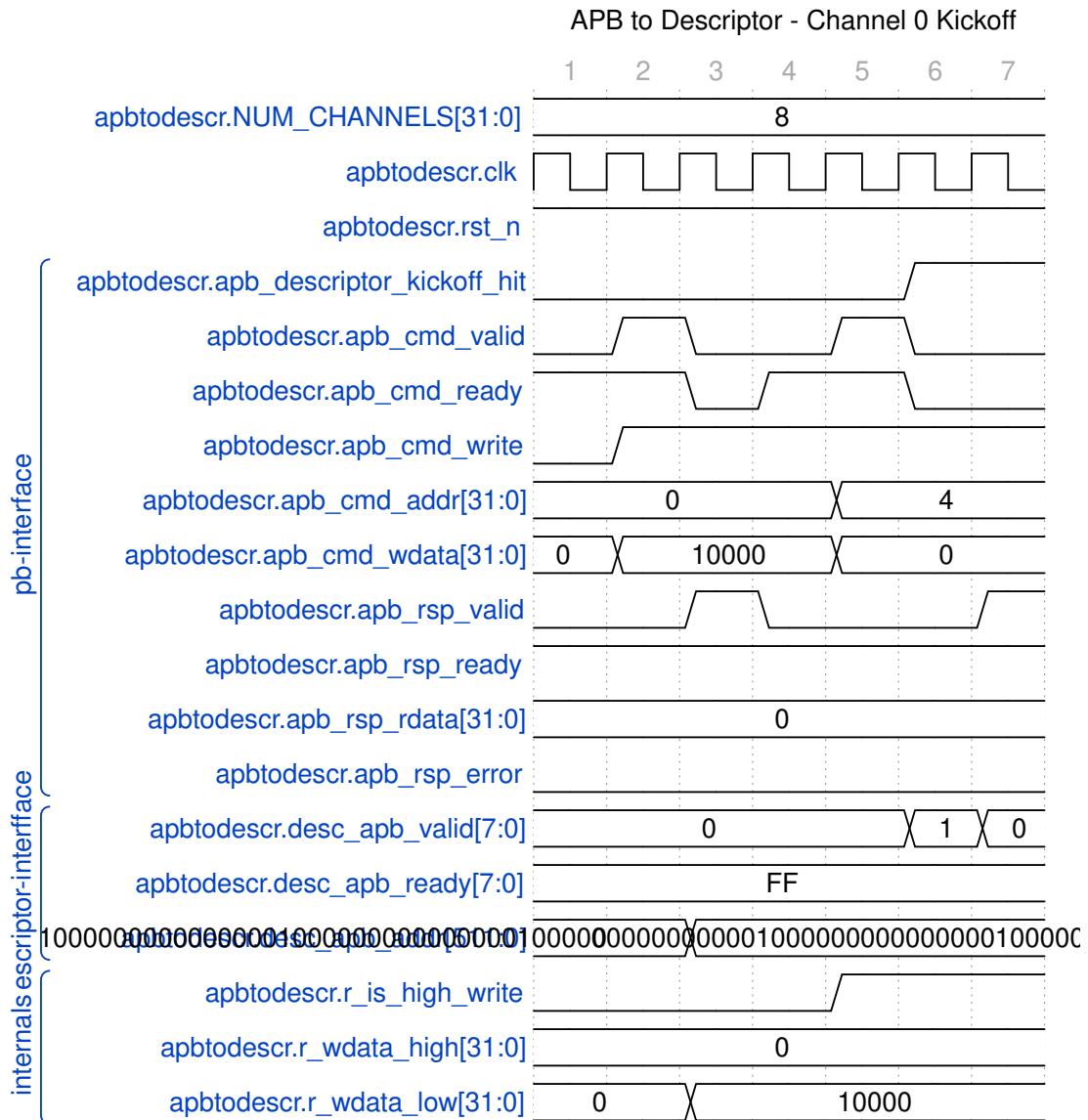


APB Backpressure Write

Source: [apbtodescr_backpressure_write.json](#)

18.6.3 Channel Kick-off Examples

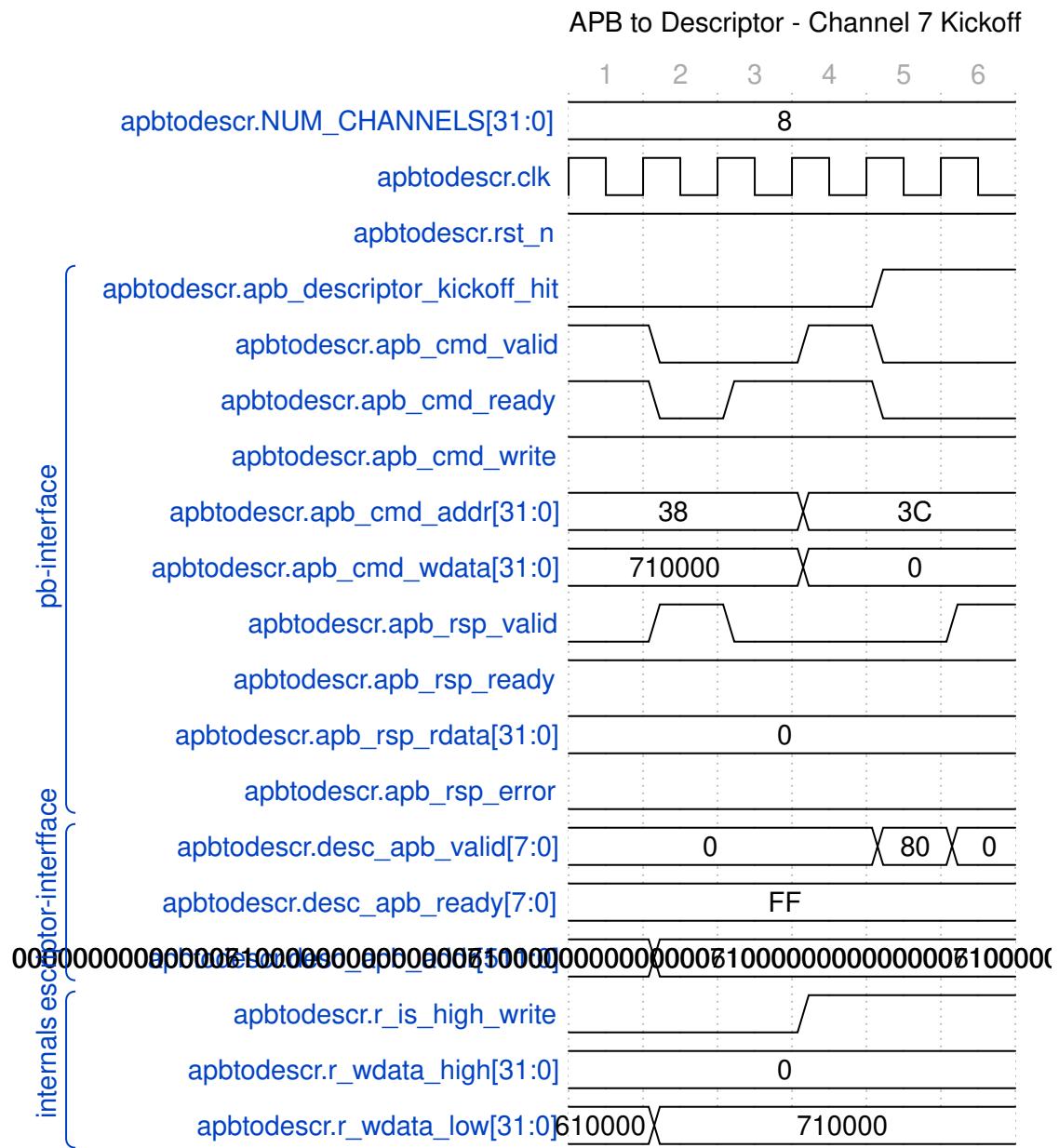
18.6.3.1 *Waveform 2.13.3: Channel 0 Kick-off*



Channel 0 Kick-off

Source: [apbtodescr_ch0_kickoff.json](#)

18.6.3.2 Waveform 2.13.4: Channel 7 Kick-off



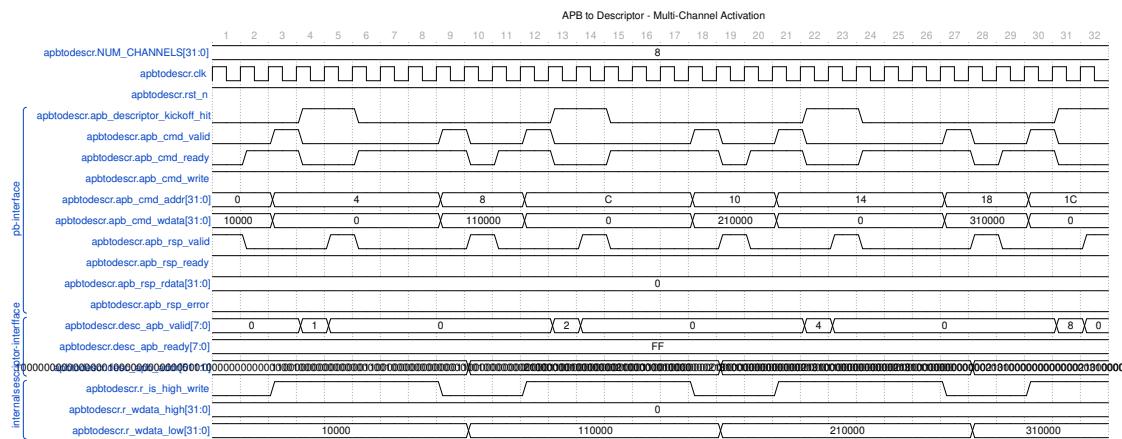
Channel 7 Kick-off

Source: [apbtodescr_ch7_kickoff.json](#)

18.6.4 Multi-Channel Operation

The following diagram shows multiple channels being kicked off in sequence:

18.6.4.1 Waveform 2.13.5: Multi-Channel Kick-off Sequence



Multi-Channel Kick-off

Source: [apbtodescr_multi_channel.json](#)

18.7 Integration Example

```
apbtodescr #(
    .ADDR_WIDTH      (32),
    .DATA_WIDTH      (32),
    .NUM_CHANNELS    (8)
) u_apbtodescr (
    .clk                    (clk),
    .rst_n                  (rst_n),
    // APB CMD/RSP from PeakRDL
    .apb_cmd_valid          (apb_kickoff_cmd_valid),
    .apb_cmd_ready          (apb_kickoff_cmd_ready),
    .apb_cmd_addr           (apb_kickoff_cmd_addr),
    .apb_cmd_wdata          (apb_kickoff_cmd_wdata),
    .apb_cmd_write          (apb_kickoff_cmd_write),
    .apb_rsp_valid          (apb_kickoff_rsp_valid),
    .apb_rsp_ready          (apb_kickoff_rsp_ready),
    .apb_rsp_rdata          (apb_kickoff_rsp_rdata),
    .apb_rsp_error          (apb_kickoff_rsp_error),
    // To descriptor engines
    .desc_apb_valid         (desc_apb_valid),
    .desc_apb_ready         (desc_apb_ready),
    .desc_apb_addr          (desc_apb_addr),
```

```
// Integration
    .apb_descriptor_kickoff_hit (kickoff_hit)
);
```

18.8 Common Issues

18.8.1 Issue 1: APB Response Not Returning

Symptom: APB write hangs without response

Root Causes: 1. Descriptor engine not asserting ready 2. FSM stuck in ROUTE state 3. apb_rsp_ready not being asserted

Solution: Check descriptor engine acceptance and FSM state.

18.8.2 Issue 2: Wrong Channel Kicked Off

Symptom: Different channel starts than expected

Root Cause: HIGH write sent to different channel than LOW

Solution: Ensure software writes LOW then HIGH to same channel address.

18.9 Related Documentation

- **Parent:** 01_stream_core.md - Top-level integration
 - **Consumer:** 05_descriptor_engine.md - Receives kick-off
 - **Register File: PeakRDL-generated stream_regs.sv**
-

18.10 Revision History

APB to Descriptor Router Revision History

Version	Date	Author	Description
0.90	2025-11-22	seang	Initial block specification
0.91	2026-01-02	seang	Added table captions and figure

Version	Date	Author	Description
			numbers

Last Updated: 2026-01-02

19 APB Configuration Block

Module: stream_config_block.sv **Location:** projects/components/stream/rtl/top/
Category: TOP (Integration) **Parent:** stream_top.sv **Status:** Implemented **Last Updated:** 2025-11-30

19.1 Overview

The stream_config_block module maps PeakRDL-generated register outputs to STREAM core configuration inputs. It provides a clean interface layer between the register file and the STREAM DMA engine core, handling field extraction, width conversion, and global enable gating.

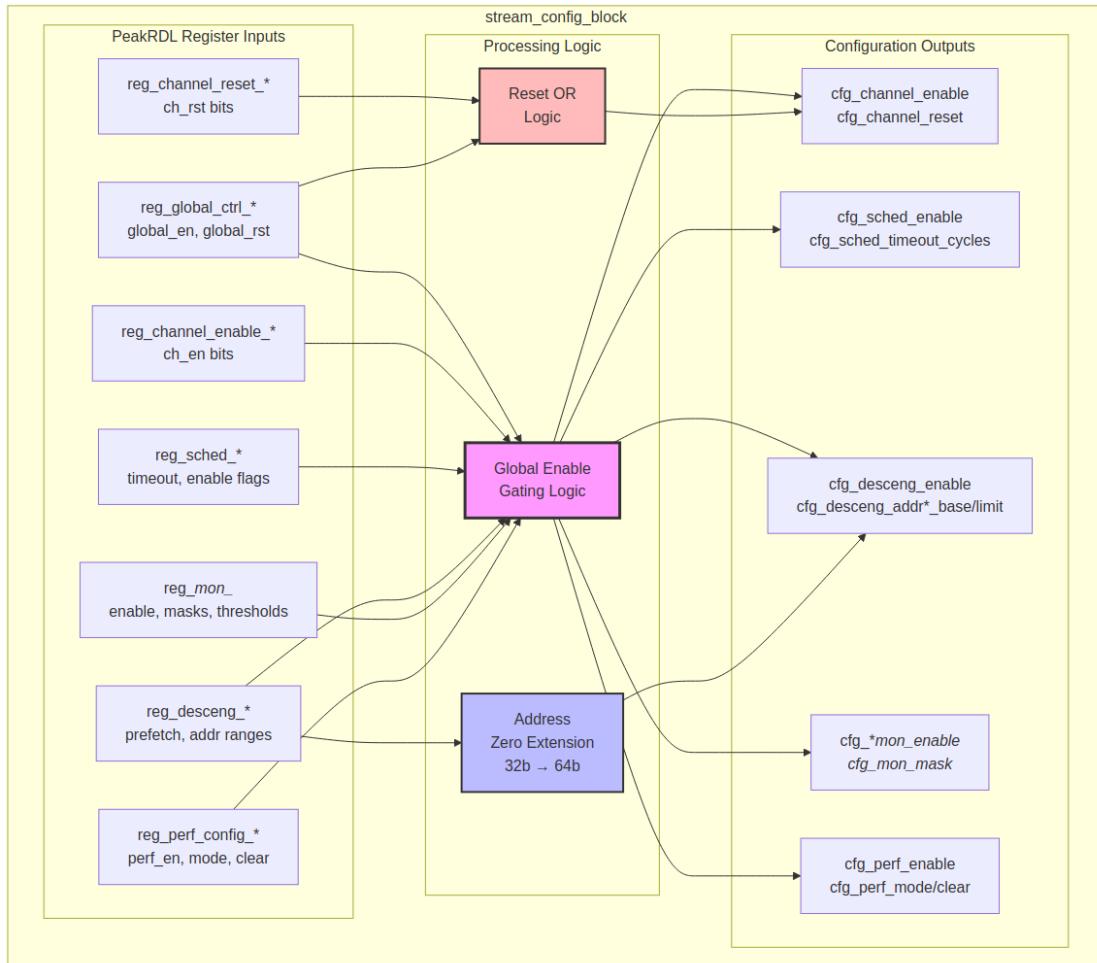
19.1.1 Key Features

- **Field Extraction:** Extracts fields from packed register values
 - **Width Conversion:** Converts register fields to configuration signal widths
 - **Global Enable Gating:** Gates enables by global enable register
 - **Address Extension:** Zero-extends 32-bit register addresses to 64-bit
 - **Clean Interface:** Isolates register naming from core logic
-

19.2 Architecture

19.2.1 Block Diagram

19.2.2 Figure 1: Stream Config Block Diagram



Stream Config Block Diagram

Source: [14_stream_config_block.mmd](#)

19.2.3 Configuration Groups

```
stream_config_block
  Global and Channel Control
    cfg_channel_enable (gated by global_en)
    cfg_channel_reset (OR'd with global_rst)
  Scheduler Configuration
    cfg_sched_enable (gated by global_en)
    cfg_sched_timeout_cycles
```

```

cfg_sched_*_enable flags
Descriptor Engine Configuration
  cfg_desceng_enable (gated by global_en)
  cfg_desceng_prefetch
  cfg_desceng_addr*_base/limit (zero-extended)
Monitor Configurations (3 sets)
  Descriptor AXI Monitor (cfg_desc_mon_*)
  Read Engine Monitor (cfg_rdeng_mon_*)
  Write Engine Monitor (cfg_wreng_mon_*)
AXI Transfer Configuration
  cfg_axi_rd_xfer_beats
  cfg_axi_wr_xfer_beats
Performance Profiler
  cfg_perf_enable (gated by global_en)
  cfg_perf_mode
  cfg_perf_clear

```

19.3 Parameters

Parameters

Parameter	Type	Default	Description
NUM_CHANNELS	int	8	Number of DMA channels
ADDR_WIDTH	int	64	Address width for config outputs

19.4 Port List

19.4.1 Clock and Reset

Clock and Reset

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

19.4.2 PeakRDL Register Inputs

Global Control:

PeakRDL Register Inputs

Signal	Direction	Width	Description
reg_global_ct	input	1	Global enable
rl_global_en			
reg_global_ct	input	1	Global reset
rl_global_rst			
reg_channel_e_nable_ch_en	input	8	Per-channel enable bits
reg_channel_r_eset_ch_rst	input	8	Per-channel reset bits

Scheduler Configuration:

PeakRDL Register Inputs

Signal	Direction	Width	Description
reg_sched_timeout_cycles_*	input	16	Timeout threshold
reg_sched_config_*	input	1	Various enables

Descriptor Engine Configuration:

PeakRDL Register Inputs

Signal	Direction	Width	Description
reg_desceng_config_*	input	varies	Enable and threshold
reg_desceng_addr_*_*	input	32	Address range limits

Monitor Configurations (3 sets):

Each monitor (DAXMON, RDMON, WRMON) has:

- reg_*_enable_* - Enable flags
- reg_*_timeout_* - Timeout configuration
- reg_*_latency_thresh_* - Latency threshold
- reg_*_pkt_mask_* - Packet type mask
- reg_*_err_cfg_* - Error configuration
- reg_*_mask_* - Event masks

AXI Transfer Configuration:

PeakRDL Register Inputs

Signal	Direction	Width	Description
reg_axi_xfer_rd_beats	input	8	Read burst size
reg_axi_xfer_wr_beats	input	8	Write burst size
config_rd_xfe			
config_wr_xfe			

Performance Profiler:

PeakRDL Register Inputs

Signal	Direction	Width	Description
reg_perf_conf	input	1	Profiler enable
ig_perf_en			
reg_perf_conf	input	1	Profiling mode
ig_perf_mode			
reg_perf_conf	input	1	Clear profiler
ig_perf_clear			

19.4.3 Configuration Outputs

Global and Channel:

Configuration Outputs

Signal	Direction	Width	Description
cfg_channel_enable	output	NUM_CHAN NELS	Gated channel enables
cfg_channel_reset	output	NUM_CHAN NELS	Combined channel resets

Scheduler:

Configuration Outputs

Signal	Direction	Width	Description
cfg_sched_enable	output	1	Gated scheduler enable
cfg_sched_timeout	output	16	Timeout value
eout_cycles			

Signal	Direction	Width	Description
cfg_sched_*_enable	output	1	Feature enables

Descriptor Engine:

Configuration Outputs

Signal	Direction	Width	Description
cfg_desceng_enable	output	1	Gated desceng enable
cfg_desceng_prefetch	output	1	Prefetch enable
cfg_desceng_fifo_thresh	output	4	FIFO threshold
cfg_desceng_addr*_base	output	ADDR_WIDT H	Zero-extended address
cfg_desceng_addr*_limit	output	ADDR_WIDT H	Zero-extended address

Monitor Outputs (3 sets):

Each monitor outputs ~16 configuration signals covering: - Enable flags - Timeout values - Thresholds - Masks

AXI Transfer:

Configuration Outputs

Signal	Direction	Width	Description
cfg_axi_rd_xfer_beats	output	8	Read burst size
cfg_axi_wr_xfer_beats	output	8	Write burst size

Performance Profiler:

Configuration Outputs

Signal	Direction	Width	Description
cfg_perf_enable	output	1	Gated profiler enable
cfg_perf_mode	output	1	Profiling mode

Signal	Direction	Width	Description
cfg_perf_clea r	output	1	Clear signal

19.5 Operation

19.5.1 Global Enable Gating

```
// Gate all channel enables by global enable
assign cfg_channel_enable = reg_channel_enable_ch_en &
{NUM_CHANNELS{reg_global_ctrl_global_en}};

// Gate scheduler enable
assign cfg_sched_enable = reg_sched_config_sched_en &
reg_global_ctrl_global_en;

// Gate monitor enables
assign cfg_desc_mon_enable = reg_daxmon_enable_mon_en &
reg_global_ctrl_global_en;
```

19.5.2 Global Reset OR

```
// Channel resets are OR'd with global reset
assign cfg_channel_reset = reg_channel_reset_ch_RST | 
{NUM_CHANNELS{reg_global_ctrl_global_RST}};
```

19.5.3 Address Zero Extension

```
// Zero-extend 32-bit register addresses to ADDR_WIDTH (typically 64-bit)
assign cfg_desceng_addr0_base = {{(ADDR_WIDTH-32){1'b0}}, 
reg_desceng_addr0_base_addr0_base};
assign cfg_desceng_addr0_limit = {{(ADDR_WIDTH-32){1'b0}}, 
reg_desceng_addr0_limit_addr0_limit};
```

19.6 Integration Example

```
stream_config_block #(
    .NUM_CHANNELS    (8),
    .ADDR_WIDTH      (64)
) u_config_block (
    .clk            (clk),
    .rst_n          (rst_n),

    // From PeakRDL register file
```

```

    .reg_global_ctrl_global_en      (regs_global_ctrl_global_en),
    .reg_global_ctrl_global_RST     (regs_global_ctrl_global_RST),
    .reg_channel_enable_ch_en      (regs_channel_enable_ch_en),
    // ... more register inputs

    // To stream_core
    .cfg_channel_enable           (cfg_channel_enable),
    .cfg_channel_reset            (cfg_channel_reset),
    .cfg_sched_enable              (cfg_sched_enable),
    // ... more config outputs
);

```

19.7 Related Documentation

- **Parent:** 01_stream_core.md - Receives configuration signals
- **Register File:** PeakRDL-generated stream_regs.sv
- **Scheduler:** 04_scheduler.md - Uses sched_* config
- **Descriptor Engine:** 05_descriptor_engine.md - Uses desceng_* config
- **Monitors:** [Monitor documentation for each AXI monitor](#)

19.8 Revision History

APB Configuration Block Revision History

Version	Date	Author	Description
0.90	2025-11-22	seang	Initial block specification
0.91	2026-01-02	seang	Added table captions and figure numbers

Last Updated: 2026-01-02

20 Performance Profiler

Module: perf_profiler.sv **Location:** projects/components/stream/rtl/fub/ **Category:** FUB (Functional Unit Block) **Parent:** stream_top.sv **Status:** Implemented **Last Updated:** 2025-11-30

20.1 Overview

The perf_profiler module captures timing information for channel activity to enable performance analysis and bottleneck identification. It monitors channel idle signals and records either timestamps or elapsed times to a FIFO for software retrieval.

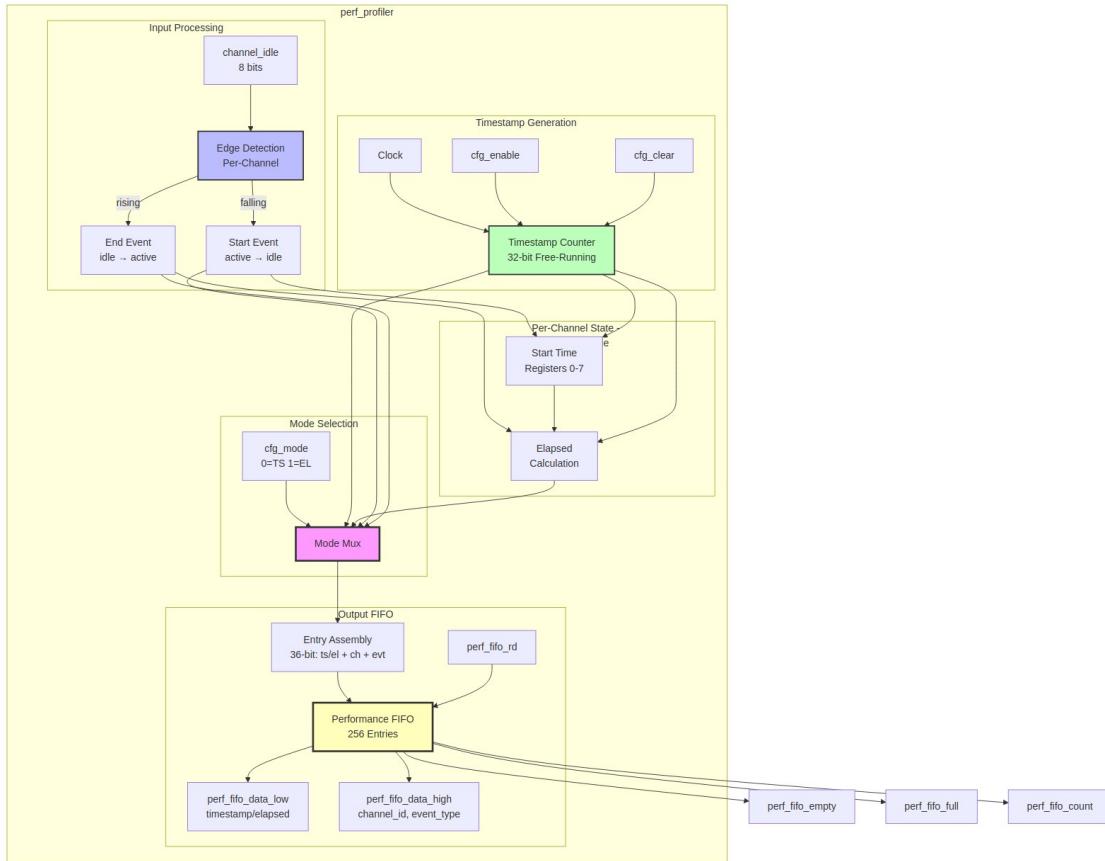
20.1.1 Key Features

- **Two Profiling Modes:**
 - Mode 0 (TIMESTAMP): Captures timestamps on idle signal edges
 - Mode 1 (ELAPSED): Captures elapsed time directly
 - **Per-Channel Profiling:** Tracks all 8 channels independently
 - **256-Entry FIFO:** Buffers performance data for software polling
 - **Channel ID Tagging:** Each entry includes source channel
 - **Configurable via APB:** Enable, mode, and clear controls
-

20.2 Architecture

20.2.1 Block Diagram

20.2.2 Figure 2.15.1: Performance Profiler Block Diagram



Performance Profiler Block Diagram

Source: [15_perf_profiler_block.mmd](#)

20.2.3 Profiling Modes

Mode 0 - TIMESTAMP:

Records timestamp when:

- Channel transitions idle ⇌ active (start)
- Channel transitions active ⇌ idle (end)

Software calculates: `elapsed = timestamp_end - timestamp_start`

Advantages:

- Simpler hardware

- More flexible analysis
- Can detect partial transfers

Mode 1 - ELAPSED:

Records elapsed time when:

- Channel returns to idle

`elapsed_time = current_timestamp - start_timestamp`

Advantages:

- Hardware calculates duration
- Simpler software processing
- Direct performance metric

20.2.4 FIFO Entry Format (36-bit)

FIFO Entry Format

Bits	Field	Description
[31:0]	timestamp/elapsed	Timestamp or elapsed time
[34:32]	channel_id	Source channel (0-7)
[35]	event_type	0=start, 1=end (timestamp mode)

20.3 Parameters

Parameters

Parameter	Type	Default	Description
NUM_CHANNELS	int	8	Number of channels to monitor
CHANNEL_WIDTH	int	$\$clog2(\text{NUM_CHANNELS})$	Channel ID width
TIMESTAMP_WIDTH	int	32	Timestamp counter width
FIFO_DEPTH	int	256	Performance FIFO depth
FIFO_ADDR_WIDTH	int	$\$clog2(\text{FIFO_D})$	FIFO address

Parameter	Type	Default	Description
TH		EPTH)	width

20.4 Port List

20.4.1 Clock and Reset

Clock and Reset

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

20.4.2 Channel Monitoring

Channel Monitoring

Signal	Direction	Width	Description
channel_idle	input	NUM_CHANNELS	Per-channel idle signals

20.4.3 Configuration Interface

Configuration Interface

Signal	Direction	Width	Description
cfg_enable	input	1	Enable profiling
cfg_mode	input	1	0=timestamp, 1=elapsed
cfg_clear	input	1	Clear FIFO and counters

20.4.4 FIFO Read Interface

FIFO Read Interface

Signal	Direction	Width	Description
perf_fifo_rd	input	1	Read strobe (pops FIFO)

Signal	Direction	Width	Description
perf_fifo_da_ta_low	output	32	Timestamp/elapsed [31:0]
perf_fifo_da_ta_high	output	32	{28'b0, event_type, channel_id}
perf_fifo_em_pty	output	1	FIFO empty flag
perf_fifo_fu_ll	output	1	FIFO full flag
perf_fifo_count	output	16	Number of entries

20.5 Operation

20.5.1 Edge Detection

```
// Detect idle signal transitions
assign w_idle_rising = channel_idle & ~r_idle_prev; // Active ⇢ Idle (end)
assign w_idle_falling = ~channel_idle & r_idle_prev; // Idle ⇢ Active (start)
```

20.5.2 Timestamp Counter

```
// Free-running 32-bit counter
// Increments every cycle when profiling enabled
// Wraps at 2^32 - 1 (software handles rollover)
always_ff @(posedge clk or negedge rst_n) begin
    if (!rst_n) begin
        r_timestamp_counter <= '0;
    end else if (cfg_clear) begin
        r_timestamp_counter <= '0;
    end else if (cfg_enable) begin
        r_timestamp_counter <= r_timestamp_counter + 1'b1;
    end
end
```

20.5.3 Start Time Capture (Elapsed Mode)

```
// Per-channel start time tracking
generate
    for (ch = 0; ch < NUM_CHANNELS; ch++) begin
        always_ff @(posedge clk) begin
            if (cfg_enable && cfg_mode == MODE_ELAPSED) begin
                if (w_idle_falling[ch]) begin
```

```

        r_start_time[ch] <= r_timestamp_counter;
        r_channel_active[ch] <= 1'b1;
    end
    else if (w_idle_rising[ch]) begin
        r_channel_active[ch] <= 1'b0;
    end
end
end
endgenerate

```

20.5.4 Two-Register Read Sequence

Software reads FIFO via APB:

1. Check perf_fifo_empty == 0
 2. Read PERF_FIFO_DATA_LOW:
 - Triggers perf_fifo_rd strobe
 - Pops FIFO, latches 36-bit entry
 - Returns timestamp/elapsed [31:0]
 3. Read PERF_FIFO_DATA_HIGH:
 - Returns {28'b0, event_type, channel_id}
 - No FIFO pop
 4. Parse data and repeat
-

20.6 Integration Example

```

perf_profiler #(
    .NUM_CHANNELS      (8),
    .TIMESTAMP_WIDTH   (32),
    .FIFO_DEPTH        (256)
) u_perf_profiler (
    .clk                (clk),
    .rst_n              (rst_n),

    // From schedulers
    .channel_idle       (scheduler_idle),

    // Configuration
    .cfg_enable         (cfg_perf_enable),
    .cfg_mode           (cfg_perf_mode),
    .cfg_clear          (cfg_perf_clear),

    // FIFO read interface (to APB)
    .perf_fifo_rd       (perf_fifo_rd),
    .perf_fifo_data_low (perf_fifo_data_low),
    .perf_fifo_data_high(perf_fifo_data_high),

```

```
.perf_fifo_empty    (perf_fifo_empty),
.perf_fifo_full     (perf_fifo_full),
.perf_fifo_count    (perf_fifo_count)
);
```

20.7 Software Usage

20.7.1 Example: Measure Channel 0 Transfer Time

```
// Enable timestamp mode
write_reg(PERF_CONFIG, PERF_EN | MODE_TIMESTAMP);

// Start transfer on channel 0
write_reg(CH0_CTRL_LOW, desc_addr_low);
write_reg(CH0_CTRL_HIGH, desc_addr_high);

// Wait for completion
while (!read_reg(CH0_STATUS) & COMPLETE);

// Read performance data
while (!(read_reg(PERF_STATUS) & FIFO_EMPTY)) {
    uint32_t low = read_reg(PERF_FIFO_DATA_LOW);           // Pops FIFO
    uint32_t high = read_reg(PERF_FIFO_DATA_HIGH);         // Reads latched

    uint32_t timestamp = low;
    uint8_t channel   = high & 0x7;
    uint8_t event     = (high >> 3) & 0x1;

    if (channel == 0) {
        if (event == 0) // Start
            start_time = timestamp;
        else           // End
            elapsed = timestamp - start_time;
    }
}
```

20.8 Common Issues

20.8.1 Issue 1: FIFO Overflow

Symptom: Performance data lost

Root Cause: Software not polling fast enough

Solution: - Poll more frequently - Use larger FIFO depth - Consider interrupt on half-full

20.8.2 Issue 2: Timestamp Rollover

Symptom: Negative elapsed times calculated

Root Cause: 32-bit counter wrapped during measurement

Solution: Handle rollover in software:

```
if (end_time < start_time)
    elapsed = (0xFFFFFFFF - start_time) + end_time + 1;
```

20.9 Related Documentation

- **Parent:** 01_stream_core.md - Top-level integration
- **Data Source:** 04_scheduler.md - Provides idle signals
- **Configuration:** [14_apb_config.md - Config register mapping](#)

20.10 Revision History

Performance Profiler Revision History

Version	Date	Author	Description
0.90	2025-11-22	seang	Initial block specification
0.91	2026-01-02	seang	Added table captions and figure numbers

Last Updated: 2026-01-02

21 MonBus AXI-Lite Group

Module: monbus_axil_group.sv **Location:** projects/components/stream/rtl/macro/

Category: MACRO (Integration) **Parent:** stream_top.sv **Status:** Implemented **Last Updated:** 2025-11-30

21.1 Overview

The `monbus_axil_group` module receives monitor bus packets from STREAM channels, applies configurable filtering, and routes filtered packets to either an error/interrupt FIFO (accessible via AXI-Lite slave) or a master write interface (writes to memory).

21.1.1 Key Features

- **Single Monitor Bus Input:** STREAM is memory-to-memory (no network paths)
- **Per-Protocol Filtering:** Configurable packet type masking
- **Dual Output Paths:**
 - Error/Interrupt FIFO - generates interrupt when not empty
 - Master Write FIFO - writes to configurable address range
- **Protocol Support:** AXI, AXIS, CORE (3 protocols)
- **Built-in AXI-Lite Skid Buffering:** For timing closure

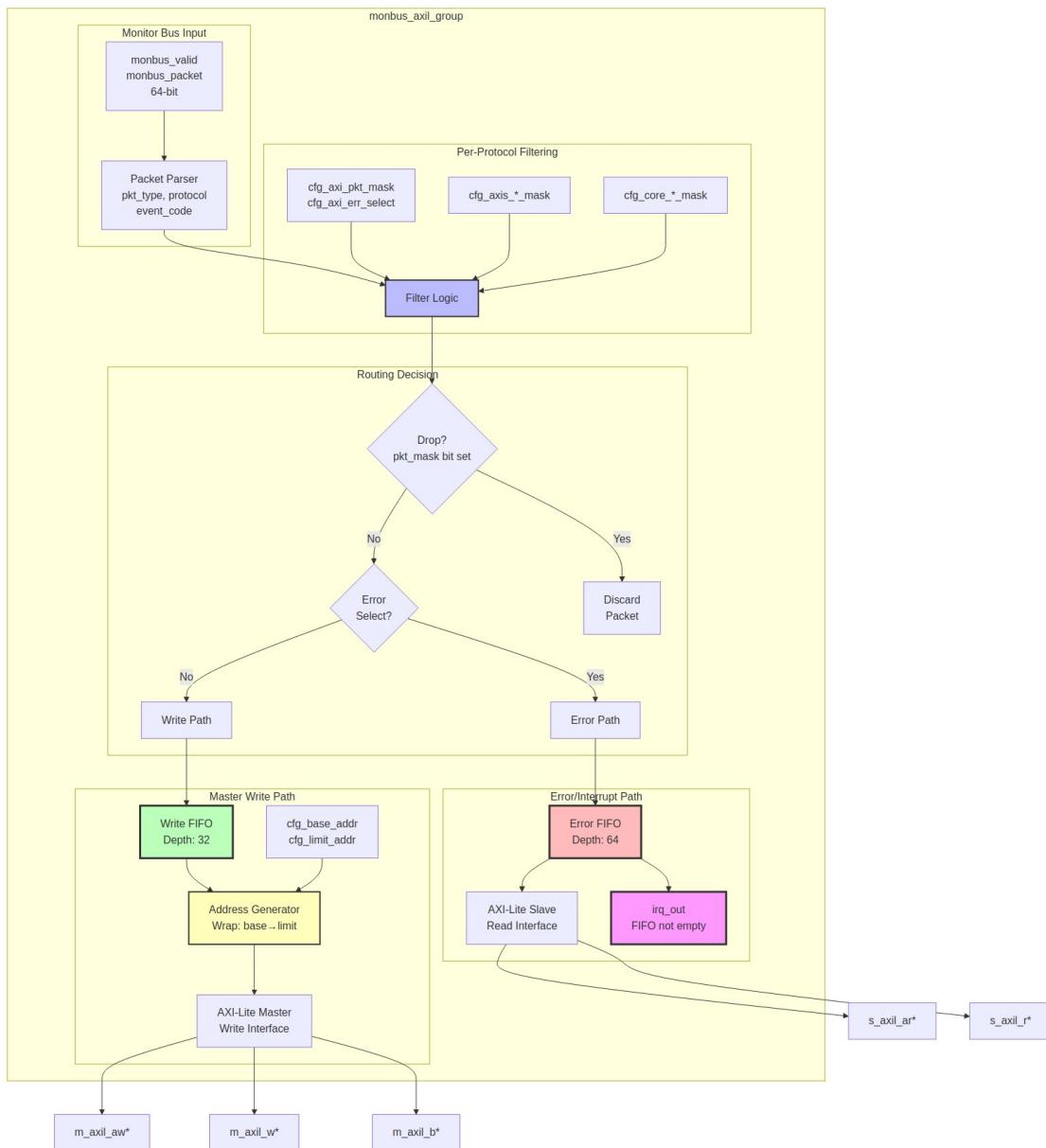
21.1.2 Simplified from RAPIDS

RAPIDS has source + sink data paths requiring TWO monitor buses. STREAM has memory-to-memory only, so this module has ONE monitor bus input (no arbitration needed).

21.2 Architecture

21.2.1 Block Diagram

21.2.2 Figure 2.16.1: MonBus AXI-Lite Group Block Diagram



MonBus AXI-Lite Group Block Diagram

Source: [16_monbus_axil_group_block.mmd](#)

21.2.3 Filter Decision Tree

For each incoming packet:

1. Extract: pkt_type, pkt_protocol, pkt_event_code
 2. Check protocol-specific masks:
 - pkt_mask: Drop if bit[pkt_type] = 1
 - err_select: Route to error FIFO if bit[pkt_type] = 1
 - event_mask: Drop if bit[event_code] = 1
 3. Route:
 - Dropped packets: Consumed silently
 - Error-selected: To error/interrupt FIFO
 - Remaining: To master write FIFO
-

21.3 Parameters

Parameters

Parameter	Type	Default	Description
FIFO_DEPTH_ER R	int	64	Error/interrupt FIFO depth
FIFO_DEPTH_WR ITE	int	32	Master write FIFO depth
ADDR_WIDTH	int	32	AXI address width
DATA_WIDTH	int	32	AXI data width
NUM_PROTOCOLS	int	3	AXI, AXIS, CORE

21.4 Port List

21.4.1 Clock and Reset

Clock and Reset

Signal	Direction	Width	Description
axi_aclk	input	1	AXI clock
axi_aresetn	input	1	AXI active-low reset

21.4.2 Monitor Bus Input

Monitor Bus Input

Signal	Direction	Width	Description
monbus_valid	input	1	Packet valid
monbus_ready	output	1	Ready to accept
monbus_packet	input	64	64-bit monitor packet

21.4.3 AXI-Lite Slave Read Interface

AXI-Lite Slave Read Interface

Signal	Direction	Width	Description
s_axil_arval_id	input	1	Read address valid
s_axil_arready_y	output	1	Read address ready
s_axil_araddr	input	ADDR_WIDTH	Read address
s_axil_arprot	input	3	Protection bits
s_axil_rvalid	output	1	Read data valid
s_axil_rready	input	1	Read data ready
s_axil_rdata	output	DATA_WIDTH	Read data
s_axil_rresp	output	2	Read response

21.4.4 AXI-Lite Master Write Interface

AXI-Lite Master Write Interface

Signal	Direction	Width	Description
m_axil_awval_id	output	1	Write address valid
m_axil_awready_y	input	1	Write address ready
m_axil_awaddr	output	ADDR_WIDTH	Write address

Signal	Direction	Width	Description
m_axil_awprot	output	3	Protection bits
m_axil_wvalid	output	1	Write data valid
m_axil_wready	input	1	Write data ready
m_axil_wdata	output	DATA_WIDTH	Write data
m_axil_wstrb	output	DATA_WIDTH/8	Write strobes
m_axil_bvalid	input	1	Write response valid
m_axil_bready	output	1	Write response ready
m_axil_bresp	input	2	Write response

21.4.5 Interrupt Output

Interrupt Output

Signal	Direction	Width	Description
irq_out	output	1	Interrupt (error FIFO not empty)

21.4.6 Configuration Interface

Configuration Interface

Signal	Direction	Width	Description
cfg_base_addr	input	ADDR_WIDTH	Base for master writes
cfg_limit_addr	input	ADDR_WIDTH	Limit for master writes

21.4.7 Protocol Configuration (per protocol)

AXI Protocol:

Protocol Configuration

Signal	Direction	Width	Description
cfg_axi_pkt_mask	input	16	Drop mask for packet types
cfg_axi_err_select	input	16	Error FIFO select
cfg_axi_*_mask	input	16	Per-event-type masks

AXIS Protocol:

Protocol Configuration

Signal	Direction	Width	Description
cfg_axis_pkt_mask	input	16	Drop mask
cfg_axis_err_select	input	16	Error FIFO select
cfg_axis_*_mask	input	16	Per-event-type masks

CORE Protocol:

Protocol Configuration

Signal	Direction	Width	Description
cfg_core_pkt_mask	input	16	Drop mask
cfg_core_err_select	input	16	Error FIFO select
cfg_core_*_mask	input	16	Per-event-type masks

21.4.8 Debug/Status

Debug/Status

Signal	Direction	Width	Description
err_fifo_full	output	1	Error FIFO full
write_fifo_full	output	1	Write FIFO full

Signal	Direction	Width	Description
err_fifo_count	output	8	Error FIFO count
write_fifo_count	output	8	Write FIFO count

21.5 Operation

21.5.1 Packet Type Filtering

```
// Protocol-specific filtering (example: AXI)
case (pkt_protocol)
    3'b000: begin // AXI
        // Check if packet type is dropped
        pkt_drop = cfg_axi_pkt_mask[pkt_type];

        // Check if packet goes to error FIFO
        pkt_to_err_fifo = cfg_axi_err_select[pkt_type] && !pkt_drop;

        // Check individual event masking
        case (pkt_type)
            PktTypeError:      pkt_event_masked =
                cfg_axi_error_mask[pkt_event_code];
            PktTypeTimeout:   pkt_event_masked =
                cfg_axi_timeout_mask[pkt_event_code];
            // ... more packet types
        endcase
    end
endcase
```

21.5.2 Master Write Address Generation

```
// Address wraps within configured range
always_comb begin
    next_write_addr = current_write_addr + (DATA_WIDTH == 64 ? 8 : 4);
    if (next_write_addr > cfg_limit_addr) begin
        next_write_addr = cfg_base_addr;
    end
end
```

21.5.3 Master Write FSM

```
typedef enum logic [2:0] {
    WRITE_IDLE      = 3'b000,
    WRITE_ADDR      = 3'b001,
    WRITE_DATA_LOW  = 3'b010,
```

```

    WRITE_DATA_HIGH  = 3'b011, // For 32-bit data width only
    WRITE RESP      = 3'b100
} write_state_t;

```

21.6 Integration Example

```

monbus_axil_group #(
    .FIFO_DEPTH_ERR      (64),
    .FIFO_DEPTH_WRITE    (32),
    .ADDR_WIDTH          (32),
    .DATA_WIDTH          (32),
    .NUM_PROTOCOLS       (3)
) u_monbus_axil_group (
    .axi_aclk            (clk),
    .axi_aresetn         (rst_n),

    // Monitor bus input
    .monbus_valid        (stream_mon_valid),
    .monbus_ready        (stream_mon_ready),
    .monbus_packet       (stream_mon_packet),

    // AXI-Lite slave read (for error FIFO access)
    .s_axil_arvalid     (axil_slave_arvalid),
    .s_axil_arready     (axil_slave_arready),
    // ... more slave signals

    // AXI-Lite master write (for logging)
    .m_axil_awvalid     (axil_master_awvalid),
    .m_axil_awready     (axil_master_awready),
    // ... more master signals

    // Interrupt
    .irq_out             (monbus_interrupt),

    // Configuration
    .cfg_base_addr       (cfg_monbus_base_addr),
    .cfg_limit_addr      (cfg_monbus_limit_addr),
    .cfg_axi_pkt_mask    (cfg_axi_pkt_mask),
    // ... more protocol configuration
);

```

21.7 Common Issues

21.7.1 Issue 1: Missing Monitor Packets

Symptom: Expected events not appearing in FIFO

Root Causes: 1. Packet type masked in `cfg_pkt_mask` 2. Event masked in `cfg_*_mask` 3. FIFO full (packets dropped)

Solution: Check mask configuration and FIFO status.

21.7.2 Issue 2: Master Writes Stall

Symptom: Write FIFO fills up, backpressure to monitor bus

Root Causes: 1. AXI-Lite master target not responding 2. Address range exhausted (wrapping too fast)

Solution: Increase address range or reduce logging rate via masks.

21.8 Related Documentation

- **Parent:** [01_stream_core.md](#) - Provides monitor bus packets
 - **Packet Format:** Monitor Bus Protocol documentation
 - **Configuration:** [14_apb_config.md](#) - Register mapping
-

21.9 Revision History

MonBus AXI-Lite Group Revision History

Version	Date	Author	Description
0.90	2025-11-22	seang	Initial block specification
0.91	2026-01-02	seang	Added table captions and figure numbers

Last Updated: 2026-01-02

22 Chapter 3: External Interfaces

This chapter documents the external interfaces of the STREAM DMA engine (stream_top_ch8.sv).

22.1 Interface Specifications

STREAM has seven primary external interfaces:

22.1.1 01_axi4_interface_spec.md

- **AXI4 Master Interfaces (3 total)**
 - Descriptor fetch (256-bit, read-only)
 - Data read (parameterizable, default 512-bit)
 - Data write (parameterizable, default 512-bit)
- Protocol specifications and assumptions
- Transfer modes and alignment requirements
- Timing and performance characteristics

22.1.2 02_axil4_interface_spec.md

- **AXI4-Lite Interfaces (2 total, in monbus_axil_group)**
 - Error FIFO slave read (s_axil_err_*): 32-bit, read-only
 - Monitor data master write (m_axil_mon_*): 32-bit, write-only
- Used for monitor bus packet access and logging
- Part of monbus_axil_group integration (USE_AXI_MONITORS=1)

22.1.3 03_apb_interface_spec.md

- **APB4 Slave Interface**
- Full APB4 protocol with PSEL/PENABLE
- Optional CDC for asynchronous clock domains (CDC_ENABLE parameter)
- Address map: 0x000-0x03F kick-off, 0x100-0x3FF registers

22.1.4 05_monbus_interface_spec.md

- **MonBus Internal Protocol**
- Unified 64-bit monitoring bus (stream_core → monbus_axil_group)
- Event packet format and encoding
- Agent ID assignments for STREAM components
- Converted to AXI-Lite at top level

22.2 STREAM Interface Summary (stream_top_ch8.sv)

Interface	Type	Direction	Width	Purpose
APB4	Slave	Input	32-bit data, 12-bit addr	Configuration and kick-off
AXI4 Descriptor	Master Read	Output	256-bit	Descriptor fetch
AXI4 Data Read	Master Read	Output	512-bit (param)	Source data read
AXI4 Data Write	Master Write	Output	512-bit (param)	Destination data write
AXIL Error FIFO	Slave Read	Input	32-bit	Monitor error/interrupt FIFO
AXIL Monitor Write	Master Write	Output	32-bit	Monitor data to memory
IRQ	Output	Output	1-bit	Interrupt (error FIFO not empty)

Note: AXIL interfaces are active when USE_AXI_MONITORS=1. MonBus is internal (stream_core → monbus_axil_group).

Last Updated: 2025-12-01

Dependencies: - AMBA AXI4 Protocol Specification v4.0 - AMBA Monitor Bus Protocol (internal spec) - STREAM Architecture Overview

23 AXI4 Interface Specification for STREAM

23.1 Overview

This document defines the AXI4 interface specification for the STREAM DMA engine. STREAM uses three AXI4 master interfaces for memory access:

1. **Descriptor Fetch Master** - 256-bit read-only interface for fetching descriptors

2. **Data Read Master** - Parameterizable width (default 512-bit) for reading source data
3. **Data Write Master** - Parameterizable width (default 512-bit) for writing destination data

Note: This document focuses on STREAM-specific implementation details. For generic AXI4 protocol information, refer to the AMBA AXI4 Protocol Specification v4.0.

23.2 STREAM AXI4 Interface Summary

23.2.1 Number of Interfaces

STREAM implements **3 AXI4 Master Interfaces**:

Number of Interfaces

Interface	Type	Width	Channels	Purpose
Descriptor Fetch	Master Read	256-bit (fixed)	AR, R	Fetch descriptors from memory
Data Read	Master Read	512-bit (param)	AR, R	Read source data to SRAM
Data Write	Master Write	512-bit (param)	AW, W, B	Write SRAM data to destination

23.2.2 Interface Parameters

Interface Parameters

Parameter	Description	Valid Values	Default
DATA_WIDTH	AXI data bus width in bits	32, 64, 128, 256, 512, 1024	32
ADDR_WIDTH	AXI address bus width in bits	32, 64	37
ID_WIDTH	AXI ID tag width in bits	1-16	8
USER_WIDTH	AXI user signal width in bits	0-16	1

Parameter	Description	Valid Values	Default
	(optional)		

23.2.3 Interface Types and Transfer Modes

Interface Types and Transfer Modes

Interface Group	Channels	Transfer Mode	Address Alignment	Monitor	DCG	Notes
AXI4 Master Read-Split	AR, R	Flexible	4-byte	Yes	Yes	Dat a inte rfac es wit h chu nk ena bles
AXI4 Master Write-Split	AW, W, B	Flexible	4-byte	Yes	Yes	Dat a inte rfac es wit h chu nk ena bles
AXI4 Master Read	AR, R	Simplified	Bus-width	No	Yes	Con trol inte rfac es, full y alig

Interface Group	Channel Is	Transfer Mode	Address Alignment	Monitor	DCG	Notes
						ned
AXI4 Master Write	AW, W, B	Simplified	Bus-width	Yes	Yes	Control interfaces, fully aligned

23.2.4 Interface Group Parameter Settings

Interface Group Parameter Settings

Interface Group	Data Width	Address Width	ID Width	User Width	Transfer Mode
AXI4 Master Read-Split	512 bits	37 bits	8 bits	1 bit	Flexible
AXI4 Master Write-Split	512 bits	37 bits	8 bits	1 bit	Flexible
AXI4 Master Read	32 bits	37 bits	8 bits	1 bit	Simplified
AXI4 Master Write	32 bits	37 bits	8 bits	1 bit	Simplified

23.2.5 Interface Configuration Summary

Interface Configuration Summary

Interface Type	Interface Group	Transfer Mode	Alignment	Notes
Descriptor Sink	AXI4 Master Read	Simplified	32-bit aligned	Control interface
Descriptor Source	AXI4 Master Read	Simplified	32-bit aligned	Control interface

Interface Type	Interface Group	Transfer Mode	Alignment	Notes
Data Source	AXI4 Master Read-Split	Flexible	4-byte aligned	High-bandwidth data
Data Sink	AXI4 Master Write-Split	Flexible	4-byte aligned	High-bandwidth data
Program Sink	AXI4 Master Write	Simplified	32-bit aligned	Control interface
Program Source	AXI4 Master Write	Simplified	32-bit aligned	Control interface
Flag Sink	AXI4 Master Read	Simplified	32-bit aligned	Control interface
Flag Source	AXI4 Master Read	Simplified	32-bit aligned	Control interface

23.3 Transfer Mode Specifications

This specification defines two distinct transfer modes to optimize different interface types:

23.3.1 Mode 1: Simplified Transfer Mode (Control Interfaces)

Used for control interfaces (descriptors, programs, flags) that prioritize simplicity and predictable timing.

23.3.1.1 *Simplified Mode Assumptions*

Simplified Mode Assumptions

Aspect	Requirement
Address Alignment	All addresses aligned to full data bus width
Transfer Size	All transfers use maximum size equal to bus width
Burst Type	Incrementing bursts only ($AxBURST = 2'b01$)
Transfer Complexity	Maximum simplicity for predictable operation

23.3.2 Mode 2: Flexible Transfer Mode (Data Interfaces)

Used for high-bandwidth data interfaces that need to handle arbitrary address alignment while maintaining efficiency.

23.3.2.1 *Flexible Mode Assumptions*

Flexible Mode Assumptions

Aspect	Requirement
Address Alignment	4-byte aligned addresses (minimum alignment)
Transfer Sizes	Multiple sizes supported: 4, 8, 16, 32, 64 bytes
Burst Type	Incrementing bursts only ($AxBURST = 2'b01$)
Alignment Strategy	Progressive alignment to optimize bus utilization

23.4 Mode 1: Simplified Transfer Mode Specification

23.4.1 Assumption 1: Address Alignment to Data Bus Width

Assumption 1: Address Alignment to Data Bus Width

Aspect	Requirement
Alignment Rule	All AXI transactions aligned to data bus width
32-bit bus (4 bytes)	Address[1:0] must be 2'b00
64-bit bus (8 bytes)	Address[2:0] must be 3'b000
128-bit bus (16 bytes)	Address[3:0] must be 4'b0000
256-bit bus (32 bytes)	Address[4:0] must be 5'b00000
512-bit bus (64 bytes)	Address[5:0] must be 6'b000000

Aspect	Requirement
1024-bit bus (128 bytes)	Address[6:0] must be 7'b0000000
Rationale	Maximizes bus efficiency and eliminates unaligned access complexity

23.4.2 Assumption 2: Fixed Transfer Size

Assumption 2: Fixed Transfer Size

Aspect	Requirement
Transfer Size Rule	All transfers use maximum size equal to bus width
32-bit bus	AxSIZE = 3'b010 (4 bytes)
64-bit bus	AxSIZE = 3'b011 (8 bytes)
128-bit bus	AxSIZE = 3'b100 (16 bytes)
256-bit bus	AxSIZE = 3'b101 (32 bytes)
512-bit bus	AxSIZE = 3'b110 (64 bytes)
1024-bit bus	AxSIZE = 3'b111 (128 bytes)
Rationale	Maximizes bus utilization and simplifies address alignment

23.5 Mode 2: Flexible Transfer Mode Specification

23.5.1 Assumption 1: 4-Byte Address Alignment

Assumption 1: 4-Byte Address Alignment

Aspect	Requirement
Alignment Rule	All AXI transactions aligned to 4-byte boundaries
Address Constraint	Address[1:0] must be 2'b00
Rationale	Balances flexibility with AXI protocol requirements
Benefit	Supports arbitrary data placement while maintaining AXI compliance

23.5.2 Assumption 2: Multiple Transfer Sizes

Assumption 2: Multiple Transfer Sizes

Transfer Size	AxSIZE Value	Use Case
4 bytes	3'b010	Initial alignment, small transfers
8 bytes	3'b011	Progressive alignment
16 bytes	3'b100	Progressive alignment
32 bytes	3'b101	Progressive alignment
64 bytes	3'b110	Optimal full-width transfers
128 bytes	3'b111	Maximum efficiency (1024-bit bus)

23.5.3 Assumption 3: Progressive Alignment Strategy

Assumption 3: Progressive Alignment Strategy

Aspect	Requirement
Alignment Goal	Align to 64-byte boundaries for optimal bus utilization
Alignment Sequence	Use progressive sizes: $4 \rightarrow 8 \rightarrow 16 \rightarrow 32 \rightarrow 64$ bytes
Optimization	Choose largest possible transfer size at each step
Example	Address 0x1004: 4-byte transfer \rightarrow aligned to 0x1008, then larger transfers

23.5.4 Assumption 4: Chunk Enable Support

Assumption 4: Chunk Enable Support

Aspect	Requirement
Chunk Granularity	16 chunks of 32-bits each (512-bit bus)
Write Strobes	Generated from chunk enables for precise byte control
Alignment Transfers	Chunk patterns optimized for alignment

Aspect	Requirement
	sequences
Benefits	Precise data validity, optimal memory utilization

23.6 Common Protocol Assumptions (Both Modes)

23.6.1 Assumption 1: Incrementing Bursts Only

Assumption 1: Incrementing Bursts Only

Aspect	Requirement
Burst Type	All AXI bursts use incrementing address mode ($AxBURST = 2'b01$)
Excluded Types	No FIXED (2'b00) or WRAP (2'b10) bursts supported
Rationale	Simplifies address generation logic and covers most use cases
Benefit	Eliminates wrap boundary calculations and fixed address handling

23.6.2 Assumption 2: No Address Wraparound

Assumption 2: No Address Wraparound

Aspect	Requirement
Wraparound Rule	Transactions never wrap around top of address space
Example	No $0xFFFFFFFF \rightarrow 0x00000000$ transitions
Rationale	Real systems never allow this due to memory layout
Benefit	Dramatically simplified boundary crossing detection logic

23.7 Flexible Mode: Address Calculation Examples

23.7.1 Progressive Alignment Examples

Example 1: Address 0x1004 → 0x1040 (64-byte boundary)

Progressive Alignment Examples

Step	Address	Size	AxSIZE	Length	Bytes Transferred	Notes
1	0x1004	4 bytes	3'b010	1 beat	4	Initial alignment
2	0x1008	8 bytes	3'b011	1 beat	8	Progressive alignment
3	0x1010	16 bytes	3'b100	1 beat	16	Progressive alignment
4	0x1020	32 bytes	3'b101	1 beat	32	Progressive alignment
5	0x1040	64 bytes	3'b110	N beats	64×N	Optimal transfers

Example 2: Address 0x1010 → 0x1040 (64-byte boundary)

Progressive Alignment Examples

Step	Address	Size	AxSIZE	Length	Bytes Transferred	Notes
1	0x1010	16 bytes	3'b100	1 beat	16	Optimal initial size
2	0x1020	32 bytes	3'b101	1 beat	32	Progressive

Step	Address	Size	AxSIZE	Length	Bytes Transferred	Notes
						alignm ent
3	0x1040	64 bytes	3'b110	N beats	64×N	Optim al transf ers

23.7.2 Chunk Enable Pattern Examples

512-bit Bus with 16×32-bit chunks

Chunk Enable Pattern Examples

Transfer Size	Address Offset	Chunk Pattern	Description
4 bytes	0x04	16'h0002	Chunk 1 only
8 bytes	0x08	16'h000C	Chunks 2-3
16 bytes	0x10	16'h00F0	Chunks 4-7
32 bytes	0x20	16'hFF00	Chunks 8-15
64 bytes	0x00	16'hFFFF	All chunks

23.8 Master Read Interface Specification

23.8.1 Read Address Channel (AR)

Read Address Channel

Signal	Width	Direction	Simplified Mode	Flexible Mode	Description
ar_addr	ADDR_WIDTH	Master→Slave	Bus-width aligned	4-byte aligned	Read address
ar_len	8	Master→Slave	0-255	0-255	Burst length - 1
ar_size	3	Master→Slave	Fixed per bus	Variable: 4-64 bytes	Transfer size
ar_burst	2	Master→Slave	2'b01 (INCR only)	2'b01 (INCR only)	Burst type
ar_id	ID_WIDTH	Master→Slave	Any	Any	Transaction ID

Signal	Width	Direction	Simplified Mode	Flexible Mode	Description
ar_lock	1	Master→Slave	1'b0	1'b0	Lock type (normal)
ar_cach	4	Master→Slave	Implementation specific	4'b0011	Cache attributes
ar_prot	3	Master→Slave	Implementation specific	3'b000	Protection attributes
ar_qos	4	Master→Slave	4'b0000	4'b0000	Quality of Service
ar_region	4	Master→Slave	4'b0000	4'b0000	Region identifier
ar_user	USER_WIDTH	Master→Slave	Optional	Optional	User-defined
ar_valid	1	Master→Slave	0 or 1	0 or 1	Address valid
ar_ready	1	Slave→Master	0 or 1	0 or 1	Address ready

23.8.2 Read Data Channel (R)

Read Data Channel

Signal	Width	Direction	Description
r_data	DATA_WIDTH	Slave→Master	Read data
r_id	ID_WIDTH	Slave→Master	Transaction ID
r_resp	2	Slave→Master	Read response
r_last	1	Slave→Master	Last transfer in burst
r_user	USER_WIDTH	Slave→Master	User-defined (optional)
r_valid	1	Slave→Master	Read data valid
r_ready	1	Master→Slave	Read data ready

23.9 Master Write Interface Specification

23.9.1 Write Address Channel (AW)

Write Address Channel

Signal	Width	Direction	Simplified Mode	Flexible Mode	Description
aw_addr	ADDR_WIDTH	Master→Slave	Bus-width aligned	4-byte aligned	Write address
aw_len	8	Master→Slave	0-255	0-255	Burst length - 1
aw_size	3	Master→Slave	Fixed per bus	Variable: 4-64 bytes	Transfer size
aw_burst	2	Master→Slave	2'b01 (INCR only)	2'b01 (INCR only)	Burst type
aw_id	ID_WIDTH	Master→Slave	Any	Any	Transaction ID
aw_lock	1	Master→Slave	1'b0	1'b0	Lock type (normal)
aw_cacheline	4	Master→Slave	Implementation specific	4'b0011	Cache attributes
aw_prot	3	Master→Slave	Implementation specific	3'b000	Protection attributes
aw_qos	4	Master→Slave	4'b0000	4'b0000	Quality of Service
aw_region	4	Master→Slave	4'b0000	4'b0000	Region identifier
aw_user	USER_WIDTH	Master→Slave	Optional	Optional	User-defined
aw_valid	1	Master→Slave	0 or 1	0 or 1	Address valid
aw_ready	1	Slave→Master	0 or 1	0 or 1	Address ready

23.9.2 Write Data Channel (W)

Write Data Channel

Signal	Width	Direction	Simplified Mode	Flexible Mode	Description
w_data	DATA_WIDTH	Master→Slave	Write data	Write data	Write data
w_strb	DATA_WIDTH /8	Master→Slave	All 1's	From chunk enables	Write strobes
w_last	1	Master→Slave	Last transfer	Last transfer	Last transfer in burst
w_user	USER_WIDTH	Master→Slave	Optional	Optional	User-defined
w_valid	1	Master→Slave	0 or 1	0 or 1	Write data valid
w_ready	1	Slave→Master	0 or 1	0 or 1	Write data ready

23.9.3 Write Response Channel (B)

Write Response Channel

Signal	Width	Direction	Description
b_id	ID_WIDTH	Slave→Master	Transaction ID
b_resp	2	Slave→Master	Write response
b_user	USER_WIDTH	Slave→Master	User-defined (optional)
b_valid	1	Slave→Master	Response valid
b_ready	1	Master→Slave	Response ready

23.10 Address Calculation Rules

23.10.1 Simplified Mode Address Generation

Simplified Mode Address Generation

Parameter	Formula	Description
First Address	Must be bus-width aligned	Starting address
Address N	First_Address + (N × Bus_Width_Bytes)	Address for beat N
Alignment Check	(Address % Bus_Width_Bytes) == 0	Must always be true

23.10.2 Flexible Mode Address Generation

Flexible Mode Address Generation

Parameter	Formula	Description
First Address	Must be 4-byte aligned	Starting address
Address N	First_Address + (N × Transfer_Size)	Address for beat N
Alignment Check	(Address % 4) == 0	Must always be true
Progressive Alignment	Choose largest size ≤ bytes_to_boundary	Optimization strategy

23.10.3 4KB Boundary Considerations (Both Modes)

4KB Boundary Considerations

Validation Rule	Formula	Description
4KB Boundary	Bursts cannot cross 4KB (0x1000) boundaries	AXI specification
Max Burst Calculation	Max_Beats = (4KB - Start_Address %)	Burst limit

Validation Rule	Formula	Description
	4KB)) / Transfer_Size	
Boundary Check	Verify no 4KB crossings in burst	Mandatory validation

23.11 Write Strobe Generation

23.11.1 Simplified Mode Strobe Generation

Simplified Mode Strobe Generation

Bus Width	Strobe Pattern	Description
32-bit	4'b1111	All bytes valid
512-bit	64'hFFFFFFFFFFFF FFF	All bytes valid

23.11.2 Flexible Mode Strobe Generation

From Chunk Enables (512-bit bus example):

```
// Convert 16x32-bit chunk enables to 64x8-bit write strobes
for (int chunk = 0; chunk < 16; chunk++) begin
    if (chunk_enable[chunk]) begin
        w_strb[chunk*4 +: 4] = 4'hF; // 4 bytes per chunk
    end
end
```

Alignment Transfer Examples:

Flexible Mode Strobe Generation

Transfer Size	Chunk Pattern	Strobe Pattern	Description
4 bytes	16'h0001	64'h00000000 000000F	First 4 bytes
16 bytes	16'h000F	64'h00000000 00000FF	First 16 bytes
32 bytes	16'h0OFF	64'h00000000 0FFFFF	First 32 bytes
64 bytes	16'hFFFF	64'hFFFFFFF	All 64 bytes

Transfer Size	Chunk Pattern	Strobe Pattern	Description
		FFFFFFF	

23.12 Response Codes

23.12.1 Response Code Specification

Response Code Specification

Value	Name	Description	Simplified Mode Usage	Flexible Mode Usage
2'b00	OKA Y	Normal access success	Bus-width aligned access	4-byte aligned access
2'b01	EXO KAY	Exclusive access success	Bus-width aligned exclusive	4-byte aligned exclusive
2'b10	SLVE RR	Slave error	Slave-specific error	Slave-specific error
2'b11	DEC ERR	Decode error	Bus-width misalignment	4-byte misalignment

23.13 Implementation Benefits

23.13.1 Simplified Mode Benefits

Simplified Mode Benefits

Benefit Area	Simplification	Impact
Address Generation	Simple increment by bus width	Minimal logic complexity
Size Checking	No dynamic size validation	No validation logic needed
Strobe Generation	All strobes always high	Trivial implementation
Timing	Predictable single-size transfers	Optimal timing closure

23.13.2 Flexible Mode Benefits

Flexible Mode Benefits

Benefit Area	Capability	Impact
Data Placement	Arbitrary 4-byte aligned placement	Maximum flexibility
Bus Utilization	Progressive alignment optimization	High efficiency achieved
Chunk Control	Precise byte-level validity	Optimal memory utilization
Alignment Strategy	Automatic alignment to boundaries	Performance optimization

23.13.3 Mode Selection Guidelines

Mode Selection Guidelines

Interface Type	Recommended Mode	Rationale
High-bandwidth data	Flexible	Maximize throughput, handle arbitrary alignment
Control/status	Simplified	Predictable timing, minimal complexity
Descriptors	Simplified	Fixed-size structures, simple implementation
Programs	Simplified	Single-word writes, minimal overhead
Flags	Simplified	Fixed-size status, predictable behavior

23.14 Validation Requirements

23.14.1 Simplified Mode Validation

Simplified Mode Validation

Validation Area	Requirements
Address Alignment	Verify all addresses aligned to full bus width
Fixed Size	Verify AxSIZE always matches DATA_WIDTH
Full Strobes	Verify w_strb is always all 1's
Burst Type	Verify AxBURST is always 2'b01

23.14.2 Flexible Mode Validation

Flexible Mode Validation

Validation Area	Requirements
Address Alignment	Verify all addresses are 4-byte aligned
Size Validation	Verify AxSIZE matches actual transfer size
Chunk Consistency	Verify chunk enables match transfer size
Strobe Generation	Verify strobes generated correctly from chunks
Progressive Alignment	Verify alignment strategy optimization
Boundary Checking	Verify no 4KB boundary crossings

23.14.3 Common Validation

Common Validation

Validation Area	Requirements
No Wraparound	Verify addresses never wrap around
Incrementing Only	Verify AxBURST is always 2'b01
Response Handling	Verify proper response

Validation Area	Requirements
	generation
Error Conditions	Verify alignment violation responses

23.15 Performance Characteristics

23.15.1 Simplified Mode Performance

Simplified Mode Performance

Metric	Typical Value	Description
Latency	3 cycles	Address + Data + Response
Throughput	1 transfer per clock	Sustained rate
Efficiency	100%	Perfect bus utilization
Complexity	Minimal	Simple implementation

23.15.2 Flexible Mode Performance

Flexible Mode Performance

Metric	Alignment Phase	Optimized Phase	Description
Latency	3-15 cycles	3 cycles	Variable based on alignment
Throughput	Variable	1 transfer per clock	Depends on alignment pattern
Efficiency	25-100%	100%	Improves with alignment
Complexity	Moderate	Minimal	Progressive optimization

23.15.3 Performance Optimization Strategy

Flexible Mode Alignment Strategy: 1. **Initial Phase:** Use largest possible transfer size for current alignment 2. **Progressive Phase:** Incrementally align to larger boundaries

3. **Optimized Phase:** Use full bus-width transfers once aligned
4. **Result:** Achieve maximum efficiency while handling arbitrary starting addresses

23.16 This dual-mode approach provides the best of both worlds: simplified, predictable operation for control interfaces and flexible, high-performance operation for data interfaces.

23.17 Revision History

AXI4 Interface Specification for STREAM Revision History

Version	Date	Author	Description
0.90	2025-11-22	seang	Initial block specification
0.91	2026-01-02	seang	Added table captions and figure numbers

Last Updated: 2026-01-02

24 AXI4-Lite Interface Specification and Assumptions

24.1 Overview

This document defines the formal specification and assumptions for an AXI4-Lite interface implementation. AXI4-Lite is a subset of AXI4 optimized for simple, lightweight control register interfaces with inherent protocol simplifications.

24.2 Interface Summary

24.2.1 Number of Interfaces

- **2 Master Read Interface:** Single read channel for Monitor Packets (one for each Source and Sink)
- **2 Master Write Interface:** Single write channel for Monitor Packets plus a timestamp (one for each Source and Sink)

24.2.2 Interface Parameters

Interface Parameters

Parameter	Description	Valid Values	Default
DATA_WIDTH	AXI data bus width in bits	32, 64	32, 64
ADDR_WIDTH	AXI address bus width in bits	32, 64	37
STRB_WIDTH	Write strobe width	DATA_WIDTH/8	8

24.3 Core Protocol Assumptions

24.3.1 Inherent AXI4-Lite Simplifications

AXI4-Lite protocol inherently provides the following constraints:

Inherent AXI4-Lite Simplifications

Constraint	Description
Single Transfers Only	No burst transactions supported
No Transaction IDs	All transactions are in-order
Fixed Transfer Size	Always uses full data bus width
No User Signals	Simplified interface without user-defined extensions

24.3.2 Implementation Assumptions

24.3.2.1 *Assumption 1: Address Alignment to Data Bus Width*

Assumption 1: Address Alignment to Data Bus Width

Aspect	Requirement
Alignment Rule	All AXI4-Lite transactions aligned to data bus width
32-bit bus alignment	Address[1:0] must be 2'b00 (4-byte aligned)
64-bit bus alignment	Address[2:0] must be 3'b000 (8-byte aligned)
Rationale	Maximizes bus efficiency and eliminates unaligned access complexity

Aspect	Requirement
Benefit	Simplifies address decode and data steering logic

24.3.2.2 *Assumption 2: Fixed Transfer Size*

Assumption 2: Fixed Transfer Size

Aspect	Requirement
Transfer Size Rule	All transfers use maximum size equal to bus width
32-bit bus	AxSIZE = 3'b010 (4 bytes)
64-bit bus	AxSIZE = 3'b011 (8 bytes)
Rationale	Maximizes bus utilization and simplifies control logic
Benefit	No size decode logic required

24.3.2.3 *Assumption 3: No Address Wraparound*

Assumption 3: No Address Wraparound

Aspect	Requirement
Wraparound Rule	Transactions never wrap around top of address space
Rationale	Control register accesses never require wraparound behavior
Benefit	Simplified address boundary checking

24.3.2.4 *Assumption 4: Standard Protection Attributes*

Assumption 4: Standard Protection Attributes

Access Type	AxPROT Value	Description
Normal Access	3'b000	Data, secure, unprivileged
Privileged Access	3'b001	Data, secure, privileged
Rationale	Covers the majority of control register access patterns	

24.4 Master Read Interface Specification

24.4.1 Read Address Channel (AR)

Read Address Channel

Signal	Width	Direction	Required Values	Description
ar_addr	ADDR_WI DTH	Master→Slave	8-byte aligned	Read address
ar_prot	3	Master→Slave	Implementation specific	Protection attributes
ar_valid	1	Master→Slave	0 or 1	Address valid
ar_ready	1	Slave→Master	0 or 1	Address ready

24.4.2 Read Data Channel (R)

Read Data Channel

Signal	Width	Direction	Description
r_data	64	Slave→Master	Read data
r_resp	2	Slave→Master	Read response
r_valid	1	Slave→Master	Read data valid
r_ready	1	Master→Slave	Read data ready

24.4.3 AXI4-Lite Simplifications (Read)

AXI4-Lite Simplifications

Removed Signal	AXI4 Usage	AXI4-Lite Reason
ar_id	Transaction ID	Single transfers, no transaction IDs
ar_len	Burst length	Single transfers only
ar_size	Transfer size	Fixed to bus width
ar_burst	Burst type	Single transfers only
ar_lock	Lock type	Simplified access model
ar_cache	Cache attributes	Simplified memory model

Removed Signal	AXI4 Usage	AXI4-Lite Reason
ar_qos	Quality of Service	Simplified priority model
ar_region	Region identifier	Simplified address space
ar_user	User-defined	Simplified interface
r_id	Transaction ID	No transaction IDs needed
r_last	Last transfer	Single transfers only
r_user	User-defined	Simplified interface

24.5 Master Write Interface Specification

24.5.1 Write Address Channel (AW)

Write Address Channel

Signal	Width	Direction	Required Values	Description
aw_addr	ADDR_WI DTH	Master→Slave	8-byte aligned	Write address
aw_prot	3	Master→Slave	Implementation specific	Protection attributes
aw_valid	1	Master→Slave	0 or 1	Address valid
aw_ready	1	Slave→Master	0 or 1	Address ready

24.5.2 Write Data Channel (W)

Write Data Channel

Signal	Width	Direction	Description
w_data	32	Master→Slave	Write data
w_strb	4	Master→Slave	Write strobes (byte enables)
w_valid	1	Master→Slave	Write data valid
w_ready	1	Slave→Master	Write data ready

24.5.3 Write Response Channel (B)

Write Response Channel

Signal	Width	Direction	Description
b_resp	2	Slave→Master	Write response
b_valid	1	Slave→Master	Response valid
b_ready	1	Master→Slave	Response ready

24.5.4 AXI4-Lite Simplifications (Write)

AXI4-Lite Simplifications

Removed Signal	AXI4 Usage	AXI4-Lite Reason
aw_id	Transaction ID	Single transfers, no transaction IDs
aw_len	Burst length	Single transfers only
aw_size	Transfer size	Fixed to bus width
aw_burst	Burst type	Single transfers only
aw_lock	Lock type	Simplified access model
aw_cache	Cache attributes	Simplified memory model
aw_qos	Quality of Service	Simplified priority model
aw_region	Region identifier	Simplified address space
aw_user	User-defined	Simplified interface
w_last	Last transfer	Single transfers only
w_user	User-defined	Simplified interface
b_id	Transaction ID	No transaction IDs needed
b_user	User-defined	Simplified interface

24.6 Address Requirements

24.6.1 Address Alignment Rules

Address Alignment Rules

Alignment Type	Formula	Description
Valid Address	(Address % 4) == 0	Must be 8-byte aligned
Mandatory Alignment	Address[2:0] must be 3'b000	Per Assumption 1

24.6.2 Address Validation Examples

Address Validation Examples

Address Category	Examples	Status
Valid (8byte aligned)	0x1000, 0x1004, 0x1008, 0x100C	Accepted
Invalid (unaligned)	0x1001, 0x1002, 0x1003	DECERR response

24.7 Response Codes

24.7.1 Response Code Specification

Response Code Specification

Value	Name	Description	Usage in Control Registers
2'b00	OKAY	Normal access success	Successful register access
2'b01	EXOK AY	Exclusive access success	Not used in AXI4-Lite
2'b10	SLVER R	Slave error	Invalid register access
2'b11	DECER R	Decode error	Address decode failure or misalignment

24.7.2 Response Usage Guidelines

Response Usage Guidelines

Response Type	Usage	Description
OKAY	Normal completion	Successful register access
EXOKAY	Not applicable	AXI4-Lite doesn't support exclusive accesses
SLVERR	Register error	Invalid register operation
DECERR	Address error	Misalignment or decode failure per Assumption 1

24.8 Protection Signal Usage

24.8.1 Protection Signal Encoding

Protection Signal Encoding

Bit	Name	Description	Recommended Usage
[0]	Privilege d	0=Normal, 1=Privileged	Set based on processor mode
[1]	Non-secure	0=Secure, 1=Non-secure	Set based on security domain
[2]	Instructi on	0=Data, 1=Instruction	Always 0 for control registers

24.8.2 Common Protection Patterns

Common Protection Patterns

Pattern	AxPROT Value	Description
Normal Data Access	3'b000	Standard register access
Privileged Data Access	3'b001	Privileged register access
Debug Access	3'b010	Debug register access
Privileged Debug	3'b011	Privileged debug access

24.9 Implementation Benefits

24.9.1 Simplified Control Register Interface

Simplified Control Register Interface

Benefit Area	Simplification	Impact
Address Decode	Simple 8-byte aligned address comparison	Reduced decode logic
Transaction Handling	No burst or ID tracking required	Simplified state machines
Flow Control	Straightforward valid-ready handshakes	Reduced complexity
Response Generation	Simple OKAY/SLVERR/DECERR responses	Minimal response logic
Size Handling	Fixed 64-bit transfers only	No size decode needed

24.9.2 Address Decode Implementation

Address Decode Implementation

Implementation Aspect	Method	Benefit
4-byte Alignment Check	<code>addr[1:0] == 2'b00</code>	Simple bit masking
Address Range Check	<code>addr >= base && addr <= limit</code>	Simple comparisons
Combined Check	<code>alignment_ok && range_ok</code>	Single decode decision

24.9.3 Error Generation Logic

Error Generation Logic

Error Condition	Check	Response
Address Misalignment	<code>addr[1:0] != 2'b00</code>	Generate DECERR
Address Out of Range	<code>!addr_in_range(addr)</code>	Generate DECERR

Error Condition	Check	Response
Register Error	register_error_condition	Generate SLVERR
Normal Access	All checks pass	Generate OKAY

24.10 Timing Requirements

24.10.1 Handshake Protocol

Handshake Protocol

Protocol Rule	Requirement	Description
Valid-Ready Transfer	Transfer occurs when both VALID and READY are high	Standard AXI handshake
Valid Independence	VALID can be asserted independently of READY	Master controls valid
Ready Dependency	READY can depend on VALID state	Slave controls ready
Signal Stability	Once VALID asserted, all signals stable until READY	Data integrity

24.10.2 Channel Dependencies

Channel Dependencies

Dependency	Requirement	Description
Write Channels	AW and W channels are independent	Can be presented in any order
Write Response	B channel waits for both AW and W completion	Response dependency
Read Channels	R channel waits for AR channel completion	Response dependency
Transaction Ordering	Multiple outstanding transactions not supported	Inherent AXI4-Lite limitation

24.10.3 Reset Behavior

Reset Behavior

Reset Phase	Requirement	Description
Active Reset	aresetn is active-low reset signal	Standard AXI reset
Reset Requirements	All VALID signals deasserted during reset	Clean reset state
Reset Recovery	All VALID signals low after reset deassertion	Proper startup

24.11 Validation Requirements

24.11.1 Functional Validation

Functional Validation

Validation Area	Requirements
Address Alignment	Verify all accesses are 8-byte aligned per Assumption 1
Fixed Size	Verify all transfers are full 64-bit width per Assumption 2
Response Correctness	Verify appropriate response codes (DECERR for misaligned access)
Handshake Compliance	Verify all valid-ready handshakes
Register Behavior	Verify read/write register functionality
No Wraparound	Verify no address wraparound scenarios per Assumption 3

24.11.2 Timing Validation

Timing Validation

Validation Area	Requirements
Setup/Hold	Verify signal timing requirements
Reset Behavior	Verify proper reset sequence
Back-pressure	Verify ready signal behavior

Validation Area	Requirements
	under load

24.11.3 Error Injection Testing

Error Injection Testing

Test Type	Injection Method	Expected Response
Misaligned Address	Inject addresses with addr[2:0] != 0	DECERR response
Out of Range	Inject addresses outside valid range	DECERR response
Register Errors	Inject register-specific errors	SLVERR response

24.12 Example Transactions

24.12.1 64-bit Register Write

64-bit Register Write

Parameter	Value	Description
Bus Width	64 bits (8 bytes)	Data bus configuration
Target Address	0x1000 (8-byte aligned)	Valid aligned address
Write Data	0xDEADBEEFCA FEBABE	64-bit data value
Required Settings	aw_addr=0x100 0, aw_prot=3'b000, w_data=0xDEAD BEEFCAFEBABE, w_strb=8'b1111 1111	Transaction configuration

24.12.2 AW Transaction Flow

AW Transaction Flow

Step	Action	Signal States
1	Assert aw_valid	aw_valid=1, aw_addr=0x1000

Step	Action	Signal States
	with address	
2	Assert w_valid with data	w_valid=1, w_data=0xDEADBEEFCAFEBABE
3	Wait for handshakes	aw_ready=1, w_ready=1
4	Wait for response	b_valid=1, b_resp=OKAY
5	Complete transaction	b_ready=1

24.12.3 64-bit Register Read

64-bit Register Read

Parameter	Value	Description
Bus Width	64 bits (8 bytes)	Data bus configuration
Target Address	0x1008 (8-byte aligned)	Valid aligned address
Required Settings	ar_addr=0x1008 , ar_prot=3'b000	Transaction configuration

24.12.4 AR Transaction Flow

AR Transaction Flow

Step	Action	Signal States
1	Assert ar_valid with address	ar_valid=1, ar_addr=0x1008
2	Wait for address handshake	ar_ready=1
3	Wait for data response	r_valid=1, r_resp=OKAY
4	Complete transaction	r_ready=1
5	Capture data	r_data (64 bits)

24.12.5 Misaligned Address Example

Misaligned Address Example

Parameter	Value	Description
Bus Width	64 bits (8 bytes)	Data bus configuration
Target Address	0x1004 (misaligned)	Invalid address
Expected Behavior	Address decode detects misalignment → DECERR response → No register access	Error handling

24.13 Common Use Cases

24.13.1 Typical Applications

Typical Applications

Application	Description
Control/Status Registers	64-bit device configuration and monitoring
Memory-Mapped Peripherals	Simple register-based devices
Debug Interfaces	Debug and trace control registers
Configuration Space	PCIe configuration space access
Performance Counters	64-bit performance monitoring registers

24.13.2 Performance Considerations

Performance Considerations —

Consideration	Impact	Description
Latency	Single-cycle responses preferred	Simple registers
Throughput	Limited by single outstanding	AXI4-Lite constraint

Consideration	Impact	Description
Efficiency	transaction 64-bit transfers maximize data efficiency	Modern system optimization

24.14 Revision History

AXI4-Lite Interface Specification and Assumptions Revision History

Version	Date	Author	Description
0.90	2025-11-22	seang	Initial block specification
0.91	2026-01-02	seang	Added table captions and figure numbers

Last Updated: 2026-01-02

25 APB Programming Interface Specification for STREAM

25.1 Overview

This document defines the APB programming interface for the STREAM DMA engine.

Important: STREAM uses a **simplified programming interface** for descriptor kick-off, NOT a full APB slave. The interface uses standard valid/ready handshaking per channel to initiate descriptor-based transfers.

This interface is separate from the optional APB configuration interface that may be added in future implementations for runtime configuration of STREAM parameters.

25.2 Interface Summary

25.2.1 STREAM Programming Interface

STREAM provides a **simplified per-channel descriptor programming interface**:

STREAM Programming Interface

Signal Group	Per-Channel Signals	Protocol	Purpose
Programming	apb_valid[ch], apb_ready[ch], apb_addr[ch]	Valid/Ready handshake	Descriptor address input

Note: Despite the “apb_” prefix, this is NOT a full AMBA APB interface. It’s a simplified programming interface using APB naming convention for historical reasons.

25.2.2 Interface Parameters

Interface Parameters

Parameter	Description	Valid Values	Default
DATA_WIDTH	APB data bus width in bits	8, 16, 32	32
ADDR_WIDTH	APB address bus width in bits	16, 24, 32	32
STRB_WIDTH	Write strobe width (APB4 only)	DATA_WIDTH/8	4
NUM_SLAVES	Number of peripheral slaves	1-32	1
DEPTH	Internal buffer depth	2+	2

25.3 Core Protocol Assumptions

25.3.1 Inherent APB Simplifications

APB protocol inherently provides the following constraints:

1. **Single Master Only:** Only one bus master supported
2. **Non-Pipelined:** One transaction completes before next begins
3. **Simple 2-Phase Protocol:** Setup phase followed by access phase
4. **No Burst Transfers:** Only single transfers supported
5. **In-Order Completion:** No out-of-order transaction capability

25.3.2 Implementation Assumptions

25.3.2.1 *Assumption 1: Word-Aligned Access Only*

Assumption 1: Word-Aligned Access Only

Aspect	Requirement
Alignment Rule	All APB transfers are aligned to natural word boundaries
8-bit transfers	Byte aligned (no restriction)
16-bit transfers	2-byte aligned ($\text{PADDR}[0] = 0$)
32-bit transfers	4-byte aligned ($\text{PADDR}[1:0] = 2'b00$)
Rationale	Simplifies peripheral decode logic and ensures efficient access

25.3.2.2 *Assumption 2: Standard Transfer Sizes*

Assumption 2: Standard Transfer Sizes

Bus Width	Supported Transfer Sizes	Rationale
32-bit bus	8-bit, 16-bit, 32-bit	Covers typical register access patterns
16-bit bus	8-bit, 16-bit	Keeps decode logic simple
8-bit bus	8-bit only	Minimal complexity

25.3.2.3 *Assumption 3: Single-Cycle Default Operation*

Assumption 3: Single-Cycle Default Operation

Aspect	Requirement
Default Behavior	Most peripherals respond in a single cycle (PREADY tied high)
Optimization	PREADY optimization for simple registers
Exception	Complex peripherals may use PREADY for wait states
Rationale	Minimizes latency for control register access

25.4 Interface Signal Specification

25.4.1 Clock and Reset Signals

Clock and Reset Signals ### Address and Control Signals

Signal	IO	Description
src_clk	I	Source APB clock signal
snk_rdata[31:0]	I	Read data
src_wdata[31:0]	O	Write data

Address and Control Signals

Signal	Width	Direction	Required Values	Description
src_addr	11 bits	Master→Slave	Aligned per transfer size	Peripheral address (2KB space)
src_sel	1	Master→Slave	0 or 1	Peripheral select
src_enable	1	Master→Slave	0 or 1	Enable signal
src_write	1	Master→Slave	0 or 1	Write enable (1=write, 0=read)
snk_addr	11 bits	Master→Slave	Aligned per transfer size	Peripheral address (2KB space)
snk_sel	1	Master→Slave	0 or 1	Peripheral select
snk_enable	1	Master→Slave	0 or 1	Enable signal
snk_write	1	Master→Slave	0 or 1	Write enable (1=write, 0=read)

25.4.2 Data Signals

Data Signals

Signal	Width	Direction	Description
src_wdata	32	Master→Slave	Write data
src_rdata	32	Slave→Master	Read data
snk_wdata	32	Master→Slave	Write data
snk_rdata	32	Slave→Master	Read data

25.4.3 Response Signals

Response Signals

Signal	Width	Direction	Description
src_ready	1	Slave→Master	Transfer ready
src_slverr	1	Slave→Master	Slave error
snk_ready	1	Slave→Master	Transfer ready
snk_slverr	1	Slave→Master	Slave error

25.5 Address Space Configuration

25.5.1 Address Range Specification

Address Range Specification

Parameter	Value	Description
Address Width	11 bits (src_addr[10:0])	Full address space
Address Space	2KB (2048 bytes)	Total addressable space
Register Alignment	32-bit aligned (4-byte boundaries)	Address constraints
Usable Addresses	0x000 to 0x7FF	Valid address range

25.5.2 Address Decode Implementation

Address Decode Implementation

Register	Address Bits	Address Value	Description
reg0	src_addr[10:2]	0x000	First register

Register	Address Bits	Address Value	Description
		== 9'h000	
reg1	src_addr[10:2]	0x004	Second register
	== 9'h001		
reg2	src_addr[10:2]	0x008	Third register
	== 9'h002		
...	Up to 512 32-bit registers

25.6 Transaction Protocol

25.6.1 2-Phase Transaction States

2-Phase Transaction States

Phase	State	Duration	Signal Requirements
Setup (T1)	SETUP	1 clock cycle	src_sel=1, src_enable=0, address/data stable
Access (T2+)	ACCESS	1+ clock cycles	src_sel=1, src_enable=1, wait for src_ready
Idle	IDLE	Variable	src_sel=0, src_enable=0

25.6.2 State Transitions

State Transitions

Current State	Next State	Condition	Description
IDLE	SETUP	Transaction start	Master drives address and control
SETUP	ACCESS	Clock edge	Master asserts src_enable
ACCESS	IDLE	src_ready=1	Transaction completes
ACCESS	ACCESS	src_ready=0	Wait state continues

25.6.3 Transaction Timing Requirements

Transaction Timing Requirements

Timing Parameter	Requirement	Description
Setup Time	All master signals stable before rising src_clk	Signal stability
Hold Time	All master signals held after rising src_clk	Signal stability
Output Delay	Slave outputs valid after rising src_clk	Response timing

25.7 Advanced Features

25.7.1 Dynamic Clock Gating

Dynamic Clock Gating

Feature	Description	Benefit
Automatic Power Reduction	Clocks gated when no activity detected	Significant power savings
Configurable Idle Threshold	Programmable idle count before gating	Tunable power/performance
Multi-Domain Support	Independent gating for APB (PCLK) and backend (ACLK)	Flexible power management
Graceful Handoff	Ready signals forced to zero during gating	Protocol compliance
Activity Detection	Monitors valid signals across interfaces	Intelligent gating control

25.7.2 Clock Domain Crossing (CDC) Handshaking

Clock Domain Crossing (CDC) Handshaking

Feature	Description	Benefit
Arbitrary Frequency Support	Works across any clock frequency ratios	Design flexibility
Robust Reliability	Proven handshaking with	Zero data loss

Feature	Description	Benefit
	proper synchronization	
Command/Response Separation	Independent CDC paths	Maximum throughput
Skid Buffer Integration	Internal buffering prevents backpressure	Smooth operation
Deterministic Latency	Predictable timing characteristics	System predictability

25.7.3 Buffering and Flow Control

Buffering and Flow Control

Feature	Description	Benefit
Internal Skid Buffers	Configurable depth buffering	Improved performance
Backpressure Handling	Proper ready/valid handshaking	Flow control
Command Pipelining	Backend processes while APB handles responses	Efficiency
Response Queuing	Responses buffered for varying latencies	Latency tolerance

25.8 Error Handling

25.8.1 Error Response Specification

Error Response Specification

Condition	src_slverr	Description
Successful access	0	Normal completion
Address decode error	1	Invalid address
Protection violation	1	Insufficient privilege
Backend error	1	Downstream error condition

25.8.2 Error Response Timing

Error Response Timing

Timing Requirement	Description
src_slverr Validity	Must be valid when src_ready is asserted
Error Completion	Error response completes transaction normally
Master Responsibility	Master must check src_slverr status

25.9 Reset Behavior

25.9.1 Reset Requirements

Reset Requirements

Reset Phase	Signal Requirements	Description
Reset Assertion	All outputs to known states	Deterministic reset
Reset Deassertion	src_sel=0, src_enable=0 in first cycle	Clean startup
Reset Values	src_ready=0, src_slverr=0, src_rdata=0	Default states

25.10 Implementation Variants

25.10.1 Available Implementations

Available Implementations

Variant	Features	Use Case
Basic APB Slave	Single clock domain, internal buffering	Simple peripherals
CDC-Enabled APB Slave	Dual clock domain (PCLK/ACLK)	Mixed-frequency systems
Power-Optimized APB Slave	Dynamic clock gating	Power-sensitive designs

25.11 Performance Characteristics

25.11.1 Latency Specifications

Latency Specifications

Metric	Value	Description
Minimum Read Latency	2 clock cycles	Setup + access phases
Minimum Write Latency	2 clock cycles	Setup + access phases
CDC Additional Latency	2-6 clock cycles	Depends on clock relationships
Buffer Latency	Minimal	Skid buffer design

25.11.2 Throughput Specifications

Throughput Specifications

Metric	Value	Description
Single Transaction	2 clocks minimum	Basic transaction time
Back-to-back Transactions	Limited by src_ready	Depends on peripheral
CDC Throughput	Maintained across domains	Proper buffering

25.11.3 Power Consumption

Power Consumption

Power Category	Characteristics	Description
Active Power	Standard CMOS logic	Normal operation
Idle Power	Significantly reduced with clock gating	Power optimization
Gating Overhead	Minimal additional logic	Efficient implementation

25.12 Validation Requirements

25.12.1 Functional Validation

Functional Validation

Validation Area	Requirements
Protocol Compliance	Verify 2-phase setup/access sequence
Address Decode	Verify 11-bit address handling and alignment
Error Response	Verify src_slver generation and handling
Wait States	Verify src_ready functionality
CDC Operation	Verify cross-clock domain transfers
Clock Gating	Verify power management behavior
Buffer Operation	Verify skid buffer and flow control

25.12.2 Timing Validation

Timing Validation

Validation Area	Requirements
Setup/Hold	Verify signal timing requirements
Reset Sequence	Verify proper reset behavior
Multi-Clock	Verify CDC timing across frequency ranges
Gating Transitions	Verify clock enable/disable timing

25.13 Example Transactions

25.13.1 32-bit Register Write

32-bit Register Write

Clock Cycle	src_sel	src_enable	src_write	src_addr	src_wdat	src_ready	Phase
1	1	0	1	0x100	0xABCD	0	Setup
2	1	1	1	0x100	0xABCD	1	Access
3	0	0	-	-	-	0	Idle

25.13.2 32-bit Register Read with Wait State

32-bit Register Read with Wait State —

Clock Cycle	src_sel	src_enable	src_write	src_addr	src_rdata	src_ready	Phase
1	1	0	0	0x104	-	0	Setup
2	1	1	0	0x104	-	0	Access
3	1	1	0	0x104	0x1234	1	Complete
4	0	0	-	-	-	0	Idle

25.14 Revision History

APB Programming Interface Specification for STREAM Revision History

Version	Date	Author	Description
0.90	2025-11-22	seang	Initial block specification
0.91	2026-01-02	seang	Added table captions and figure numbers

26 Monitor Bus Architecture and Event Code Organization

26.1 STREAM-Specific Context

For STREAM DMA Engine: This document describes the generic Monitor Bus (MonBus) protocol used across all AMBA-based components. STREAM uses MonBus for unified event reporting from:

- Descriptor engines (8 sources, agent IDs 16-23)
- Schedulers (8 sources, agent IDs 48-55)
- Descriptor AXI monitor (agent ID 8)
- Read/Write AXI monitors (configurable agent IDs)

All STREAM events use Unit ID = 1 and follow the standard 64-bit packet format defined below.

26.2 Overview

The Monitor Bus architecture provides a unified, scalable framework for monitoring and error reporting across multiple bus protocols in complex SoC designs. This system supports AXI, APB, MNOC (Mesh Network on Chip), ARB (Arbiter), CORE, and custom protocols through a standardized 64-bit packet format with protocol-aware event categorization.

26.3 Interface Summary

26.3.1 Number of Interfaces

- **1 Monitor Bus Output Interface:** Unified 64-bit packet stream
- **Multiple Protocol Input Interfaces:** AXI, APB, MNOC, ARB, CORE, Custom protocol monitors
- **Local Memory Interface:** Error/interrupt packet storage
- **External Memory Interface:** Bulk packet storage

26.3.2 Interface Parameters

Interface Parameters

Parameter	Description	Valid Values	Default
PACKET_WIDTH	Monitor bus packet width	64	64
PROTOCOL_WIDTH	Protocol identifier	3	3

Parameter	Description	Valid Values	Default
	width		
EVENT_CODE_WI_DTH	Event code width	4	4
PACKET_TYPE_WIDTH	Packet type width	4	4
CHANNEL_ID_WI_DTH	Channel identifier width	6	6
UNIT_ID_WIDTH	Unit identifier width	4	4
AGENT_ID_WIDTH	Agent identifier width	8	8
EVENT_DATA_WIDHTH	Event data width	35	35

26.4 Core Design Assumptions

26.4.1 Assumption 1: Hierarchical Event Organization

Assumption 1: Hierarchical Event Organization

Aspect	Requirement
Organization Rule	Protocol → Packet Type → Event Code hierarchy
Event Space	Each protocol × packet type combination has exactly 16 event codes
Mapping	1:1 mapping between packet types and event codes
Rationale	Provides clear, scalable event organization

26.4.2 Assumption 2: Protocol Isolation

Assumption 2: Protocol Isolation

Aspect	Requirement
Isolation Rule	Each protocol owns its event space
Conflict Prevention	No cross-protocol event conflicts

Aspect	Requirement
Independent Evolution	Protocols can evolve independently
Rationale	Prevents interference and enables protocol-specific optimization

26.4.3 Assumption 3: Two-Tier Memory Architecture

Assumption 3: Two-Tier Memory Architecture

Aspect	Requirement
Local Storage	Critical events (errors/interrupts) stored locally
External Storage	Non-critical events routed to external memory
Routing Decision	Based on packet type configuration
Rationale	Balances immediate access with bulk storage needs

26.4.4 Assumption 4: Configurable Packet Routing

Assumption 4: Configurable Packet Routing

Aspect	Requirement
Routing Rule	Different packet types can route to different destinations
Configuration	Base/limit registers define routing per packet type
Priority Support	Configurable priority levels per packet type
Rationale	Enables flexible memory allocation and access patterns

26.5 Interface Signal Specification

26.5.1 Monitor Bus Output Interface

Monitor Bus Output Interface

Signal	Width	Direction	Description
mon_packet	64	Monitor→System	Monitor packet data
mon_valid	1	Monitor→System	Packet valid signal

Signal	Width	Direction	Description
mon_ready	1	System→Monitor	Ready to accept packet
mon_error	1	Monitor→System	Monitor error condition

26.5.2 Protocol Input Interfaces

Protocol Input Interfaces

Signal	Width	Direction	Description
axi_event	64	AXI Monitor→Bus	AXI event packet
axi_event_val_id	1	AXI Monitor→Bus	AXI event valid
axi_event_ready	1	Bus→AXI Monitor	Ready for AXI event
apb_event	64	APB Monitor→Bus	APB event packet
apb_event_val_id	1	APB Monitor→Bus	APB event valid
apb_event_ready	1	Bus→APB Monitor	Ready for APB event
mnoc_event	64	MNOC Monitor→Bus	MNOC event packet
mnoc_event_val_id	1	MNOC Monitor→Bus	MNOC event valid
mnoc_event_ready	1	Bus→MNOC Monitor	Ready for MNOC event
arb_event	64	ARB Monitor→Bus	ARB event packet
arb_event_val_id	1	ARB Monitor→Bus	ARB event valid
arb_event_ready	1	Bus→ARB Monitor	Ready for ARB event
core_event	64	CORE	CORE event

Signal	Width	Direction	Description
		Monitor→Bus	packet
core_event_va_lid	1	CORE	CORE event
		Monitor→Bus	valid
core_event_ready	1	Bus→CORE	Ready for
		Monitor	CORE event

26.5.3 Control and Status Signals

Control and Status Signals

Signal	Width	Direction	Description
clk	1	Input	System clock
resetn	1	Input	Active-low reset
monitor_enable	1	Input	Global monitor enable
packet_type_enables	16	Input	Per-type enable bits
local_memory_full	1	Output	Local memory full flag
external_memory_error	1	Output	External memory error

26.6 Packet Format and Field Allocation

26.6.1 64-bit Monitor Bus Packet Structure

64-bit Monitor Bus Packet Structure

Field	Bits	Width	Description
Packet Type	[63:60]	4	Event category (Error, Completion, etc.)
Protocol	[59:57]	3	Bus protocol (AXI=0, MNOC=1, APB=2, ARB=3, CORE=4)
Event Code	[56:53]	4	Specific events within category
Channel ID	[52:47]	6	Transaction/channel

Field	Bits	Width	Description
			identifier
Unit ID	[46:43]	4	Subsystem identifier
Agent ID	[42:35]	8	Module identifier
Event Data	[34:0]	35	Event-specific payload

26.6.2 Packet Type Definitions

Packet Type Definitions

Value	Name	Purpose	Applicable Protocols
0x0	Error	Protocol violations, response errors	All
0x1	Completion	Successful transaction completion	All
0x2	Threshold	Threshold crossed events	All
0x3	Timeout	Timeout conditions	All
0x4	Performance	Performance metrics	All
0x5	Credit	Credit management	MNOC only
0x6	Channel	Channel status	MNOC only
0x7	Stream	Stream events	MNOC only
0x8	Address Match	Address matching	AXI only
0x9	APB Specific	APB protocol events	APB only
0xA-0xE	Reserved	Future expansion	-
0xF	Debug	Debug and	All

Value	Name	Purpose	Applicable Protocols
		trace events	

26.7 Protocol-Specific Event Codes

26.7.1 AXI Protocol Events

26.7.1.1 Error Events (*PktType*Error + PROTOCOL_AXI)

Error Events

Code	Event Name	Description
0x0	AXI_ERR_RESP_SLVE_RR	Slave error response
0x1	AXI_ERR_RESP_DECE_RR	Decode error response
0x2	AXI_ERR_DATA_ORP_HAN	Data without command
0x3	AXI_ERR_RESP_ORP_HAN	Response without transaction
0x4	AXI_ERR_PROTOCOL	Protocol violation
0x5	AXI_ERR_BURST_LENGTH	Invalid burst length
0x6	AXI_ERR_BURST_SIZE	Invalid burst size
0x7	AXI_ERR_BURST_TYPE	Invalid burst type
0x8	AXI_ERR_ID_COLLISION	ID collision detected
0x9	AXI_ERR_WRITE_BEFORE_ADDR	Write data before address
0xA	AXI_ERR_RESP_BEFORE_DATA	Response before data complete
0xB	AXI_ERR_LAST_MISS	Missing LAST signal
0xC	AXI_ERR_STROBE_ERROR	Write strobe error

Code	Event Name	Description
	RROR	
0xD	AXI_ERR_RESERVED_D	Reserved
0xE	AXI_ERR_RESERVED_E	Reserved
0xF	AXI_ERR_USER_DEFI_NED	User-defined error

26.7.1.2 *Timeout Events (PktTypeTimeout + PROTOCOL_AXI)*

Timeout Events

Code	Event Name	Description
0x0	AXI_TIMEOUT_CMD	Command/Address timeout
0x1	AXI_TIMEOUT_DATA	Data timeout
0x2	AXI_TIMEOUT_RESP	Response timeout
0x3	AXI_TIMEOUT_HAN_DSHAKE	Handshake timeout
0x4	AXI_TIMEOUT_BURST_T	Burst completion timeout
0x5	AXI_TIMEOUT_EXCL_USIVE	Exclusive access timeout
0x6-0xE	Reserved	Future expansion
0xF	AXI_TIMEOUT_USER_DEFINED	User-defined timeout

26.7.1.3 *Performance Events (PktTypePerf + PROTOCOL_AXI)*

Performance Events

Code	Event Name	Description
0x0	AXI_PERF_ADDR_LATENCY	Address phase latency
0x1	AXI_PERF_DATA_LATENCY	Data phase latency
0x2	AXI_PERF_RESP_LAT	Response phase

Code	Event Name	Description
	ENCY	latency
0x3	AXI_PERF_TOTAL_L ATENCY	Total transaction latency
0x4	AXI_PERF_THROUG HPUT	Transaction throughput
0x5	AXI_PERF_ERROR_R ATE	Error rate
0x6	AXI_PERF_ACTIVE_C OUNT	Active transaction count
0x7	AXI_PERF_BANDWI DTH_UTIL	Bandwidth utilization
0x8	AXI_PERF_QUEUE_D EPTH	Average queue depth
0x9	AXI_PERF_BURST_EF FICIENCY	Burst efficiency metric
0xA-0xE	Reserved	Future expansion
0xF	AXI_PERF_USER_DEF INED	User-defined performance

26.7.2 APB Protocol Events

26.7.2.1 Error Events (*PktTypeError + PROTOCOL_APB*)

Error Events

Code	Event Name	Description
0x0	APB_ERR_PSLVERR	Peripheral slave error
0x1	APB_ERR_SETUP_VI OLATION	Setup phase protocol violation
0x2	APB_ERR_ACCESS_VI OLATION	Access phase protocol violation
0x3	APB_ERR_STROBE_E RROR	Write strobe error
0x4	APB_ERR_ADDR_DEC ODE	Address decode error

Code	Event Name	Description
0x5	APB_ERR_PROT_VIOLATION	Protection violation (PPROT)
0x6	APB_ERR_ENABLE_ERROR	Enable phase error (RROR)
0x7	APB_ERR_READY_ERROR	PREADY protocol error
0x8-0xE	Reserved	Future expansion
0xF	APB_ERR_USER_DEFINED	User-defined error

26.7.2.2 Timeout Events (*PktTypeTimeout + PROTOCOL_APB*)

Timeout Events

Code	Event Name	Description
0x0	APB_TIMEOUT_SETUP	Setup phase timeout
0x1	APB_TIMEOUT_ACCESS	Access phase timeout
0x2	APB_TIMEOUT_ENABLE	Enable phase timeout (PREADY stuck)
0x3	APB_TIMEOUT_PREADYSTUCK	PREADY stuck low
0x4	APB_TIMEOUT_TRANSFERS	Overall transfer timeout
0x5-0xE	Reserved	Future expansion
0xF	APB_TIMEOUT_USER_DEFINED	User-defined timeout

26.7.2.3 Completion Events (*PktTypeCompletion + PROTOCOL_APB*)

Completion Events

Code	Event Name	Description
0x0	APB_COMPL_TRANSACTION_COMPLETE	Transaction completed
0x1	APB_COMPL_READ_TRANSACTION	Read transaction

Code	Event Name	Description
	COMPLETE	complete
0x2	APB_COMPL_WRITE _COMPLETE	Write transaction complete
0x3-0xE	Reserved	Future expansion
0xF	APB_COMPL_USER_ DEFINED	User-defined completion

26.7.2.4 Threshold Events ($PktTypeThreshold + PROTOCOL_APB$)

Threshold Events

Code	Event Name	Description
0x0	APB_THRESH_LATE NCY	APB latency threshold
0x1	APB_THRESH_ERRO R_RATE	APB error rate threshold
0x2	APB_THRESH_ACCE S_COUNT	Access count threshold
0x3	APB_THRESH_BAND WIDTH	Bandwidth threshold
0x4-0xE	Reserved	Future expansion
0xF	APB_THRESH_USER_ DEFINED	User-defined threshold

26.7.2.5 Performance Events ($PktTypePerf + PROTOCOL_APB$)

Performance Events

Code	Event Name	Description
0x0	APB_PERF_READ_LA TENCY	Read transaction latency
0x1	APB_PERF_WRITE_L ATENCY	Write transaction latency
0x2	APB_PERF_THROUG HPUT	Transaction throughput
0x3	APB_PERF_ERROR_R ATE	Error rate

Code	Event Name	Description
0x4	APB_PERF_ACTIVE_COUNT	Active transaction count
0x5	APB_PERF_COMPLETED_TRANSACTION	Completed transaction count
0x6-0xE	Reserved	Future expansion
0xF	APB_PERF_USER_DEFINED	User-defined performance

26.7.2.6 Debug Events (PktTypeDebug + PROTOCOL_APB)

Debug Events

Code	Event Name	Description
0x0	APB_DEBUG_STATE_CHANGE	APB state changed
0x1	APB_DEBUG_SETUP_PHASE	Setup phase event
0x2	APB_DEBUG_ACCESS_PHASE	Access phase event
0x3	APB_DEBUG_ENABLE_PHASE	Enable phase event
0x4	APB_DEBUG_PSEL_TRACE	PSEL trace
0x5	APB_DEBUG_PENABLE_TRACE	PENABLE trace
0x6	APB_DEBUG_PREADY_TRACE	PREADY trace
0x7	APB_DEBUG_PPROT_TRACE	PPROT trace
0x8	APB_DEBUG_PSTRB_TRACE	PSTRB trace
0x9-0xE	Reserved	Future expansion
0xF	APB_DEBUG_USER_DEFINED	User-defined debug

26.7.3 MNOC Protocol Events

26.7.3.1 Error Events (*PktTypeError + PROTOCOL_MNOC*)

Error Events

Code	Event Name	Description
0x0	MNOC_ERR_PARITY	Parity error
0x1	MNOC_ERR_PROTOL OL	Protocol violation
0x2	MNOC_ERR_OVERFL OW	Buffer/Credit overflow
0x3	MNOC_ERR_UNDERFL LOW	Buffer/Credit underflow
0x4	MNOC_ERR_ORPNA N	Orphaned packet/ACK
0x5	MNOC_ERR_INVALI D	Invalid type/channel/payload
0x6	MNOC_ERR_HEADER _CRC	Header CRC error
0x7	MNOC_ERR_PAYLOA D_CRC	Payload CRC error
0x8	MNOC_ERR_SEQUEN CE	Sequence number error
0x9	MNOC_ERR_ROUTE	Routing error
0xA	MNOC_ERR_DEADLO CK	Deadlock detected
0xB-0xE	Reserved	Future expansion
0xF	MNOC_ERR_USER_D EFINED	User-defined error

26.7.3.2 Credit Events (*PktTypeCredit + PROTOCOL_MNOC*)

Credit Events

Code	Event Name	Description
0x0	MNOC_CREDIT_ALL	Credits allocated

Code	Event Name	Description
	LOCATED	
0x1	MNOC_CREDIT_CONSUMED	Credits consumed
0x2	MNOC_CREDIT_RETURNED	Credits returned
0x3	MNOC_CREDIT_OVERFLOW	Credit overflow detected
0x4	MNOC_CREDIT_UNDERFLOW	Credit underflow detected
0x5	MNOC_CREDIT_EXHAUSTED	All credits exhausted
0x6	MNOC_CREDIT_RESTORED	Credits restored
0x7	MNOC_CREDIT_EFFICIENCY	Credit efficiency metric
0x8	MNOC_CREDIT_LEAK	Credit leak detected
0x9-0xE	Reserved	Future expansion
0xF	MNOC_CREDIT_USER_DEFINED	User-defined credit event

26.7.3.3 Channel Events (*PktTypeChannel* + *PROTOCOL_MNOC*)

Channel Events

Code	Event Name	Description
0x0	MNOC_CHANNEL_OPENED	Channel opened
0x1	MNOC_CHANNEL_CLOSED	Channel closed
0x2	MNOC_CHANNEL_STALLED	Channel stalled
0x3	MNOC_CHANNEL_RESUMED	Channel resumed
0x4	MNOC_CHANNEL_CONGESTION	Channel congestion

Code	Event Name	Description
	ONGESTION	detected
0x5	MNOC_CHANNEL_PRIORITY	Channel priority change
0x6-0xE	Reserved	Future expansion
0xF	MNOC_CHANNEL_USER_DEFINED	User-defined channel event

26.7.3.4 Stream Events (PktTypeStream + PROTOCOL_MNOC)

Stream Events

Code	Event Name	Description
0x0	MNOC_STREAM_START	Stream started
	RT	
0x1	MNOC_STREAM_END	Stream ended (EOS)
	D	
0x2	MNOC_STREAM_PAUSED	Stream paused
	SE	
0x3	MNOC_STREAM_RESUME	Stream resumed
	UME	
0x4	MNOC_STREAM_OVERFLOW	Stream buffer overflow
	RFLOW	
0x5	MNOC_STREAM_UNDERFLOW	Stream buffer underflow
	DERFLOW	
0x6-0xE	Reserved	Future expansion
0xF	MNOC_STREAM_USER_DEFINED	User-defined stream event

26.7.4 ARB Protocol Events

26.7.4.1 Error Events (PktTypeError + PROTOCOL_ARB)

Error Events

Code	Event Name	Description
0x0	ARB_ERR_STARVATION	Client request starvation
	ON	
0x1	ARB_ERR_ACK_TIMEOUT	Grant ACK timeout

Code	Event Name	Description
	OUT	
0x2	ARB_ERR_PROTOCOL_VIOLATION	ACK protocol violation
0x3	ARB_ERR_CREDIT_VIOLATION	Credit system violation
0x4	ARB_ERR_FAIRNESS_VIOLATION	Weighted fairness violation
0x5	ARB_ERR_WEIGHT_UNDERFLOW	Weight credit underflow
0x6	ARB_ERR_CONCURRENT_GRANTS	Multiple simultaneous grants
0x7	ARB_ERR_INVALID_GRANT_ID	Invalid grant ID detected
0x8	ARB_ERR_ORPHAN_ACK	ACK without pending grant
0x9	ARB_ERR_GRANT_OVERLAP	Overlapping grant periods
0xA	ARB_ERR_MASK_ERROR	Round-robin mask error
0xB	ARB_ERR_STATE_MACHINE	FSM state error
0xC	ARB_ERR_CONFIGURATION	Invalid configuration
0xD-0xE	Reserved	Future expansion
0xF	ARB_ERR_USER_DEFINED	User-defined error

26.7.4.2 *Timeout Events (PktTypeTimeout + PROTOCOL_ARB)*

Timeout Events

Code	Event Name	Description
0x0	ARB_TIMEOUT_GRANT_ACK	Grant ACK timeout
0x1	ARB_TIMEOUT_REQ	Request held too

Code	Event Name	Description
0x2	UEST_HOLD	long
	ARB_TIMEOUT_WEI GHT_UPDATE	Weight update timeout
0x3	ARB_TIMEOUT_BLO	Block release timeout
	CK_RELEASE	
0x4	ARB_TIMEOUT_CRE	Credit update timeout
	DIT_UPDATE	
0x5	ARB_TIMEOUT_STAT	State machine timeout
	E_CHANGE	
0x6-0xE	Reserved	Future expansion
0xF	ARB_TIMEOUT_USER_ _DEFINED	User-defined timeout

26.7.4.3 Completion Events (*PktTypeCompletion* + *PROTOCOL_ARB*)

Completion Events

Code	Event Name	Description
0x0	ARB_COMPL_GRANT _ISSUED	Grant successfully issued
0x1	ARB_COMPL_ACK_R ECEIVED	ACK successfully received
0x2	ARB_COMPL_TRANS ACTION	Complete transaction (grant+ack)
0x3	ARB_COMPL_WEIGHT _UPDATE	Weight update completed
0x4	ARB_COMPL_CREDIT _CYCLE	Credit cycle completed
0x5	ARB_COMPL_FAIRNESS _PERIOD	Fairness analysis period
0x6	ARB_COMPL_BLOCK _PERIOD	Block period completed
0x7-0xE	Reserved	Future expansion
0xF	ARB_COMPL_USER_	User-defined

Code	Event Name	Description
	DEFINED	completion

26.7.4.4 Threshold Events ($PktTypeThreshold + PROTOCOL_ARB$)

Threshold Events

Code	Event Name	Description
0x0	ARB_THRESH_REQUEST_LATENCY	Request-to-grant latency threshold
0x1	ARB_THRESH_ACK_LATENCY	Grant-to-ACK latency threshold
0x2	ARB_THRESH_FAIRNESS_DEV	Fairness deviation threshold
0x3	ARB_THRESH_ACTIVE_REQUESTS	Active request count threshold
0x4	ARB_THRESH_GRANT_RATE	Grant rate threshold
0x5	ARB_THRESH EFFICIENCY	Grant efficiency threshold
0x6	ARB_THRESH_CREDIT_LOW	Low credit threshold
0x7	ARB_THRESH_WEIGHT_IMBALANCE	Weight imbalance threshold
0x8	ARB_THRESH_STARVATION_TIME	Starvation time threshold
0x9-0xE	Reserved	Future expansion
0xF	ARB_THRESH_USER_DEFINED	User-defined threshold

26.7.4.5 Performance Events ($PktTypePerf + PROTOCOL_ARB$)

Performance Events

Code	Event Name	Description
0x0	ARB_PERF_GRANT_ISSUED	Grant issued event
0x1	ARB_PERF_ACK RECEIVED	ACK received event
0x2	ARB_PERF_GRANT EFFICIENCY	Grant completion efficiency
0x3	ARB_PERF_FAIRNESS_METRIC	Fairness compliance metric
0x4	ARB_PERF_THROUGHPUT	Arbitration throughput

Code	Event Name	Description
0x5	ARB_PERF_LATENCY_AVG	Average latency measurement
0x6	ARB_PERF_WEIGHT_COMPLIANCE	Weight compliance metric
0x7	ARB_PERF_CREDIT_UTILIZATION	Credit utilization efficiency
0x8	ARB_PERF_CLIENT_ACTIVITY	Per-client activity metric
0x9	ARB_PERF_STARVATION_COUNT	Starvation event count
0xA	ARB_PERF_BLOCK EFFICIENCY	Block/unblock efficiency
0xB-0xE	Reserved	Future expansion
0xF	ARB_PERF_USER_DEFINED	User-defined performance

26.7.4.6 Debug Events (*PktTypeDebug + PROTOCOL_ARB*)

Debug Events

Code	Event Name	Description
0x0	ARB_DEBUG_STATE_CHANGE	Arbiter state machine change
0x1	ARB_DEBUG_MASK_UPDATE	Round-robin mask update
0x2	ARB_DEBUG_WEIGHT_CHANGE	Weight configuration change
0x3	ARB_DEBUG_CREDIT_UPDATE	Credit level update
0x4	ARB_DEBUG_CLIENT_MASK	Client enable/disable mask
0x5	ARB_DEBUG_PRIORITY_CHANGE	Priority level change
0x6	ARB_DEBUG_BLOCK_EVENT	Block/unblock event
0x7	ARB_DEBUG_QUEUE_STATUS	Request queue status

Code	Event Name	Description
0x8	ARB_DEBUG_COUNT ER_SNAPSHOT	Counter values snapshot
0x9	ARB_DEBUG_FIFO_S TATUS	FIFO status change
0xA	ARB_DEBUG_FAIRNE SS_STATE	Fairness tracking state
0xB	ARB_DEBUG_ACK_ST ATE	ACK protocol state
0xC-0xE	Reserved	Future expansion
0xF	ARB_DEBUG_USER_D EFINED	User-defined debug

26.7.5 CORE Protocol Events

26.7.5.1 Error Events (*PktType*Error + PROTOCOL_CORE)

Error Events

Code	Event Name	Description
0x0	CORE_ERR_DESCRIPTOR_MA LFORMED	Missing magic number (0x900dc0de)
0x1	CORE_ERR_DESCRIPTOR_BAD _ADDR	Invalid descriptor address
0x2	CORE_ERR_DATA_BAD_ADDR	Invalid data address (fetch or runtime)
0x3	CORE_ERR_FLAG_COMPARIS ON	Flag mask/compare mismatch
0x4	CORE_ERR_CREDIT_UNDERF LOW	Credit system violation
0x5	CORE_ERR_STATE_MACHINE	Invalid FSM state transition
0x6	CORE_ERR_DESCRIPTOR_ENG INE	Descriptor engine FSM error
0x7	CORE_ERR_FLAG_ENGINE	Flag engine FSM error
0x8	CORE_ERR_PROGRAM_ENGIN E	Program engine FSM error
0x9	CORE_ERR_DATA_ENGINE	Data engine error

Code	Event Name	Description
0xA	CORE_ERR_CHANNEL_INVAL_ID	Invalid channel ID
0xB	CORE_ERR_CONTROL_VIOLATION	Control register violation
0xC-0xE	Reserved	Future expansion
0xF	CORE_ERR_USER_DEFINED	User-defined error

26.7.5.2 *Timeout Events (PktTypeTimeout + PROTOCOL_CORE)*

Timeout Events

Code	Event Name	Description
0x0	CORE_TIMEOUT_DESCRIPTOR_FETCH	Descriptor fetch timeout
0x1	CORE_TIMEOUT_FLAG_RETRY	Flag comparison retry timeout
0x2	CORE_TIMEOUT_PROGRAM_WRITE	Program write timeout
0x3	CORE_TIMEOUT_DATA_TRANSFER	Data transfer timeout
0x4	CORE_TIMEOUT_CREDIT_WAIT	Credit wait timeout
0x5	CORE_TIMEOUT_CONTROL_WAIT	Control enable wait timeout
0x6	CORE_TIMEOUT_ENGINE_RESPONSE	Sub-engine response timeout
0x7	CORE_TIMEOUT_STATE_TRANSITION	FSM state transition timeout
0x8-0xE	Reserved	Future expansion
0xF	CORE_TIMEOUT_USER_DEFINED	User-defined timeout

26.7.5.3 Completion Events (*PktTypeCompletion* + *PROTOCOL_CORE*)

Completion Events

Code	Event Name	Description
0x0	CORE_COMPL_DESCRIPTOR_LOADED	Descriptor successfully loaded
0x1	CORE_COMPL_DESCRIPTOR_CHAIN	Descriptor chain completed
0x2	CORE_COMPL_FLAG_MATCHED	Flag comparison successful
0x3	CORE_COMPL_PROGRAM_COMPLETED	Post-programming completed
0x4	CORE_COMPL_DATA_TRANSFER	Data transfer completed
0x5	CORE_COMPL_CREDIT_CYCLE	Credit cycle completed
0x6	CORE_COMPL_CHANNEL_COMPLETE	Channel processing complete
0x7	CORE_COMPL_ENGINE_READY	Sub-engine ready
0x8-0xE	Reserved	Future expansion
0xF	CORE_COMPL_USER_DEFINED	User-defined completion

26.7.5.4 Threshold Events (*PktTypeThreshold* + *PROTOCOL_CORE*)

Threshold Events

Code	Event Name	Description
0x0	CORE_THRESH_DESCRIPTOR_QUEUE	Descriptor queue depth threshold
0x1	CORE_THRESH_CREDIT_LOW	Credit low threshold
0x2	CORE_THRESH_FLAG_RETRY_COUNT	Flag retry count threshold
0x3	CORE_THRESH_LATENCY	Processing latency threshold
0x4	CORE_THRESH_ERROR_RATE	Error rate threshold
0x5	CORE_THRESH_THROUGHPUT	Throughput threshold

Code	Event Name	Description
T		
0x6	CORE_THRESH_ACTIVE_CHANNELS	Active channel count threshold
0x7	CORE_THRESH_PROGRAM_LATENCY	Program write latency threshold
0x8	CORE_THRESH_DATA_RATE	Data transfer rate threshold
0x9-0xE	Reserved	Future expansion
0xF	CORE_THRESH_USER_DEFINED	User-defined threshold

26.7.5.5 Performance Events (*PktTypePerf* + *PROTOCOL_CORE*)

Performance Events

Code	Event Name	Description
0x0	CORE_PERF_END_OF_DATA	Stream continuation signal
0x1	CORE_PERF_END_OF_STREAM	Stream termination signal
0x2	CORE_PERF_ENTERING_IDLE	FSM returning to idle
0x3	CORE_PERF_CREDIT_INCREMENTED	Credit added by software
0x4	CORE_PERF_CREDIT_EXHAUSTED	Credit blocking execution
0x5	CORE_PERF_STATE_TRANSITION	FSM state change
0x6	CORE_PERF_DESCRIPTOR_ACTIVE	Data processing started
0x7	CORE_PERF_FLAG_RETRY	Flag comparison retry
0x8	CORE_PERF_CHANNEL_ENABLE_ENABLE	Channel enabled by software
0x9	CORE_PERF_CHANNEL_DISABLE_DISABLE	Channel disabled by software

Code	Event Name	Description
0xA	CORE_PERF_CREDIT_UTILIZATION	Credit utilization metric
0xB	CORE_PERF_PROCESSING_LATENCY	Total processing latency
0xC	CORE_PERF_QUEUE_DEPTH	Current queue depth
0xD-0xE	Reserved	Future expansion
0xF	CORE_PERF_USER_DEFINED	User-defined performance

26.7.5.6 Debug Events (*PktTypeDebug* + *PROTOCOL_CORE*)

Debug Events

Code	Event Name	Description
0x0	CORE_DEBUG_FSM_STATE_CHANGE	Descriptor FSM state change
0x1	CORE_DEBUG_DESCRIPTOR_CONTENT	Descriptor content trace
0x2	CORE_DEBUG_FLAG_ENGINE_STATE	Flag engine state trace
0x3	CORE_DEBUG_PROGRAM_ENGINE_STATE	Program engine state trace
0x4	CORE_DEBUG_CREDIT_OPERATION	Credit system operation
0x5	CORE_DEBUG_CONTROL_REGISTER	Control register access
0x6	CORE_DEBUG_ENGINE_HANDSHAKE	Engine handshake trace
0x7	CORE_DEBUG_QUEUE_STATUS	Queue status change
0x8	CORE_DEBUG_COUNTER_SNAPSHOT	Counter values snapshot
0x9	CORE_DEBUG_ADDRESS_TRACE	Address progression trace

Code	Event Name	Description
0xA	CORE_DEBUG_PAYLOAD_TRA CE	Payload content trace
0xB-0xE	Reserved	Future expansion
0xF	CORE_DEBUG_USER_DEFINE D	User-defined debug

26.8 Memory Architecture and Packet Routing

26.8.1 Two-Tier Memory Architecture

26.8.1.1 Local Error/Interrupt Memory

Local Error/Interrupt Memory

Characteristic	Description
Storage Types	Error Packets (Type 0x0) and Timeout Packets (Type 0x3)
Access Method	Immediate CPU access without memory subsystem delays
Capacity	Large enough to prevent overflow during error bursts
Priority	Critical events requiring immediate attention
Indexing	Fast search and retrieval mechanisms

26.8.1.2 Configurable External Memory

Configurable External Memory

Characteristic	Description
Storage Types	Performance, Completion, Threshold, Debug packets
Access Method	Base and limit registers define memory regions
Capacity	Bulk storage for non-critical events
DMA Support	Can be accessed via DMA for

Characteristic	Description
	efficient transfer
Time Stamping	32-bit timestamp appended when routing externally

26.8.2 Routing Configuration

26.8.2.1 *Base and Limit Registers*

Base and Limit Registers

Register Set	Purpose	Configuration
Completion Config	Type 0x1 routing	base_addr, limit_addr, enable, priority
Threshold Config	Type 0x2 routing	base_addr, limit_addr, enable, priority
Performance Config	Type 0x4 routing	base_addr, limit_addr, enable, priority
Debug Config	Type 0xF routing	base_addr, limit_addr, enable, priority

26.8.2.2 *Routing Decision Logic*

Routing Decision Logic

Packet Type	Destination	Address Calculation
Error (0x0)	Local Memory	local_error_write_pointer
Timeout (0x3)	Local Memory	local_error_write_pointer
Completion (0x1)	External Memory	completion_config.base_addr + offset
Performance (0x4)	External Memory	performance_config.base_addr + offset
Debug (0xF)	External Memory	debug_config.base_addr + offset

26.8.3 Address Space Management

26.8.3.1 *Memory Layout Example*

Memory Layout Example

Address Range	Usage	Description
0x1000_0000 - 0x1000_FFFF	Local Error	Immediate access storage

Address Range	Usage	Description
	Memory	
0x2000_0000 - 0x2001_FFFF	Performance Packets	External bulk storage
0x2010_0000 - 0x2011_FFFF	Completion Packets	External bulk storage
0x2020_0000 - 0x202F_FFFF	Debug Packets	External bulk storage

26.8.4 Transaction State and Bus Transaction Structure

26.8.4.1 *Transaction State Enumeration*

Transaction State Enumeration

State	Value	Description	Usage
TRANS_EMPT_Y	3'b000	Unused entry	Available slot
TRANS_ADDR_PHASE	3'b001	Address phase active (AXI) / Packet sent (MNOC) / Setup phase (APB)	Initial phase
TRANS_DATA_PHASE	3'b010	Data phase active (AXI) / Waiting for ACK (MNOC) / Access phase (APB)	Data transfer
TRANS_RESP_PHASE	3'b011	Response phase active (AXI) / ACK received (MNOC) / Enable phase (APB)	Response handling
TRANS_COMPLETE	3'b100	Transaction complete	Successful completion
TRANS_ERROR	3'b101	Transaction has error	Error condition
TRANS_ORPHANED	3'b110	Orphaned transaction	Missing components
TRANS_CREDIT_STALL	3'b111	Credit stall (MNOC only)	MNOC-specific stall

26.8.4.2 Enhanced Transaction Structure

Enhanced Transaction Structure

Field	Width	Description	Protocol Usage
valid	1	Entry is valid	All protocol's
protocol	3	Protocol type (AXI/MNOC/APB/ARB/C ORE)	All protocols
state	3	Transaction state	All protocols
id	32	Transaction ID (AXI) / Sequence (MNOC) / PSEL encoding (APB)	All protocols
addr	64	Transaction address / Channel addr / PADDR	All protocols
len	8	Burst length (AXI) / Packet count (MNOC) / Always 0 (APB)	AXI, MNOC
size	3	Access size (AXI) / Reserved (MNOC) / Transfer size (APB)	AXI, APB
burst	2	Burst type (AXI) / Payload type (MNOC) / PPROT[1:0] (APB)	All protocols

26.8.4.3 Phase Completion Flags

Phase Completion Flags

Flag	Description	Protocol Usage
cmd_receiv	Address phase received /	All protocols
ed	Packet sent / Setup phase	
data_starte	Data phase started / ACK	All protocols
d	expected / Access phase	
data_comp	Data phase completed / ACK	All protocols
leted	received / Enable phase	
resp_recei	Response received / Final	All protocols
ved	ACK / PREADY asserted	

26.8.4.4 Protocol-Specific Tracking Fields

Protocol-Specific Tracking Fields

Field	Width	Description	Protocol
channel	6	Channel ID (AXI ID / MNOC channel / PSEL bit position)	All protocols
eos_seen	1	EOS marker seen	MNOC only
parity_error	1	Parity error detected	MNOC only
credit_at_sta	8	Credits available at start	MNOC only
rt			
retry_count	3	Number of retries	MNOC only
desc_addr_match	1	Descriptor address match detected	AXI only
data_addr_match	1	Data address match detected	AXI only
apb_phase	2	Current APB phase	APB only
pslverr_seen	1	PSLVERR detected	APB only
pprot_value	3	PPROT value	APB only
pstrb_value	4	PSTRB value for writes	APB only
arb_grant_id	8	Current grant ID	ARB only
arb_weight	8	Current weight value	ARB only
core_fsm_state	3	Current CORE FSM state	CORE only
core_channel_id	6	CORE channel identifier	CORE only

26.8.5 APB Transaction Phases

APB Transaction Phases

Phase	Value	Description
APB_PHASE_IDLE	2'b00	Bus idle
APB_PHASE_SETUP	2'b01	Setup phase (PSEL asserted)
APB_PHASE_ACCES	2'b10	Access phase

Phase	Value	Description
S		(PENABLE asserted)
APB_PHASE_ENABL	2'b11	Enable phase
E		(waiting for PREADY)

26.8.6 APB Protection Types

APB Protection Types

Protection	Value	Description
APB_PROT_NORMA	3'b000	Normal access
L		
APB_PROT_PRIVILEGED	3'b001	Privileged access
APB_PROT_SECURE	3'b010	Secure access
APB_PROT_INSTRUCTION	3'b100	Instruction access

26.8.7 MNOC Payload Types

MNOC Payload Types

Payload	Value	Description
MNOC_PAYLOAD_CONFIG	2'b00	CONFIG_PKT
MNOC_PAYLOAD_TS	2'b01	TS_PKT
MNOC_PAYLOAD_RDA	2'b10	RDA_PKT
MNOC_PAYLOAD_RAW	2'b11	RAW_PKT

26.8.8 MNOC ACK Types

MNOC ACK Types

ACK Type	Value	Description
MNOC_ACK_STOP	2'b00	MSAP_STOP
MNOC_ACK_START	2'b01	MSAP_START

ACK Type	Value	Description
MNOC_ACK_CREDI T_ON	2'b10	MSAP_CREDIT_ON
MNOC_ACK_STOP_ AT_EOS	2'b11	MSAP_STOP_AT_EOS

26.8.9 ARB State Types

ARB State Types

State	Value	Description
ARB_STATE_IDLE	3'b000	Idle state
ARB_STATE_ARBITRATE	3'b001	Performing arbitration
ARB_STATE_GRANT	3'b010	Grant issued, waiting for ACK
ARB_STATE_BLOCKED	3'b011	Arbitration blocked
ARB_STATE_WEIGHT_UPD	3'b100	Weight update in progress
ARB_STATE_ERROR	3'b101	Error state

26.8.10 CORE State Types

CORE State Types

State	Value	Description
CORE_STATE_IDLE	3'b000	Idle state
CORE_STATE_DESC_FETCH	3'b001	Fetching descriptor
CORE_STATE_FLAG_CHECK	3'b010	Checking flag condition
CORE_STATE_PROG_RAM_WRITE	3'b011	Writing program
CORE_STATE_DATA_TRANSFER	3'b100	Transferring data
CORE_STATE_CRED_WAIT	3'b101	Waiting for credits

State	Value	Description
CORE_STATE_ERRO	3'b110	Error state
R		

26.9 Configuration and Control

26.9.1 Monitor Configuration Registers

26.9.1.1 *Global Configuration*

Global Configuration

Field	Width	Description
monitor_enable	1	Global monitor enable
error_local_enable	1	Enable local error storage
external_route_enable	1	Enable external routing
unit_id	4	Unit identifier
agent_id	8	Agent identifier
packet_type_enable	16	Per-type enable bits
s		

26.9.1.2 *Packet Type Enable Mapping*

Packet Type Enable Mapping

Bit	Enable	Description
0	PKT_ENABLE_ERRO	Enable error packets
	R	
1	PKT_ENABLE_COMP	Enable completion
	LETION	packets
2	PKT_ENABLE_THRES	Enable threshold
	HOLD	packets
3	PKT_ENABLE_TIME	Enable timeout
	OUT	packets
4	PKT_ENABLE_PERF	Enable performance
		packets

Bit	Enable	Description
5	PKT_ENABLE_CREDI T	Enable credit packets (MNOC)
6	PKT_ENABLE_CHAN NEL	Enable channel packets (MNOC)
7	PKT_ENABLE_STREA M	Enable stream packets (MNOC)
8	PKT_ENABLE_ADDR_ MATCH	Enable address match (AXI)
9	PKT_ENABLE_APB	Enable APB packets
15	PKT_ENABLE_DEBU G	Enable debug packets

26.9.2 Protocol-Specific Configuration

26.9.2.1 AXI Monitor Configuration

AXI Monitor Configuration

Field	Width	Description
active_trans_thresh old	16	Active transaction threshold
latency_threshold	32	Latency threshold (cycles)
addr_timeout_cnt	4	Address timeout count
data_timeout_cnt	4	Data timeout count
resp_timeout_cnt	4	Response timeout count
burst_boundary_ch eck	1	Enable burst boundary checking
address_match_ena ble	1	Enable address matching
desc_addr_match_b ase	64	Descriptor address match base
desc_addr_match_ mask	64	Descriptor address match mask

Field	Width	Description
data_addr_match_b	64	Data address match base
data_addr_match_	64	Data address match mask

26.9.2.2 MNOC Monitor Configuration

MNOC Monitor Configuration

Field	Width	Description
credit_low_threshol	8	Credit low threshold
d		
packet_rate_thresh	16	Packet rate threshold
old		
max_route_hops	8	Maximum routing hops
enable_credit_track	1	Enable credit tracking
ing		
enable_deadlock_d	1	Enable deadlock detection
etect		
deadlock_timeout	4	Deadlock detection timeout

26.9.2.3 ARB Monitor Configuration

ARB Monitor Configuration

Field	Width	Description
grant_timeout_cnt	16	Grant ACK timeout count
fairness_window	32	Fairness analysis window
weight_update_ena	1	Enable weight tracking
ble		
starvation_threshold	16	Starvation detection threshold
d		
efficiency_threshold	8	Grant efficiency threshold
d		

26.9.2.4 CORE Monitor Configuration

CORE Monitor Configuration

Field	Width	Description
descriptor_timeout	16	Descriptor fetch timeout count
_cnt		
flag_retry_limit	8	Maximum flag retry count
credit_low_threshold	8	Credit low threshold
processing_timeout	32	Processing timeout count
_cnt		
enable_descriptor_trace	1	Enable descriptor content tracing
enable_fsm_trace	1	Enable FSM state tracing

26.10 Validation Requirements

26.10.1 Functional Validation

Functional Validation

Validation Area	Requirements
Packet Format	Verify 64-bit packet structure and field encoding
Event Organization	Verify hierarchical event code organization
Protocol Isolation	Verify independent protocol event spaces
Routing Logic	Verify packet routing based on type and configuration
Memory Management	Verify local and external memory operations
Configuration	Verify register configuration and enable controls

26.10.2 Performance Validation

Performance Validation

Validation Area	Requirements
Throughput	Verify monitor bus can handle peak event rates
Latency	Verify low-latency path for critical events
Memory Efficiency	Verify efficient memory usage patterns
Power Consumption	Verify power-efficient operation

26.10.3 Error Handling Validation

Error Handling Validation

Validation Area	Requirements
Error Injection	Verify error detection and reporting
Overflow Handling	Verify behavior when memories fill
Configuration Errors	Verify invalid configuration detection
Recovery Mechanisms	Verify error recovery procedures

26.11 Usage Examples

26.11.1 Creating Monitor Packets

Creating Monitor Packets

Packet Type	Example Usage
AXI Error	Protocol=AXI, Type=Error, Code=AXI_ERR_RESP_SLVERR
MNOC Credit	Protocol=MNOC, Type=Credit, Code=MNOC_CREDIT_EXHAUSTED
APB Performance	Protocol=APB, Type=Performance, Code=APB_PERF_TOTAL_LATENCY

Packet Type	Example Usage
ARB Threshold	Protocol=ARB, Type=Threshold, Code=ARB_THRESH_FAIRNESS_DEV
CORE Completion	Protocol=CORE, Type=Completion, Code=CORE_COMPL_DESCRIPTOR_LOADED

26.11.2 Packet Decoding

Packet Decoding

Decoding Step	Method
Extract Type	packet[63:60]
Extract Protocol	packet[59:57]
Extract Event Code	packet[56:53]
Extract Channel ID	packet[52:47]
Extract Event Data	packet[34:0]

26.11.3 Monitor Bus Packet Helper Functions

26.11.3.1 *Packet Field Extraction*

Packet Field Extraction

Function	Return Type	Description
<code>get_packet_type(pkt)</code>	logic [3:0]	Extract packet type [63:60]
<code>get_protocol_type(pkt)</code>	protocol_type_t	Extract protocol [59:57]
<code>get_event_code(pkt)</code>	logic [3:0]	Extract event code [56:53]
<code>get_channel_id(pkt)</code>	logic [5:0]	Extract channel ID [52:47]
<code>get_unit_id(pkt)</code>	logic [3:0]	Extract unit ID [46:43]
<code>get_agent_id(pkt)</code>	logic [7:0]	Extract agent ID [42:35]
<code>get_event_data(pkt)</code>	logic [34:0]	Extract event data [34:0]

26.11.3.2 *Packet Creation Function*

Packet Creation Function

Function	Parameters	Description
create_monitor_packet()	packet_type, protocol, event_code, channel_id, unit_id, agent_id, event_data	Create complete 64-bit packet

26.11.3.3 *Event Code Creation Functions*

Event Code Creation Functions

Function	Parameter	Description
create_axi_error_event()	axi_error_code_t	Create AXI error event code
create_axi_timeout_event()	axi_timeout_code_t	Create AXI timeout event code
create_axi_completion_event()	axi_completion_code_t	Create AXI completion event code
create_axi_threshold_event()	axi_threshold_code_t	Create AXI threshold event code
create_axi_performance_event()	axi_performance_code_t	Create AXI performance event code
create_axi_address_match_event()	axi_addr_match_code_t	Create AXI address match event code
create_axi_debug_event()	axi_debug_code_t	Create AXI debug event code
create_apb_error_event()	apb_error_code_t	Create APB error event code
create_apb_timeout_event()	apb_timeout_code_t	Create APB timeout event code
create_apb_completion_event()	apb_completion_code_t	Create APB completion event code
create_mnoc_error_event()	mnoc_error_code_t	Create MNOC error event code
create_mnoc_timeout_event()	mnoc_timeout_code_t	Create MNOC timeout event

Function	Parameter	Description
t_event()		code
create_mnoc_completion_event()	mnoc_completion_code_t	Create MNOC completion event code
create_mnoc_credit_event()	mnoc_credit_code_t	Create MNOC credit event code
create_mnoc_channel_event()	mnoc_channel_code_t	Create MNOC channel event code
create_mnoc_stream_event()	mnoc_stream_code_t	Create MNOC stream event code
create_arb_error_event()	arb_error_code_t	Create ARB error event code
create_arb_timeout_event()	arb_timeout_code_t	Create ARB timeout event code
create_arb_completion_event()	arb_completion_code_t	Create ARB completion event code
create_arb_threshold_event()	arb_threshold_code_t	Create ARB threshold event code
create_arb_performance_event()	arb_performance_code_t	Create ARB performance event code
create_arb_debug_event()	arb_debug_code_t	Create ARB debug event code
create_core_error_event()	core_error_code_t	Create CORE error event code
create_core_timeout_event()	core_timeout_code_t	Create CORE timeout event code
create_core_completion_event()	core_completion_code_t	Create CORE completion event code
create_core_threshold_event()	core_threshold_code_t	Create CORE threshold event code
create_core_performance_event()	core_performance_code_t	Create CORE performance event code
create_core_debug_event()	core_debug_code_t	Create CORE debug event code

26.11.3.4 Validation Functions

Validation Functions

Function	Parameters	Description
<code>is_valid_event_for_packet_type()</code>	packet_type, protocol, event_code	Validate event code for packet type and protocol

26.11.3.5 String Functions for Debugging

String Functions for Debugging

Function	Parameter	Description
<code>get_axi_error_name()</code>	axi_error_code_t	Get human-readable AXI error name
<code>get_arb_error_name()</code>	arb_error_code_t	Get human-readable ARB error name
<code>get_core_error_name()</code>	core_error_code_t	Get human-readable CORE error name
<code>get_packet_type_name()</code>	logic [3:0]	Get packet type name string
<code>get_protocol_name()</code>	protocol_type_t	Get protocol name string
<code>get_event_name()</code>	packet_type, protocol, event_code	Get comprehensive event name

26.12 Debug and Monitoring Signals

26.12.1 Essential Debug Signals

Essential Debug Signals

Signal	Width	Purpose
<code>debug_packet_counts</code>	32×16	Packet count per type
<code>debug_protocol_counts</code>	32×5	Packet count per protocol
<code>debug_error_counts</code>	32	Total error packet count
<code>debug_local_memory_level</code>	16	Local memory usage level

Signal	Width	Purpose
debug_external_memory_level	16	External memory usage level

26.12.2 Performance Counters

Performance Counters

Counter	Width	Purpose
total_packets_processed	32	Total packets processed
packets_dropped	32	Packets dropped due to overflow
routing_errors	32	Routing configuration errors
memory_full_events	32	Memory full occurrences

26.13 Protocol Coverage Summary

26.13.1 Complete Protocol Event Matrix

Complete Protocol Event Matrix

Protocol	Error	Timeout	Completion	Threshold	Performance	Debug	Protocol-Specific
AXI	YES	YES	YES 16	YES 16	YES 16	YES	AddrMatch
	16	16				16	YES 16
MNOC	YES	YES	YES 16	NO 0	NO 0	NO	Credit/0
	16	16				0	Channel/Stream YES 48
APB	YES	YES	YES 16	YES 16	YES 16	YES	None
	16	16				16	
ARB	YES	YES	YES 16	YES 16	YES 16	YES	None
	16	16				16	
CORE	YES	YES	YES 16	YES 16	YES 16	YES	None
	16	16				16	

26.14 Total Event Codes: 544 defined across all protocols and packet types.

26.15 Revision History

Monitor Bus Architecture and Event Code Organization Revision History

Version	Date	Author	Description
0.90	2025-11-22	seang	Initial block specification
0.91	2026-01-02	seang	Added table captions and figure numbers

Last Updated: 2026-01-02

27 STREAM Register Map

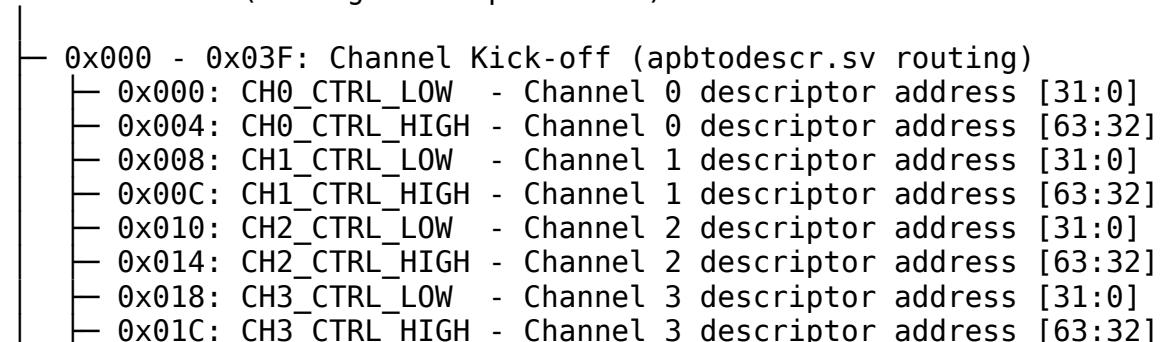
27.1 Overview

The STREAM DMA engine register interface consists of two distinct regions:

1. **Channel Kick-off Registers** (0x000 - 0x03F) - Direct routing to descriptor engines
2. **Configuration and Status Registers** (0x100 - 0xFFFF) - PeakRDL-generated register file

27.2 Address Space Layout

Base Address (configurable parameter)



```

    └── 0x020: CH4_CTRL_LOW - Channel 4 descriptor address [31:0]
    └── 0x024: CH4_CTRL_HIGH - Channel 4 descriptor address [63:32]
    └── 0x028: CH5_CTRL_LOW - Channel 5 descriptor address [31:0]
    └── 0x02C: CH5_CTRL_HIGH - Channel 5 descriptor address [63:32]
    └── 0x030: CH6_CTRL_LOW - Channel 6 descriptor address [31:0]
    └── 0x034: CH6_CTRL_HIGH - Channel 6 descriptor address [63:32]
    └── 0x038: CH7_CTRL_LOW - Channel 7 descriptor address [31:0]
    └── 0x03C: CH7_CTRL_HIGH - Channel 7 descriptor address [63:32]

└── 0x040 - 0x0FF: Reserved

└── 0x100 - 0x3FF: Configuration and Status Registers
    ├── 0x100 - 0x11F: Global Control and Status
    ├── 0x120 - 0x13F: Per-Channel Control
    ├── 0x140 - 0x16F: Per-Channel Status
    ├── 0x170 - 0x17F: Engine Completion and Error Status
    ├── 0x180 - 0x1FF: Monitor FIFO Status
    ├── 0x200 - 0x21F: Scheduler Configuration
    ├── 0x220 - 0x23F: Descriptor Engine Configuration
    ├── 0x240 - 0x25F: Descriptor AXI Monitor Configuration
    ├── 0x260 - 0x27F: Read Engine AXI Monitor Configuration
    ├── 0x280 - 0x29F: Write Engine AXI Monitor Configuration
    ├── 0x2A0 - 0x2AF: AXI Transfer Configuration
    └── 0x2B0 - 0x2BF: Performance Profiler Configuration

```

27.3 Register Details

27.3.1 Channel Kick-off Registers (0x000 - 0x03F)

These registers are NOT traditional registers. Writes are routed directly to descriptor engine APB ports via `apbtodescr.sv`.

Note: Descriptor addresses are 64-bit (ADDR_WIDTH parameter, default 64). On 32-bit APB bus, each channel requires TWO registers (LOW/HIGH).

Channel Kick-off Registers

Offset	Register	Type	Reset	Description
0x00 0	CH0_CTRL_L OW	WO	N/A	Channel 0 descriptor address [31:0]
0x00 4	CH0_CTRL_H IGH	WO	N/A	Channel 0 descriptor address [63:32]
0x00 8	CH1_CTRL_L OW	WO	N/A	Channel 1 descriptor address [31:0]

Offset	Register	Type	Reset	Description
0x00	CH1_CTRL_H	WO	N/A	Channel 1 descriptor address [63:32]
C	IGH			
0x01	CH2_CTRL_L	WO	N/A	Channel 2 descriptor address [31:0]
0	OW			
0x01	CH2_CTRL_H	WO	N/A	Channel 2 descriptor address [63:32]
4	IGH			
0x01	CH3_CTRL_L	WO	N/A	Channel 3 descriptor address [31:0]
8	OW			
0x01	CH3_CTRL_H	WO	N/A	Channel 3 descriptor address [63:32]
C	IGH			
0x02	CH4_CTRL_L	WO	N/A	Channel 4 descriptor address [31:0]
0	OW			
0x02	CH4_CTRL_H	WO	N/A	Channel 4 descriptor address [63:32]
4	IGH			
0x02	CH5_CTRL_L	WO	N/A	Channel 5 descriptor address [31:0]
8	OW			
0x02	CH5_CTRL_H	WO	N/A	Channel 5 descriptor address [63:32]
C	IGH			
0x03	CH6_CTRL_L	WO	N/A	Channel 6 descriptor address [31:0]
0	OW			
0x03	CH6_CTRL_H	WO	N/A	Channel 6 descriptor address [63:32]
4	IGH			
0x03	CH7_CTRL_L	WO	N/A	Channel 7 descriptor address [31:0]
8	OW			
0x03	CH7_CTRL_H	WO	N/A	Channel 7 descriptor address [63:32]
C	IGH			

Write Behavior: - Descriptor address is 64-bit, split across LOW and HIGH registers - Write to HIGH register triggers descriptor engine kick-off - LOW register write is buffered, HIGH register write initiates transfer - Both registers must be written in order (LOW then HIGH) - Write blocks until descriptor engine accepts (back-pressure) - Read not supported (returns error)

Example:

```
// Start DMA transfer on channel 0
// Descriptor at physical address 0x0000_0001_8000_0000 (64-bit)
```

```

// Write lower 32 bits first (buffered)
write32(BASE + CH0_CTRL_LOW, 0x8000_0000);

// Write upper 32 bits second (triggers kick-off)
write32(BASE + CH0_CTRL_HIGH, 0x0000_0001); // Blocks until accepted

// For descriptors in lower 4GB (typical case):
write32(BASE + CH0_CTRL_LOW, 0x8000_0000);
write32(BASE + CH0_CTRL_HIGH, 0x0000_0000); // Upper 32 bits = 0

```

27.3.2 Global Control and Status (0x100 - 0x11F)

27.3.2.1 GLOBAL_CTRL (0x100)

Master control register for entire STREAM engine.

GLOBAL_CTRL

Bits	Field	Type	Reset	Description
31:2	Reserved	RO	0x0	Reserved
1	GLOBAL_RS	RW	0	Global reset (self-clearing) T
0	GLOBAL_E	RW	0	Global enable (1=enabled, 0=disabled) N

Usage:

```

// Enable STREAM engine
write32(BASE + GLOBAL_CTRL, 0x1);

// Reset all channels
write32(BASE + GLOBAL_CTRL, 0x3); // Set both EN and RST
// RST self-clears after one cycle

```

27.3.2.2 GLOBAL_STATUS (0x104)

Overall system status.

GLOBAL_STATUS

Bits	Field	Type	Description
31:1	Reserved	RO	Reserved

Bits	Field	Type	Description
0	SYSTEM_IDLE	RO	System idle (all channels idle)

27.3.2.3 *VERSION* (0x108)

Version and configuration information (read-only).

VERSION

Bits	Field	Type	Value	Description
31:24	Reserved	RO	0x00	Reserved
23:16	NUM_CHANNELS	RO	0x08	Number of channels (8)
15:8	MAJOR	RO	0x00	Major version (0)
7:0	MINOR	RO	0x5A	Minor version (90 decimal = 0.90)

27.3.3 Per-Channel Control and Status (0x120 - 0x17F)

27.3.3.1 *CHANNEL_ENABLE* (0x120)

Per-channel enable control (bit vector).

CHANNEL_ENABLE

Bits	Field	Type	Reset	Description
31:8	Reserved	RO	0x0	Reserved
7:0	CH_EN	RW	0x00	Channel enable [7:0] (1=enabled)

Usage:

```
// Enable channels 0, 1, 2
write32(BASE + CHANNEL_ENABLE, 0x07);
```

```
// Disable channel 1, keep others
uint32_t val = read32(BASE + CHANNEL_ENABLE);
val &= ~(1 << 1); // Clear bit 1
write32(BASE + CHANNEL_ENABLE, val);
```

27.3.3.2 *CHANNEL_RESET* (0x124)

Per-channel reset control (bit vector, self-clearing).

CHANNEL_RESET

Bits	Field	Type	Reset	Description
31:8	Reserved	RO	0x0	Reserved
7:0	CH_RST	RW	0x00	Channel reset [7:0] (write 1 to reset)

Usage:

```
// Reset channels 0 and 3
write32(BASE + CHANNEL_RESET, 0x09); // Bits 0 and 3
// Self-clears after reset completes
```

27.3.3.3 CHANNEL_IDLE (0x140)

Per-channel idle status (bit vector, read-only).

CHANNEL_IDLE

Bits	Field	Type	Description
31:8	Reserved	RO	Reserved
7:0	CH_IDLE	RO	Channel idle [7:0] (1=idle, 0=active)

27.3.3.4 DESC_ENGINE_IDLE (0x144)

Per-channel descriptor engine idle status.

DESC_ENGINE_IDLE

Bits	Field	Type	Description
31:8	Reserved	RO	Reserved
7:0	DESC_IDLE	RO	Descriptor engine idle [7:0] (1=idle, 0=active)

27.3.3.5 SCHEDULER_IDLE (0x148)

Per-channel scheduler idle status.

SCHEDULER_IDLE

Bits	Field	Type	Description
31:8	Reserved	RO	Reserved
7:0	SCHED_IDL	RO	Scheduler idle [7:0] (1=idle, 0=active)

27.3.3.6 CH_STATE[0..7] (0x150 - 0x16C)

Per-channel scheduler FSM state (8 registers, stride 0x4).

CH_STATE[0..7]

Offset	Register	Bits	Field	Type	Description
0x150	CH0_STAT_E	31:7	Reserved	RO	Reserved
		6:0	STATE	RO	Channel 0 scheduler state (one-hot)
0x154	CH1_STAT_E	6:0	STATE	RO	Channel 1 scheduler state (one-hot)
0x158	CH2_STAT_E	6:0	STATE	RO	Channel 2 scheduler state (one-hot)
0x15C	CH3_STAT_E	6:0	STATE	RO	Channel 3 scheduler state (one-hot)
0x160	CH4_STAT_E	6:0	STATE	RO	Channel 4 scheduler state (one-hot)
0x164	CH5_STAT_E	6:0	STATE	RO	Channel 5 scheduler state (one-hot)
0x168	CH6_STAT_E	6:0	STATE	RO	Channel 6 scheduler state (one-hot)
0x16C	CH7_STAT_E	6:0	STATE	RO	Channel 7 scheduler state (one-hot)

State Encoding (One-Hot):

Bit 0 (0x01) = CH_IDLE	- Channel idle, waiting for descriptor
Bit 1 (0x02) = CH_FETCH_DESC	- Fetching descriptor from memory
Bit 2 (0x04) = CH_XFER_DATA	- Concurrent read AND write transfer
Bit 3 (0x08) = CH_COMPLETE	- Transfer complete
Bit 4 (0x10) = CH_NEXT_DESC	- Fetching next chained descriptor
Bit 5 (0x20) = CH_ERROR	- Error state
Bit 6 (0x40) = CH_RESERVED	- Reserved for future use

Note: Only ONE bit should be set at a time (one-hot encoding). Multiple bits set indicates a logic error.

27.3.4 Engine Completion and Error Status (0x170 - 0x17F)

27.3.4.1 SCHED_ERROR (0x170)

Per-channel scheduler error flags.

SCHED_ERROR

Bits	Field	Type	Description
31:8	Reserved	RO	Reserved
7:0	SCHED_ER	RO	Scheduler error bits [7:0] (1=error, 0=no error) R

27.3.4.2 AXI_RD_COMPLETE (0x174)

Per-channel AXI read engine completion status.

AXI_RD_COMPLETE

Bits	Field	Type	Description
31:8	Reserved	RO	Reserved
7:0	RD_COMPL	RO	Read completion bits [7:0] (1=complete, 0=pending) ETE

27.3.4.3 AXI_WR_COMPLETE (0x178)

Per-channel AXI write engine completion status.

AXI_WR_COMPLETE

Bits	Field	Type	Description
31:8	Reserved	RO	Reserved
7:0	WR_COMP	RO	Write completion bits [7:0] (1=complete, 0=pending) LETE

27.3.5 Monitor FIFO Status (0x180 - 0x1FF)

These registers are active when USE_AXI_MONITORS=1.

27.3.5.1 MON_FIFO_STATUS (0x180)

Monitor bus FIFO status indicators.

MON_FIFO_STATUS

Bits	Field	Type	Description
31:4	Reserved	RO	Reserved
3	MON_FIFO_UNF	RO	FIFO underflow detected (1=error) L

Bits	Field	Type	Description
2	MON_FIFO_OVF_L	RO	FIFO overflow detected (1=error)
1	MON_FIFO_EMP_TY	RO	FIFO empty (1=empty, 0=data available)
0	MON_FIFO_FULL	RO	FIFO full (1=full, 0=space available)

27.3.5.2 *MON_FIFO_COUNT* (0x184)

Monitor bus FIFO entry count.

MON_FIFO_COUNT

Bits	Field	Type	Description
31:16	Reserved	RO	Reserved
15:0	FIFO_COUNT	RO	Number of entries in FIFO [15:0]

27.3.6 Scheduler Configuration (0x200 - 0x21F)

27.3.6.1 *SCHED_TIMEOUT_CYCLES* (0x200)

Timeout threshold for scheduler (global for all channels).

SCHED_TIMEOUT_CYCLES

Bits	Field	Type	Reset	Description
31:16	Reserved	RO	0x0	Reserved
15:0	TIMEOUT_CYCLES	RW	1000	Timeout in clock cycles

27.3.6.2 *SCHED_CONFIG* (0x204)

Scheduler feature enables (global for all channels).

SCHED_CONFIG

Bits	Field	Type	Reset	Description
31:5	Reserved	RO	0x0	Reserved
4	PERF_EN	RW	0	Performance monitoring enable
3	COMPL_EN	RW	1	Completion reporting enable

Bits	Field	Type	Reset	Description
2	ERR_EN	RW	1	Error reporting enable
1	TIMEOUT_E	RW	1	Timeout detection enable
0	SCHED_EN	RW	1	Scheduler enable

27.3.7 Descriptor Engine Configuration (0x220 - 0x23F)

27.3.7.1 *DESCENG_CONFIG (0x220)*

Descriptor engine feature enables (global for all channels).

DESCENG_CONFIG

Bits	Field	Type	Reset	Description
31:6	Reserved	RO	0x0	Reserved
5:2	FIFO_THRES	RW	0x8	Prefetch FIFO threshold (4 bits) H
1	PREFETCH_E	RW	0	Prefetch enable N
0	DESCENG_EN	RW	1	Descriptor engine enable

27.3.7.2 *DESCENG_ADDR0_BASE (0x224)*

Base address for descriptor address range 0 (lower 32 bits).

DESCENG_ADDR0_BASE

Bits	Field	Type	Reset	Description
31:0	ADDR0_BA	RW	0x00000000	Address range 0 base SE

27.3.7.3 *DESCENG_ADDR0_LIMIT (0x228)*

Limit address for descriptor address range 0 (lower 32 bits).

DESCENG_ADDR0_LIMIT

Bits	Field	Type	Reset	Description
31:0	ADDR0_LIM	RW	0xFFFFFFF	Address range 0 limit

Bits	Field	Type	Reset	Description
	IT		FF	

27.3.7.4 *DESCENG_ADDR1_BASE* (0x22C)

Base address for descriptor address range 1 (lower 32 bits).

DESCENG_ADDR1_BASE

Bits	Field	Type	Reset	Description
31:0	ADDR1_BA	RW	0x00000000	Address range 1 base

SE 0

27.3.7.5 *DESCENG_ADDR1_LIMIT* (0x230)

Limit address for descriptor address range 1 (lower 32 bits).

DESCENG_ADDR1_LIMIT

Bits	Field	Type	Reset	Description
31:0	ADDR1_LIM	RW	0xFFFFFFF	Address range 1 limit

IT FF

27.3.8 Descriptor AXI Monitor Configuration (0x240 - 0x25F)

27.3.8.1 *DAXMON_ENABLE* (0x240)

Descriptor AXI master monitor enable controls.

DAXMON_ENABLE

Bits	Field	Type	Reset	Description
31:5	Reserved	RO	0x0	Reserved
4	PERF_EN	RW	0	Performance packet enable
3	TIMEOUT_E	RW	1	Timeout detection enable
N				
2	COMPL_EN	RW	0	Completion packet enable
1	ERR_EN	RW	1	Error detection enable
0	MON_EN	RW	1	Master enable for descriptor monitor

27.3.8.2 *DAXMON_TIMEOUT* (0x244)

Descriptor AXI monitor timeout threshold.

DAXMON_TIMEOUT

Bits	Field	Type	Reset	Description
31:0	TIMEOUT_CYCL_ES	RW	10000	Timeout in clock cycles

27.3.8.3 DAXMON_LATENCY_THRESH (0x248)

Descriptor AXI monitor latency threshold.

DAXMON_LATENCY_THRESH

Bits	Field	Type	Reset	Description
31:0	LATENCY_THR_ESH	RW	5000	Latency threshold in clock cycles

27.3.8.4 DAXMON_PKT_MASK (0x24C)

Descriptor AXI monitor packet type filtering.

DAXMON_PKT_MASK

Bits	Field	Type	Reset	Description
31:16	Reserved	RO	0x0	Reserved
15:0	PKT_MAS_K	RW	0xFFFF_F	Packet type mask (1=enable, 0=disable)

27.3.8.5 DAXMON_ERR_CFG (0x250)

Descriptor AXI monitor error selection and filtering.

DAXMON_ERR_CFG

Bits	Field	Type	Reset	Description
31:16	Reserved	RO	0x0	Reserved
15:8	ERR_MASK	RW	0xFF	Error type filtering mask
7:4	Reserved	RO	0x0	Reserved
3:0	ERR_SELE_CT	RW	0x0	Error type selection

27.3.8.6 DAXMON_MASK1 (0x254)

Descriptor AXI monitor timeout and completion masks.

DAXMON_MASK1

Bits	Field	Type	Reset	Description
31:16	Reserved	RO	0x0	Reserved
15:8	COMPL_MASK	RW	0x00	Completion mask
7:0	TIMEOUT_MA SK	RW	0xFF	Timeout mask

27.3.8.7 DAXMON_MASK2 (0x258)

Descriptor AXI monitor threshold and performance masks.

DAXMON_MASK2

Bits	Field	Type	Reset	Description
31:16	Reserved	RO	0x0	Reserved
15:8	PERF_MAS K	RW	0x00	Performance mask
7:0	THRESH_M ASK	RW	0xFF	Threshold mask

27.3.8.8 DAXMON_MASK3 (0x25C)

Descriptor AXI monitor address and debug masks.

DAXMON_MASK3

Bits	Field	Type	Reset	Description
31:16	Reserved	RO	0x0	Reserved
15:8	DEBUG_MA SK	RW	0x00	Debug mask
7:0	ADDR_MAS K	RW	0xFF	Address mask

27.3.9 Read Engine AXI Monitor Configuration (0x260 - 0x27F)

27.3.9.1 RDMON_ENABLE (0x260)

Read engine AXI master monitor enable controls.

RDMON_ENABLE

Bits	Field	Type	Reset	Description
31:5	Reserved	RO	0x0	Reserved
4	PERF_EN	RW	0	Performance packet enable
3	TIMEOUT_E N	RW	1	Timeout detection enable
2	COMPL_EN	RW	0	Completion packet enable
1	ERR_EN	RW	1	Error detection enable
0	MON_EN	RW	1	Master enable for read monitor

27.3.9.2 RDMON_TIMEOUT (0x264)

Read engine AXI monitor timeout threshold.

RDMON_TIMEOUT

Bits	Field	Type	Reset	Description
31:0	TIMEOUT_CYCL ES	RW	10000	Timeout in clock cycles

27.3.9.3 RDMON_LATENCY_THRESH (0x268)

Read engine AXI monitor latency threshold.

RDMON_LATENCY_THRESH

Bits	Field	Type	Reset	Description
31:0	LATENCY_THR ESH	RW	5000	Latency threshold in clock cycles

27.3.9.4 RDMON_PKT_MASK (0x26C)

Read engine AXI monitor packet type filtering.

RDMON_PKT_MASK

Bits	Field	Type	Reset	Description
31:16	Reserved	RO	0x0	Reserved
15:0	PKT_MAS K	RW	0xFFFF F	Packet type mask (1=enable, 0=disable)

27.3.9.5 RDMON_ERR_CFG (0x270)

Read engine AXI monitor error selection and filtering.

RDMON_ERR_CFG

Bits	Field	Type	Reset	Description
31:16	Reserved	RO	0x0	Reserved
15:8	ERR_MASK	RW	0xFF	Error type filtering mask
7:4	Reserved	RO	0x0	Reserved
3:0	ERR_SELE	RW	0x0	Error type selection CT

27.3.9.6 RDMON_MASK1 (0x274)

Read engine AXI monitor timeout and completion masks.

RDMON_MASK1

Bits	Field	Type	Reset	Description
31:16	Reserved	RO	0x0	Reserved
15:8	COMPL_MASK	RW	0x00	Completion mask
7:0	TIMEOUT_MA SK	RW	0xFF	Timeout mask

27.3.9.7 RDMON_MASK2 (0x278)

Read engine AXI monitor threshold and performance masks.

RDMON_MASK2

Bits	Field	Type	Reset	Description
31:16	Reserved	RO	0x0	Reserved
15:8	PERF_MAS K	RW	0x00	Performance mask
7:0	THRESH_M ASK	RW	0xFF	Threshold mask

27.3.9.8 RDMON_MASK3 (0x27C)

Read engine AXI monitor address and debug masks.

RDMON_MASK3

Bits	Field	Type	Reset	Description
31:16	Reserved	RO	0x0	Reserved
15:8	DEBUG_MA SK	RW	0x00	Debug mask
7:0	ADDR_MAS K	RW	0xFF	Address mask

27.3.10 Write Engine AXI Monitor Configuration (0x280 - 0x29F)

27.3.10.1 *WRMON_ENABLE (0x280)*

Write engine AXI master monitor enable controls.

WRMON_ENABLE

Bits	Field	Type	Reset	Description
31:5	Reserved	RO	0x0	Reserved
4	PERF_EN	RW	0	Performance packet enable
3	TIMEOUT_E N	RW	1	Timeout detection enable
2	COMPL_EN	RW	0	Completion packet enable
1	ERR_EN	RW	1	Error detection enable
0	MON_EN	RW	1	Master enable for write monitor

27.3.10.2 *WRMON_TIMEOUT (0x284)*

Write engine AXI monitor timeout threshold.

WRMON_TIMEOUT

Bits	Field	Type	Reset	Description
31:0	TIMEOUT_CYCL ES	RW	10000	Timeout in clock cycles

27.3.10.3 *WRMON_LATENCY_THRESH (0x288)*

Write engine AXI monitor latency threshold.

WRMON_LATENCY_THRESH

Bits	Field	Type	Reset	Description
31:0	LATENCY_THR ESH	RW	5000	Latency threshold in clock cycles

27.3.10.4 *WRMON_PKT_MASK* (0x28C)

Write engine AXI monitor packet type filtering.

WRMON_PKT_MASK

Bits	Field	Type	Reset	Description
31:16	Reserved	RO	0x0	Reserved
15:0	PKT_MAS K	RW	0xFFFF F	Packet type mask (1=enable, 0=disable)

27.3.10.5 *WRMON_ERR_CFG* (0x290)

Write engine AXI monitor error selection and filtering.

WRMON_ERR_CFG

Bits	Field	Type	Reset	Description
31:16	Reserved	RO	0x0	Reserved
15:8	ERR_MASK	RW	0xFF	Error type filtering mask
7:4	Reserved	RO	0x0	Reserved
3:0	ERR_SELE CT	RW	0x0	Error type selection

27.3.10.6 *WRMON_MASK1* (0x294)

Write engine AXI monitor timeout and completion masks.

WRMON_MASK1

Bits	Field	Type	Reset	Description
31:16	Reserved	RO	0x0	Reserved
15:8	COMPL_MASK	RW	0x00	Completion mask
7:0	TIMEOUT_MA SK	RW	0xFF	Timeout mask

27.3.10.7 WRMON_MASK2 (0x298)

Write engine AXI monitor threshold and performance masks.

WRMON_MASK2

Bits	Field	Type	Reset	Description
31:16	Reserved	RO	0x0	Reserved
15:8	PERF_MAS_K	RW	0x00	Performance mask
7:0	THRESH_MASK	RW	0xFF	Threshold mask

27.3.10.8 WRMON_MASK3 (0x29C)

Write engine AXI monitor address and debug masks.

WRMON_MASK3

Bits	Field	Type	Reset	Description
31:16	Reserved	RO	0x0	Reserved
15:8	DEBUG_MSK	RW	0x00	Debug mask
7:0	ADDR_MAS_K	RW	0xFF	Address mask

27.3.11 AXI Transfer Configuration (0x2A0 - 0x2AF)

27.3.11.1 AXI_XFER_CONFIG (0x2A0)

AXI read and write transfer burst sizes.

AXI_XFER_CONFIG

Bits	Field	Type	Reset	Description
31:16	Reserved	RO	0x0	Reserved
15:8	WR_XFER_B_EATS	RW	15	AXI write transfer beats (AWLEN: 0-255 = 1-256)
7:0	RD_XFER_BE_ATS	RW	15	AXI read transfer beats (ARLEN: 0-255 = 1-256)

Usage:

```
// Configure for 16-beat bursts (default)
write32(BASE + AXI_XFER_CONFIG, 0x0F0F);
```

```
// Configure for 64-beat bursts
write32(BASE + AXI_XFER_CONFIG, 0x3F3F);
```

```
// Configure for maximum 256-beat bursts
write32(BASE + AXI_XFER_CONFIG, 0xFFFF);
```

27.3.12 Performance Profiler Configuration (0x2B0 - 0x2BF)

27.3.12.1 *PERF_CONFIG (0x2B0)*

Performance profiler enable and mode controls.

PERF_CONFIG

Bits	Field	Type	Reset	Description
31:3	Reserved	RO	0x0	Reserved
2	PERF_CLEA	RW	0	Clear counters (write 1, self-clearing) R
1	PERF_MOD	RW	0	Mode: 0=count, 1=histogram E
0	PERF_EN	RW	0	Performance profiler enable

Usage:

```
// Enable performance profiler in count mode
write32(BASE + PERF_CONFIG, 0x01);
```

```
// Enable in histogram mode
write32(BASE + PERF_CONFIG, 0x03);
```

```
// Clear counters
write32(BASE + PERF_CONFIG, 0x05); // EN + CLEAR
// PERF_CLEAR self-clears after one cycle
```

27.4 Typical Usage Flow

27.4.1 Initialization

```
// 1. Global enable
write32(BASE + GLOBAL_CTRL, 0x1);

// 2. Configure scheduler
write32(BASE + SCHED_TIMEOUT_CYCLES, 10000);
write32(BASE + SCHED_CONFIG, 0x1F); // All features enabled

// 3. Configure descriptor engine
write32(BASE + DESCENG_CONFIG, 0x01); // Enable, no prefetch
write32(BASE + DESCENG_ADDR0_BASE, 0x8000_0000);
write32(BASE + DESCENG_ADDR0_LIMIT, 0x8FFF_FFFF);

// 4. Configure monitors (minimal reporting)
write32(BASE + DAXMON_CONFIG, 0x05); // Error + timeout only
write32(BASE + RDMON_CONFIG, 0x05);
write32(BASE + WRMON_CONFIG, 0x05);

// 5. Enable desired channels
write32(BASE + CHANNEL_ENABLE, 0xFF); // All 8 channels
```

27.4.2 Start Transfer

```
// Write 64-bit descriptor address to channel kick-off registers
// Descriptor at address 0x0000_0000_8000_0100 (64-bit)
write32(BASE + CH0_CTRL_LOW, 0x8000_0100); // Lower 32 bits
(buffered)
write32(BASE + CH0_CTRL_HIGH, 0x0000_0000); // Upper 32 bits
(triggers kick-off)
// Transfer starts immediately (blocks until descriptor engine ready)
```

27.4.3 Poll for Completion

```
// Check channel 0 idle status
while (!(read32(BASE + CHANNEL_IDLE) & 0x01)) {
    // Wait for channel 0 to become idle
}

// Or check scheduler state (one-hot encoding)
while ((read32(BASE + CH0_STATE) & 0x7F) != 0x01) {
    // Wait for CH_IDLE state (bit 0 = 0x01)
}
```

27.4.4 Error Handling

```
// Check all channel states for errors (one-hot encoding)
for (int ch = 0; ch < 8; ch++) {
    uint32_t state = read32(BASE + CH0_STATE + (ch * 4)) & 0x7F;
    if (state & 0x20) { // CH_ERROR (bit 5)
        // Reset channel
        write32(BASE + CHANNEL_RESET, 1 << ch);
    }
}
```

27.5 Register Summary Table

Register Summary Table

Offset Range	Description	Count	Type
0x000-0x03F	Channel kick-off registers (LOW/HIGH)	16	Write-routing
0x100-0x11F	Global control and status	3	RW/RO
0x120-0x13F	Per-channel control	2	RW
0x140-0x16F	Per-channel status	11	RO
0x170-0x17F	Engine completion and error status	3	RO
0x180-0x1FF	Monitor FIFO status	2	RO
0x200-0x21F	Scheduler configuration	2	RW
0x220-0x23F	Descriptor engine configuration	5	RW
0x240-0x25F	Descriptor AXI monitor configuration	8	RW
0x260-0x27F	Read engine AXI monitor configuration	8	RW
0x280-0x29F	Write engine AXI monitor configuration	8	RW
0x2A0-0x2AF	AXI transfer configuration	1	RW
0x2B0-0x2BF	Performance profiler configuration	1	RW

Total: 70 registers (16 kick-off + 54 config/status)

27.6 PeakRDL Generation

To generate SystemVerilog from the register definition:

```
cd projects/components/stream/rtl/stream_macro/
peakrdl regblock stream_regs.rdl -o generated/
```

This generates: - `stream_regs_pkg.sv` - Register definitions package - `stream_regs.sv` - APB slave register interface

Revision History:

PeakRDL Generation

Versio n	Date	Author	Description
1.0	2025-10-20	sean galloway	Initial creation
1.1	2025-12-01	sean galloway	Added complete monitor registers from RDL Added engine completion/error status (0x170) Added monitor FIFO status (0x180) Added AXI transfer config (0x2A0) Added performance profiler config (0x2B0)

28 Chapter 5: Programming Models

This chapter provides software developer guidance for using the STREAM DMA engine.

28.1 Contents

28.1.1 01_initialization.md

- Power-on initialization sequence
- Register configuration
- Channel setup
- Configuration presets (minimal, high-performance)

28.1.2 02_single_transfer.md

- Descriptor format (256-bit)
- Kick-off address map and write sequence
- Simple single-descriptor transfer

- C code examples with complete workflow

28.1.3 03_chained_transfers.md

- Multi-descriptor chains
- Descriptor linking via next_descriptor_ptr
- Chain termination methods
- Scatter-gather operations
- Prefetch mode for performance

28.1.4 04_multi_channel.md

- 8-channel concurrent operations
- Priority-based scheduling
- Resource sharing strategies
- Channel pooling and load balancing
- Performance considerations

28.1.5 05_error_handling.md

- Error types (AXI, timeout, internal)
- Error detection registers
- Recovery procedures
- Interrupt-based error handling
- Debug monitoring

28.2 Planned (Future)

28.2.1 06_performance_tuning.md

- Burst size selection
- Priority tuning
- SRAM depth considerations
- Maximizing throughput

28.2.2 07_software_examples.md

- Complete working examples
- Linux driver skeleton
- Bare-metal usage
- Common use cases

Status: Core programming guides complete (01-05)

Target Audience: - Software engineers integrating STREAM - Driver developers - System architects - Application developers

Last Updated: 2025-12-01

29 Chapter 6: Configuration Reference

Version: 0.90 **Last Updated:** 2025-11-22 **Purpose:** Complete reference for all STREAM configuration signals

29.1 Overview

STREAM provides comprehensive runtime configuration through APB registers and compile-time parameters. This chapter documents all configuration signals, their valid ranges, default values, and recommended settings for different use cases.

29.1.1 Configuration Categories

Configuration Categories

Category	Signals	Purpose
Channel Control	2	Enable/reset individual channels
Scheduler	6	Transfer scheduling and timeouts
Descriptor Engine	7	Descriptor fetch behavior
AXI Monitors	45 (15 per monitor × 3)	Debug/trace filtering
AXI Transfer	2	Burst configuration
Performance	3	Profiling and metrics
Total	65 configuration signals	Full system control

29.2 1. Channel Control Configuration

29.2.1 cfg_channel_enable[**NUM_CHANNELS**-1:0]

Type: Per-channel enable **Width:** 1 bit \times **NUM_CHANNELS** (default 8) **Default:** 8'hFF (all enabled)

Register: CHANNEL_ENABLE @ 0x120

Description: Enables or disables individual DMA channels. When disabled, the channel:
- Ignores descriptor kick-off requests
- Completes current transfer if in progress
- Enters idle state
- Remains in idle until re-enabled

Valid Values: - 1'b1: Channel enabled (can accept transfers)
- 1'b0: Channel disabled (ignores new transfers)

Use Cases:

```
// Enable only channels 0, 2, 4 (even channels)
cfg_channel_enable = 8'b01010101;
```

```
// Disable all channels (emergency stop)
cfg_channel_enable = 8'b00000000;
```

```
// Enable all channels (normal operation)
cfg_channel_enable = 8'b11111111;
```

Interaction with Global Enable: The final channel enable is: `cfg_channel_enable[i] & reg_global_ctrl_global_en`

29.2.2 cfg_channel_reset[**NUM_CHANNELS**-1:0]

Type: Per-channel reset **Width:** 1 bit \times **NUM_CHANNELS** (default 8) **Default:** 8'h00 (no resets active) **Register:** CHANNEL_RESET @ 0x124

Description: Asserts reset for individual channels without affecting other channels or global state. When asserted:
- Channel FSM returns to IDLE
- Pending transfers aborted
- SRAM buffers flushed
- Descriptor engine reset

Valid Values: - 1'b1: Channel in reset (clears state)
- 1'b0: Channel operating normally

Use Cases:

```
// Reset channel 3 after error
cfg_channel_reset = 8'b00001000;
wait_cycles(10);
cfg_channel_reset = 8'b00000000; // Release reset
```

```
// Reset all channels (soft reset)
```

```
cfg_channel_reset = 8'hFF;  
wait_cycles(10);  
cfg_channel_reset = 8'h00;
```

IMPORTANT: Assert reset for minimum 10 clock cycles to ensure complete FSM reset.

29.3 2. Scheduler Configuration

29.3.1 cfg_sched_timeout_cycles[15:0]

Type: Timeout threshold **Width:** 16 bits **Default:** 16'd1000 (1,000 cycles) **Register:** SCHED_TIMEOUT_CYCLES @ 0x200[15:0]

Description: Number of clock cycles before a channel operation times out. Applies to: - Descriptor fetch latency - AXI read/write response latency - Scheduler state transitions

Valid Range: 100 to 65535 cycles **Typical Values:** - Fast SRAM: 100-500 cycles - DDR3/DDR4: 1000-10000 cycles - High-latency: 10000-65535 cycles

Calculation:

Timeout cycles = (Expected latency × 10) + margin

Example for DDR4 @ 200 MHz:

- Expected read latency: 100 cycles (500 ns)
 - Safety margin: 10x
 - Timeout = $100 \times 10 = 1000$ cycles
-

29.3.2 cfg_sched_enable

Type: Global scheduler enable **Width:** 1 bit **Default:** 1'b1 (enabled) **Register:** SCHED_CONFIG @ 0x204[0]

Description: Master enable for all schedulers. When disabled, all channels stop scheduling new operations.

29.3.3 cfg_sched_timeout_enable

Type: Timeout detection enable **Width:** 1 bit **Default:** 1'b1 (enabled) **Register:** SCHED_CONFIG @ 0x204[1]

Description: Enables timeout detection for scheduler operations. Disable for simulation or known slow memory.

29.3.4 cfg_sched_err_enable

Type: Error detection enable **Width:** 1 bit **Default:** 1'b1 (enabled) **Register:** SCHED_CONFIG @ 0x204[2]

Description: Enables error packet generation for scheduler errors (AXI SLVERR, DECERR, timeouts).

29.3.5 cfg_sched_compl_enable

Type: Completion event enable **Width:** 1 bit **Default:** 1'b1 (enabled by default) **Register:** SCHED_CONFIG @ 0x204[3]

Description: Enables MonBus packets for transfer completion events. Generate high traffic, use sparingly.

29.3.6 cfg_sched_perf_enable

Type: Performance monitoring enable **Width:** 1 bit **Default:** 1'b0 (disabled by default) **Register:** SCHED_CONFIG @ 0x204[4]

Description: Enables performance profiling packets (latency, throughput metrics).

29.4 3. Descriptor Engine Configuration

29.4.1 cfg_desceng_enable

Type: Descriptor engine global enable **Width:** 1 bit **Default:** 1'b1 (enabled) **Register:** DESCENG_CTRL @ 0x220[0]

Description: Global enable for all descriptor engines. When disabled, descriptor fetch stops.

29.4.2 cfg_desceng_prefetch

Type: Prefetch enable **Width:** 1 bit **Default:** 1'b0 (disabled) **Register:** DESCENG_CTRL @ 0x220[1]

Description: Enables descriptor prefetching to hide fetch latency.

Prefetch Behavior: - Enabled: Fetch next descriptor while current transfer executes - Disabled: Wait for current transfer completion before fetching next

Performance Impact: - Prefetch ON: +15-30% throughput for chained descriptors - Prefetch OFF: Simpler logic, lower area

29.4.3 cfg_desceng_fifo_thresh[3:0]

Type: FIFO threshold **Width:** 4 bits **Default:** 4'h8 (8 entries) **Register:** DESCENG_CTRL @ 0x220[7:4]

Description: Number of entries in descriptor FIFO before asserting backpressure.

Valid Range: 1-15 entries **Typical Values:** - Low latency: 2-4 entries - Balanced: 8 entries (default)
- High throughput: 12-15 entries

29.4.4 cfg_desceng_addr0_base[31:0]

Type: Base address for descriptor region 0 **Width:** 32 bits **Default:** 32'h0000_0000 **Register:** DESCENG_ADDR0_BASE @ 0x224

Description: Base address of first descriptor memory region. Descriptors fetched from this region if within range.

Alignment: Must be aligned to descriptor size (256 bits = 32 bytes)

29.4.5 cfg_desceng_addr0_limit[31:0]

Type: Limit address for descriptor region 0 **Width:** 32 bits **Default:** 32'hFFFF_FFFF (no limit)
Register: DESCENG_ADDR0_LIMIT @ 0x228

Description: Upper limit of first descriptor region. Descriptors beyond this address use region 1.

29.4.6 cfg_desceng_addr1_base[31:0]

Type: Base address for descriptor region 1 **Width:** 32 bits **Default:** 32'h0000_0000 **Register:** DESCENG_ADDR1_BASE @ 0x22C

Description: Base address of second descriptor memory region (optional).

29.4.7 *cfg_desceng_addr1_limit[31:0]*

Type: Limit address for descriptor region 1 **Width:** 32 bits **Default:** 32'hFFFF_FFFF **Register:** DESCENG_ADDR1_LIMIT @ 0x230

Description: Upper limit of second descriptor region.

29.5 4. AXI Monitor Configuration

STREAM includes three independent AXI monitors with identical configuration sets:

1. **Descriptor AXI Monitor (*cfg_desc_mon_**)** - Monitors descriptor fetch AXI master
2. **Read Engine Monitor (*cfg_rdeng_mon_**)** - Monitors data read AXI master
3. **Write Engine Monitor (*cfg_wreng_mon_**)** - Monitors data write AXI master

Each monitor has 15 configuration signals with the same structure.

29.5.1 4.1 Descriptor AXI Monitor (*cfg_desc_mon_)**

Register Base: 0x240-0x25F

29.5.1.1 *cfg_desc_mon_enable*

Type: Monitor master enable **Width:** 1 bit **Default:** 1'b1 (enabled) **Register:** DAXMON_ENABLE @ 0x240[0]

Description: Master enable for descriptor AXI monitor. All monitor packets disabled when this is 0.

29.5.1.2 *cfg_desc_mon_err_enable*

Type: Error packet enable **Width:** 1 bit **Default:** 1'b1 (enabled) **Register:** DAXMON_ENABLE @ 0x240[1]

Description: Enables error packet generation (SLVERR, DECERR, protocol violations).

29.5.1.3 *cfg_desc_mon_perf_enable*

Type: Performance packet enable **Width:** 1 bit **Default:** 1'b0 (disabled) **Register:** DAXMON_ENABLE @ 0x240[2]

Description: Enables performance monitoring packets (latency, bandwidth).

WARNING: High packet rate - use only during debug.

29.5.1.4 *cfg_desc_mon_timeout_enable*

Type: Timeout detection enable **Width:** 1 bit **Default:** 1'b1 (enabled) **Register:** DAXMON_ENABLE @ 0x240[3]

Description: Enables timeout packet generation when transactions exceed threshold.

29.5.1.5 *cfg_desc_mon_timeout_cycles[31:0]*

Type: Timeout threshold **Width:** 32 bits **Default:** 32'd10000 **Register:** DAXMON_TIMEOUT @ 0x244

Description: Number of cycles before transaction times out.

29.5.1.6 *cfg_desc_mon_latency_thresh[31:0]*

Type: Latency threshold **Width:** 32 bits **Default:** 32'd1000 **Register:** DAXMON_LATENCY @ 0x248

Description: Latency threshold for performance warnings.

29.5.1.7 *cfg_desc_mon_pkt_mask[15:0]*

Type: Packet type filter **Width:** 16 bits (1 bit per packet type) **Default:** 16'h00FF (errors + completions) **Register:** DAXMON_PKT_MASK @ 0x24C[15:0]

Description: Bit mask to filter packet types. Only packet types with corresponding bit set are generated.

Packet Type Mapping:

Bit	Packet Type
0	AR command
1	AW command

2	R completion
3	W data
4	B response
5	Error
6	Timeout
7	Completion
8-15	Reserved

Examples:

```
// Only errors
cfg_desc_mon_pkt_mask = 16'h0020; // Bit 5
```

```
// Errors + timeouts
cfg_desc_mon_pkt_mask = 16'h0060; // Bits 5-6
```

```
// All packets
cfg_desc_mon_pkt_mask = 16'hFFFF;
```

29.5.1.8 *cfg_desc_mon_err_select[3:0]*

Type: Error type selector **Width:** 4 bits **Default:** 4'hF (all errors) **Register:** DAXMON_ERR_SELECT @ 0x24C[19:16]

Description: Selects which error types to monitor.

Error Type Bits:

Bit	Error Type
0	SLVERR
1	DECERR
2	Protocol violation
3	Reserved

29.5.1.9 *cfg_desc_mon_err_mask[7:0]*

Type: Error event filter **Width:** 8 bits **Default:** 8'hFF (all errors) **Register:** DAXMON_MASK1 @ 0x250[7:0]

Description: Bit mask for error packet generation (channel-specific filtering).

29.5.1.10 *cfg_desc_mon_timeout_mask[7:0]*

Type: Timeout channel filter **Width:** 8 bits **Default:** 8'hFF (all channels) **Register:** DAXMON_MASK1 @ 0x250[15:8]

Description: Channel mask for timeout packets. Set bit enables timeout detection for corresponding channel.

29.5.1.11 *cfg_desc_mon_compl_mask[7:0]*

Type: Completion channel filter **Width:** 8 bits **Default:** 8'h00 (no channels) **Register:** DAXMON_MASK1 @ 0x250[23:16]

Description: Channel mask for completion packets.

WARNING: Completion packets are high volume - enable only for specific channels.

29.5.1.12 *cfg_desc_mon_thresh_mask[7:0]*

Type: Threshold event filter **Width:** 8 bits **Default:** 8'hFF (all channels) **Register:** DAXMON_MASK2 @ 0x254[7:0]

Description: Channel mask for latency threshold exceedance packets.

29.5.1.13 *cfg_desc_mon_perf_mask[7:0]*

Type: Performance packet filter **Width:** 8 bits **Default:** 8'h00 (no channels) **Register:** DAXMON_MASK2 @ 0x254[15:8]

Description: Channel mask for performance monitoring packets.

29.5.1.14 *cfg_desc_mon_addr_mask[7:0]*

Type: Address-based filter **Width:** 8 bits **Default:** 8'hFF (all addresses) **Register:** DAXMON_MASK2 @ 0x254[23:16]

Description: Channel mask for address-range-based packet filtering.

29.5.1.15 *cfg_desc_mon_debug_mask[7:0]*

Type: Debug event filter **Width:** 8 bits **Default:** 8'h00 (no debug packets) **Register:** DAXMON_MASK2 @ 0x254[31:24]

Description: Channel mask for debug-level packets (verbose trace).

29.5.24.2 Read Engine Monitor (*cfg_rdeng_mon_)**

Register Base: 0x260-0x27F

All signals identical to Descriptor Monitor (Section 4.1), but applied to data read AXI master.

Key Differences: - Monitors data transfers (not descriptors) - Higher throughput → more packets - Recommended: Keep *cfg_rdeng_mon_compl_enable* = 0 unless debugging

29.5.34.3 Write Engine Monitor (*cfg_wreng_mon_)**

Register Base: 0x280-0x29F

All signals identical to Descriptor Monitor (Section 4.1), but applied to data write AXI master.

Key Differences: - Monitors write transactions (AW/W/B channels) - B response timing critical for performance - Recommended: Enable only error packets by default

29.6 5. AXI Transfer Configuration

29.6.1 *cfg_axi_rd_xfer_beats[7:0]*

Type: Read transfer size **Width:** 8 bits **Default:** 8'd16 (16 beats) **Register:** AXI_RD_XFER @ 0x2A0[7:0]

Description: Default number of beats per AXI read burst. Actual burst size may be less to respect 4KB boundaries.

Valid Range: 1-256 beats (AXI4 standard) **Typical Values:** - Small transfers: 4-8 beats - Balanced: 16 beats (default) - Large transfers: 32-64 beats - Maximum throughput: 128-256 beats

Calculation:

Transfer size (bytes) = beats × (DATA_WIDTH / 8)

Example for DATA_WIDTH = 512 bits:

- 16 beats = $16 \times 64 = 1024$ bytes (1 KB)
 - 64 beats = $64 \times 64 = 4096$ bytes (4 KB)
-

29.6.2 cfg_axi_wr_xfer_beats[7:0]

Type: Write transfer size **Width:** 8 bits **Default:** 8'd16 (16 beats) **Register:** AXI_WR_XFER @ 0x2A0[15:8]

Description: Default number of beats per AXI write burst.

Same constraints as **cfg_axi_rd_xfer_beats**

29.7 6. Performance Profiler Configuration

29.7.1 cfg_perf_enable

Type: Profiler enable **Width:** 1 bit **Default:** 1'b0 (disabled) **Register:** PERF_CTRL @ 0x2B0[0]

Description: Enables performance profiling for all channels. When enabled, profiler captures:
- Transfer start/end timestamps
- Latency per channel
- Throughput measurements
- Channel utilization

29.7.2 cfg_perf_mode

Type: Profiling mode **Width:** 1 bit **Default:** 1'b0 (timestamp mode) **Register:** PERF_CTRL @ 0x2B0[1]

Description: Selects profiling mode:
- 1'b0: Timestamp mode - Record absolute timestamps
- 1'b1: Elapsed time mode - Record delta times

Use Cases: - Timestamp: Correlate events across multiple blocks - Elapsed: Measure operation latencies

29.7.3 cfg_perf_clear

Type: Clear profiler state **Width:** 1 bit (write-only) **Default:** 1'b0 **Register:** PERF_CTRL @ 0x2B0[2]

Description: Write 1'b1 to clear profiler FIFOs and counters. Self-clearing (automatically returns to 0).

29.8 7. Configuration Presets

29.8.17.1 Minimal Configuration (Tutorial/Embedded)

Use Case: Educational, minimal logic, single-channel operation

```
// Channel control
cfg_channel_enable = 8'b00000001; // Only channel 0

// Scheduler
cfg_sched_enable = 1'b1;
cfg_sched_timeout_cycles = 16'd500;      // Short timeout (SRAM)
cfg_sched_timeout_enable = 1'b1;
cfg_sched_err_enable = 1'b1;
cfg_sched_compl_enable = 1'b0;           // Disable completion packets
cfg_sched_perf_enable = 1'b0;             // Disable performance
packets

// Descriptor engine
cfg_desceng_enable = 1'b1;
cfg_desceng_prefetch = 1'b0;              // No prefetch (simpler)
cfg_desceng_fifo_thresh = 4'h4;            // Small FIFO

// All monitors DISABLED (reduce logic)
cfg_desc_mon_enable = 1'b0;
cfg_rdeng_mon_enable = 1'b0;
cfg_wreng_mon_enable = 1'b0;

// AXI transfer
cfg_axi_rd_xfer_beats = 8'd8;           // Small bursts
cfg_axi_wr_xfer_beats = 8'd8;

// Performance profiler
cfg_perf_enable = 1'b0;                  // Disabled
```

29.8.27.2 Balanced Configuration (Typical FPGA)

Use Case: General-purpose DMA, moderate channels, balanced performance/area

```
// Channel control
cfg_channel_enable = 8'hFF;               // All 8 channels

// Scheduler
cfg_sched_enable = 1'b1;
cfg_sched_timeout_cycles = 16'd5000;       // DDR4 timeout
```

```

cfg_sched_timeout_enable = 1'b1;
cfg_sched_err_enable = 1'b1;
cfg_sched_compl_enable = 1'b0;           // Errors only
cfg_sched_perf_enable = 1'b0;

// Descriptor engine
cfg_desceng_enable = 1'b1;
cfg_desceng_prefetch = 1'b1;             // Enable prefetch
cfg_desceng_fifo_thresh = 4'h8;          // Balanced FIFO

// Descriptor monitor (errors only)
cfg_desc_mon_enable = 1'b1;
cfg_desc_mon_err_enable = 1'b1;
cfg_desc_mon_perf_enable = 1'b0;
cfg_desc_mon_timeout_enable = 1'b1;
cfg_desc_mon_timeout_cycles = 32'd10000;
cfg_desc_mon_pkt_mask = 16'h0060;        // Errors + timeouts

// Read/write monitors (errors only)
cfg_rdeng_mon_enable = 1'b1;
cfg_rdeng_mon_err_enable = 1'b1;
cfg_rdeng_mon_perf_enable = 1'b0;

cfg_wreng_mon_enable = 1'b1;
cfg_wreng_mon_err_enable = 1'b1;
cfg_wreng_mon_perf_enable = 1'b0;

// AXI transfer
cfg_axi_rd_xfer_beats = 8'd32;          // Moderate bursts
cfg_axi_wr_xfer_beats = 8'd32;

// Performance profiler
cfg_perf_enable = 1'b1;                  // Enable profiling
cfg_perf_mode = 1'b1;                    // Elapsed time mode

```

29.8.37.3 High-Performance Configuration (ASIC/Datacenter)

Use Case: Maximum throughput, all channels active, full monitoring

```

// Channel control
cfg_channel_enable = 8'hFF;              // All channels

// Scheduler
cfg_sched_enable = 1'b1;
cfg_sched_timeout_cycles = 16'd20000;      // High latency tolerance

```

```

cfg_sched_timeout_enable = 1'b1;
cfg_sched_err_enable = 1'b1;
cfg_sched_compl_enable = 1'b1;           // Full monitoring
cfg_sched_perf_enable = 1'b1;

// Descriptor engine
cfg_desceng_enable = 1'b1;
cfg_desceng_prefetch = 1'b1;
cfg_desceng_fifo_thresh = 4'hF;          // Max FIFO depth

// All monitors ENABLED with full profiling
cfg_desc_mon_enable = 1'b1;
cfg_desc_mon_err_enable = 1'b1;
cfg_desc_mon_perf_enable = 1'b1;         // Performance monitoring
cfg_desc_mon_timeout_enable = 1'b1;
cfg_desc_mon_timeout_cycles = 32'd20000;
cfg_desc_mon_pkt_mask = 16'hFFFF;        // All packet types

// Same for read/write monitors
cfg_rdeng_mon_enable = 1'b1;
cfg_rdeng_mon_err_enable = 1'b1;
cfg_rdeng_mon_perf_enable = 1'b1;

cfg_wreng_mon_enable = 1'b1;
cfg_wreng_mon_err_enable = 1'b1;
cfg_wreng_mon_perf_enable = 1'b1;

// AXI transfer
cfg_axi_rd_xfer_beats = 8'd128;         // Large bursts
cfg_axi_wr_xfer_beats = 8'd128;

// Performance profiler
cfg_perf_enable = 1'b1;
cfg_perf_mode = 1'b0;                    // Timestamp mode
(correlation)

```

29.8.47.4 Debug Configuration (Verbose Monitoring)

Use Case: Debugging integration issues, detailed trace analysis

```

// Enable specific channel for debug
cfg_channel_enable = 8'b00000001;        // Channel 0 only

// Scheduler
cfg_sched_enable = 1'b1;

```

```

cfg_sched_timeout_cycles = 16'd65535;      // Long timeout for debug
cfg_sched_timeout_enable = 1'b1;
cfg_sched_err_enable = 1'b1;
cfg_sched_compl_enable = 1'b1;               // All events
cfg_sched_perf_enable = 1'b1;

// Descriptor engine
cfg_desceng_enable = 1'b1;
cfg_desceng_prefetch = 1'b0;                  // Simpler for debug

// All monitors ENABLED with verbose trace
cfg_desc_mon_enable = 1'b1;
cfg_desc_mon_err_enable = 1'b1;
cfg_desc_mon_perf_enable = 1'b1;
cfg_desc_mon_timeout_enable = 1'b1;
cfg_desc_mon_pkt_mask = 16'hFFFF;            // ALL packets
cfg_desc_mon_compl_mask = 8'h01;              // Channel 0 completions
cfg_desc_mon_debug_mask = 8'h01;              // Channel 0 debug packets

// Same for read/write monitors
cfg_rdeng_mon_enable = 1'b1;
cfg_rdeng_mon_pkt_mask = 16'hFFFF;            // Verbose
cfg_rdeng_mon_compl_mask = 8'h01;

cfg_wreng_mon_enable = 1'b1;
cfg_wreng_mon_pkt_mask = 16'hFFFF;
cfg_wreng_mon_compl_mask = 8'h01;

// AXI transfer (small for debug)
cfg_axi_rd_xfer_beats = 8'd4;
cfg_axi_wr_xfer_beats = 8'd4;

// Performance profiler
cfg_perf_enable = 1'b1;
cfg_perf_mode = 1'b0;                         // Timestamps

```

29.9 8. Configuration Best Practices

29.9.18.1 Monitor Configuration Guidelines

General Rules:

- Start with errors only:** Enable only `cfg_*_mon_err_enable` initially. Add other packets as needed.

2. **Completion packets are expensive:** Only enable `cfg_*_mon_compl_enable` for specific channels during debug.
 3. **Performance packets flood MonBus:** Enable `cfg_*_mon_perf_enable` sparingly (1-2 channels maximum).
 4. **Use masks aggressively:** Set channel masks to enable monitoring only on channels of interest.
-

29.9.28.2 Timeout Configuration

Calculation Method:

Recommended timeout = (Expected latency × Safety factor) + Margin

Safety factor:

- SRAM: 2-5x
- DDR3/DDR4: 5-10x
- High-latency: 10-20x

Margin: +100 cycles minimum

Examples:

SRAM @ 200 MHz:

- Expected: 20 cycles (100 ns)
- Safety: 5x
- Timeout: $20 \times 5 + 100 = 200$ cycles

DDR4 @ 200 MHz:

- Expected: 100 cycles (500 ns)
 - Safety: 10x
 - Timeout: $100 \times 10 + 100 = 1100$ cycles
-

29.9.38.3 Prefetch Configuration

Enable prefetch when: - Descriptor chains > 2 descriptors - Memory latency > 50 cycles - Throughput is priority

Disable prefetch when: - Area is constrained - Single descriptors only - Simplicity is priority

29.9.48.4 Burst Size Selection

Read Burst Size:

```
Optimal burst size = min(  
    Memory controller page size,  
    4KB (AXI limit),  
    SRAM FIFO depth / 2  
)
```

Example for DDR4 (8KB page), FIFO depth 512 entries:

- Page size: 8192 bytes = 128 beats (512-bit)
- AXI limit: 4096 bytes = 64 beats
- FIFO limit: 512/2 = 256 beats
- Optimal: min(128, 64, 256) = 64 beats

Write Burst Size: - Usually same as read burst size - May be smaller if write FIFO depth is limited

29.10 9. Configuration Register Map Summary

9. Configuration Register Map Summary

Address	Register Name	Fields	Section
0x100	GLOBAL_CTRL	global_en, global_RST	-
0x120	CHANNEL_ENABLE	ch_en[7:0]	1
0x124	CHANNEL_RESET	ch_RST[7:0]	1
0x200	SCHED_TIMEOUT_CYCLES	timeout_cycle s[15:0]	2
0x204	SCHED_CONFIG	enable, timeout_en, err_en, compl_en, perf_en	2
0x220	DESCENG_CONFIG	enable, prefetch, fifo_thresh	3
0x224	DESCENG_ADDR0_BASE	addr0_base[3 1:0]	3
0x228	DESCENG_ADDR0_LIMIT	addr0_limit[3 1:0]	3

Address	Register Name	Fields	Section
0x22C	DESCENG_ADDR1_BASE	addr1_base[3:0]	3
0x230	DESCENG_ADDR1_LIMIT	addr1_limit[3:0]	3
0x240-0x25F	DAXMON_*	Descriptor monitor (15 signals)	4.1
0x260-0x27F	RDMON_*	Read engine monitor (15 signals)	4.2
0x280-0x29F	WRMON_*	Write engine monitor (15 signals)	4.3
0x2A0	AXI_XFER_CFG	rd_xfer_beats ,	5
		wr_xfer_beat s	
0x2B0	PERF_CTRL	perf_enable, perf_mode, perf_clear	6

Total Address Space: 0x000-0x3FF (1KB)

29.11 10. Software Configuration Examples

29.11.1 10.1 C/C++ Initialization (Minimal)

```
// Minimal configuration for single-channel operation
void stream_init_minimal(volatile uint32_t *base_addr) {
    // Global enable
    base_addr[0x100/4] = 0x00000001; // GLOBAL_CTRL.global_en

    // Enable channel 0 only
    base_addr[0x120/4] = 0x00000001; // CHANNEL_ENABLE.ch_en

    // Scheduler config
```

```

    base_addr[0x200/4] = 500;           // SCHED_TIMEOUT_CYCLES (SRAM)
    base_addr[0x204/4] = 0x00000007;   // SCHED_CONFIG: enable |
timeout_en | err_en

    // Descriptor engine
    base_addr[0x220/4] = 0x00000041; // enable | fifo_thresh=4

    // Disable all monitors (minimal)
    base_addr[0x240/4] = 0x00000000; // DAXMON_ENABLE
    base_addr[0x260/4] = 0x00000000; // RDMON_ENABLE
    base_addr[0x280/4] = 0x00000000; // WRMON_ENABLE

    // AXI transfer config
    base_addr[0x2A0/4] = 0x00000808; // 8 beats read + write
}


```

29.11.2 10.2 C/C++ Initialization (Balanced)

```

// Balanced configuration for typical FPGA
void stream_init_balanced(volatile uint32_t *base_addr) {
    // Global enable
    base_addr[0x100/4] = 0x00000001;

    // Enable all 8 channels
    base_addr[0x120/4] = 0x000000FF;

    // Scheduler config
    base_addr[0x200/4] = 5000;           // SCHED_TIMEOUT_CYCLES (DDR4)
    base_addr[0x204/4] = 0x00000007;   // SCHED_CONFIG: enable |
timeout_en | err_en

    // Descriptor engine
    base_addr[0x220/4] = 0x00000083; // enable | prefetch |
fifo_thresh=8

    // Descriptor AXI monitor (errors only)
    base_addr[0x240/4] = 0x0000000B; // enable | err_en | timeout_en
    base_addr[0x244/4] = 10000;        // timeout_cycles
    base_addr[0x24C/4] = 0x00000060; // pkt_mask: errors + timeouts

    // Read/write monitors (errors only)
    base_addr[0x260/4] = 0x0000000B; // RDMON: enable | err_en |
timeout_en
    base_addr[0x280/4] = 0x0000000B; // WRMON: enable | err_en |
timeout_en

```

```
// AXI transfer config  
base_addr[0x2A0/4] = 0x00002020; // 32 beats read + write  
  
// Enable performance profiler  
base_addr[0x2B0/4] = 0x00000003; // enable | elapsed mode  
}
```

29.12 11. Troubleshooting Configuration Issues

29.12.1 Problem: No transfers occurring

Check: 1. cfg_channel_enable[n] set for channel n? 2. cfg_sched_enable = 1? 3. cfg_desceng_enable = 1? 4. Global enable set?

29.12.2 Problem: Timeout errors

Check: 1. cfg_sched_timeout_cycles too small? 2. Memory latency higher than expected? 3. AXI backpressure not handled?

Solution: - Increase timeout: cfg_sched_timeout_cycles = 20000 - Disable temporarily: cfg_sched_timeout_enable = 0

29.12.3 Problem: MonBus overflow

Check: 1. Too many completion packets enabled? 2. Performance packets enabled on all channels? 3. Debug packets enabled?

Solution: - Disable completion: cfg_*_mon_compl_enable = 0 - Use channel masks: cfg_*_mon_compl_mask = 8'h01 (channel 0 only) - Reduce packet types: cfg_*_mon_pkt_mask = 16'h0060 (errors + timeouts)

29.12.4 Problem: Low throughput

Check: 1. Prefetch disabled? 2. Burst size too small? 3. FIFO threshold too conservative?

Solution: - Enable prefetch: cfg_desceng_prefetch = 1 - Increase bursts: cfg_axi_rd_xfer_beats = 64 - Increase FIFO: cfg_desceng_fifo_thresh = 12

29.13 Related Documentation

- [Register Map](#) - Complete APB register specification
 - [Clocks and Reset](#) - Timing requirements
 - [Programming Guide](#) - Software API examples
-

Last Updated: 2025-12-01 Maintained By: STREAM Architecture Team

30 Product Requirements Document (PRD)

30.1 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

30.2 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.

30.2.1 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

30.2.2 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)
-

30.3 2. Key Design Principles

30.3.12.1 Simplifications from RAPIDS

STREAM is intentionally simplified for tutorial purposes:

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

30.3.22.2 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)
-

30.4 3. Architecture Overview

30.4.13.1 Top-Level Block Diagram

STREAM (Scatter-gather Transfer Rapid Engine for AXI Memory)

- APB Config Slave
 - Channel registers (8 channels, kick-off via write)
- Descriptor Fetch
 - AXI Master (Read) - Fetch descriptors from memory
- Scheduler Group (shared across all channels)
 - Descriptor Engine (FIFO, parsing - FROM RAPIDS)
 - Scheduler (Simplified FSM coordination)
 - Channel Arbitration (8 independent channels)
- Data Path (shared resources)
 - AXI Master (Read) - Parameterized data width
 - Simple SRAM - Dual-port buffer (FROM RAPIDS)
 - AXI Master (Write) - Parameterized data width
- MonBus Reporter
 - MonBus Master - 64-bit monitoring packets

30.4.23.2 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

30.5 4. Interfaces

30.5.14.1 External Interfaces

Interface	Type	Width	Purpose	Notes
APB Slave	Slave	32-bit	Configuration, channel kick-off	Write to channel register starts transfer
AXI Master (Descriptor)	Master	256-bit	Fetch descriptors from memory	Dedicated descriptor fetch path
AXI Master (Data Read)	Master	Parameterizable	Read source data	Multiple engine versions (compile-time)
AXI Master (Data Write)	Master	Parameterizable	Write destination data	Multiple engine versions (compile-time)
MonBus Master	Master	64-bit	Monitor packet output	Standard AMBA format

30.5.24.2 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 BEATS (not bytes!)
[191:160]	next_descriptor_p	Address of next descriptor (0 = last in

Bits	Field	Description
	tr	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)

30.6 5. Key Components

30.6.15.1 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

30.6.25.2 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	Gates APB interface

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!

30.6.35.3 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)

30.6.45.3 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
           // Channel has read request
          // Engine ready for request
     // Source address (aligned)
 // Beats left to read
    // Preferred burst length
   // Channel ID for tracking

```

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)

```

30.6.5.4 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

```

30.6.6 5.5 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

30.7 6. Configuration and Control

30.7.1 6.1 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	Channel 0 kick-off (write descriptor address)
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

30.7.26.2 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

30.8 7. Resource Sharing and Arbitration

30.8.17.1 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

30.8.27.2 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 ...

30.9 8. Error Detection and Recovery

30.9.18.1 Error Types

Error Type	Detection	Response
Invalid descriptor	Valid bit = 0	Skip, move to next
Alignment error	Address not aligned	Set error flag, halt channel
AXI SLVERR	AXI response	Set error flag, halt channel
AXI DECERR	AXI response	Set error flag, halt channel

Error Type	Detection	Response
Timeout	Transaction timeout	Set error flag, halt channel
SRAM overflow	Buffer full	Backpressure, wait

30.9.28.2 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

30.10 9. MonBus Integration

30.10.1 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.)

30.10.2 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

30.10.3 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

30.11 10. Design Constraints

30.11.1 10.1 Tutorial Constraints (Intentional Simplifications)

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

30.11.2 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

30.12 11. Verification Strategy

30.12.1 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

```

30.12.2 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

30.13 12. Performance Characteristics

30.13.1 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)

30.13.2 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

30.13.3 12.3 Resource Utilization by Engine Version

V1 Configuration (Tutorial - Minimum Area): - Total: ~9,500 LUTs + 64 KB SRAM - Descriptor Engine (8x): ~2,400 LUTs - Scheduler (8x): ~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.

30.14 13. Development Roadmap

30.14.1 13.1 Phase 1: Foundation (Current)

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

30.14.2 13.2 Phase 2: Core Blocks

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

30.14.3 13.3 Phase 3: Data Path

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

30.14.4 13.4 Phase 4: Integration

- Top-level module
- MonBus reporter
- Integration testbench
- Single-channel validation

30.14.5 13.5 Phase 5: Multi-Channel

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

30.14.6 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

30.15 14. Educational Value

30.15.1 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

30.15.2 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

30.16 15. Success Criteria

30.16.1 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

30.16.2 15.2 Quality

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

30.16.3 15.3 Performance

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

30.17 16. Open Questions (For Review)

30.17.1 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

30.17.2 16.2 AXI Descriptor Master

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

30.17.3 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

30.17.4 16.4 SRAM Partitioning

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

30.18 16. Attribution and Contribution Guidelines

30.18.1 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.

30.19 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

30.20 17. References

30.20.1 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

30.20.2 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

30.21 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)

31 Claude Code Guide: STREAM Subsystem

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

31.1 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

31.2 📖 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

31.3 Critical Rules for This Subsystem

31.3.1 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.

31.3.2 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!)

31.3.3 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/
    ┌── area!
      └── scheduler_tb.py          # ★ STREAM TB classes (project
      └── descriptor_engine_tb.py # Scheduler testbench
      └── axi_engine_tb.py        # Descriptor engine testbench
    └── tests/fub_tests/
      └── tbclasses/              # AXI engine testbenches
        └── tests/                # Test runners import from
```

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

31.3.4 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(TBBBase):
    @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.

31.3.5 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?”

31.3.6 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

31.3.7 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)

31.4 Architecture Quick Reference

31.4.1 Block Organization

STREAM Architecture (Estimated: 8-10 modules)

```

├── APB Config
    └── apb_config_slave.sv      (To be created)

```

```

    └── Scheduler Group
        ├── descriptor_engine.sv      (FROM RAPIDS - adapt imports)
        ├── scheduler.sv              (Simplified from RAPIDS)
        └── channel_arbiter.sv        (To be created)

    └── Data Path
        ├── axi_read_engine_v1.sv     (Version 1: Basic)
        ├── axi_read_engine_v2.sv     (Version 2: Pipelined)
        ├── axi_write_engine_v1.sv    (Version 1: Basic)
        ├── axi_write_engine_v2.sv    (Version 2: Pipelined)
        └── simple_sram.sv           (FROM RAPIDS - no changes)

    └── MonBus Reporter
        └── monbus_reporter.sv       (To be created)

```

31.4.2 Module Status

Module	Source	Status	Notes
descriptor_engin e.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.s vh	New	✓ Created	Package imports
scheduler.sv	RAPIDS (simplify)	⌚ Pending	Remove credit 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

Module	Source	Status	Notes
			integration

31.5 Common User Questions and Responses

31.5.1 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

31.5.2 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

31.5.3 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

31.5.4 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 × 64 = 1024 bytes
```

See: PRD.md Section 4.2 for complete descriptor specification

31.5.5 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

31.6 Integration Patterns

31.6.1 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_app_pwrite      (apb_pwrite),
.s_app_pwdata     (apb_pwdata),
.s_app_pready     (apb_pready),
.s_app_prdata     (apb_prdata),
.s_app_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)
);

```

31.6.2 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 x 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 x 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]);

```

31.6.3 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)
);

```

31.7 Anti-Patterns to Catch

31.7.1 X Anti-Pattern 1: Adding Alignment Fixup

x 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."

31.7.2 X Anti-Pattern 2: Using Length in Bytes

x 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
```

31.7.3 X Anti-Pattern 3: Circular Buffer Descriptors

x 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
```

31.7.4 X Anti-Pattern 4: Assuming Exclusive Channel Access

x 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)
);
```

31.8 Debugging Workflow

31.8.1 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
```

31.8.2 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

31.8.3 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?

31.9 Testing Guidance

31.9.1 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
```

31.9.2 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

31.10 Key Documentation Links

31.10.1 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

31.11 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)

```
verilator --lint-only
```

```
projects/components/stream/rtl/stream_fub/scheduler.sv
```

Search for modules

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

31.12 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/
-

31.13 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

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

32 STREAM Register Definitions

Purpose: PeakRDL register files and generated RTL for STREAM configuration

32.1 Directory Structure

```
regs/  
└── README.md                      # This file  
└── stream_regs.rdl                # Register definition (to be created)  
  └── generated/                   # PeakRDL-generated outputs (to be  
    └── created)  
      └── rtl/                     # Generated RTL files  
        └── stream_regs.sv
```

```
└── stream_regs_pkg.sv  
docs/  
└── stream_regs.html
```

32.2 Register Generation Workflow

32.2.1 Phase 1: Define Register Map (Future)

Create `stream_regs.rdl` following the same pattern as `apb_hpet`.

32.2.2 Phase 2: PeakRDL-Generated Registers (Future)

Following the same pattern as `projects/components/apb_hpet/`:

1. Define Register Map (`stream_regs.rdl`)

```
addrmap stream_regs {  
    name = "STREAM Configuration Registers";  
  
    // Global control  
    reg {  
        field { sw=rw; hw=r; } enable;  
        field { sw=rw; hw=r; } reset;  
    } GLOBAL_CTRL @ 0x00;  
  
    // Channel registers (8 channels × 16 bytes)  
    regfile channel_regs {  
        reg { ... } CH_CTRL;  
        reg { ... } CH_STATUS;  
        reg { ... } CH_RD_BURST;  
        reg { ... } CH_WR_BURST;  
    };  
  
    channel_regs CH[8] @ 0x10 += 0x10;  
};
```

2. Generate RTL

```
cd projects/components/stream/regs  
../../bin/peakrdl_generate.py stream_regs.rdl --copy-rtl  
./rtl/stream_macro
```

3. Create APB Config Wrapper

- Instantiate generated register block (similar to `apb_hpet.sv`)

- Add apb_slave_cdc wrapper if clock domain crossing needed
 - Wire register outputs to STREAM control signals
-

32.3 Register Map (Planned)

32.3.1 Global Registers

Address	Name	Access	Description
0x00	GLOBAL_CTRL	RW	Global enable, reset
0x04	GLOBAL_STAT	RO US	Global status, channel idle/error
0x08	GLOBAL_CONF	RW IG	Global configuration
0x0C	Reserved	-	-

32.3.2 Channel Registers (8 × 0x10 bytes)

Each channel (0-7) has a 16-byte register block starting at $0x10 + (\text{channel_id} \times 0x10)$:

Offset	Name	Access	Description
+0x00	CHx_CTRL	WO	Descriptor address (write to kick off)
+0x04	CHx_STATUS	RO	Channel status, idle, error
+0x08	CHx_RD_BURS	RW T	Read burst length (for engine config)
+0x0C	CHx_WR_BURS	RW T	Write burst length (for engine config)

Example: - Channel 0 registers: 0x10, 0x14, 0x18, 0x1C - Channel 1 registers: 0x20, 0x24, 0x28, 0x2C - Channel 7 registers: 0x80, 0x84, 0x88, 0x8C

32.4 Integration Pattern

32.4.1 PeakRDL-Generated Implementation (Future):

```
// rtl/stream_macro/apb_config.sv
module apb_config (
    // APB interface
    input logic [ADDR_WIDTH-1:0] paddr,
    // ...
);
    // Instantiate PeakRDL-generated registers
    stream_regs u_regs (
        .pclk      (pclk),
        .presetn   (presetn),
        .paddr     (paddr),
        .psel      (psel),
        // ... APB signals

        // Register field outputs
        .global_ctrl_enable (ch_enable_internal),
        .ch0_ctrl_desc_addr (ch_desc_addr[0]),
        .ch0_rd_burst       (ch_read_burst_len[0]),
        // ... generated field outputs
    );
    // Optional: Add CDC wrapper if crossing clock domains
    // (like HPET's apb_slave_cdc)
endmodule
```

32.5 Reference Examples

HPET PeakRDL Implementation: -

projects/components/apb_hpet/peakrdl/hpet_regs.rdl - Register definition -
projects/components/apb_hpet/peakrdl/generated/ - Generated outputs -
projects/components/apb_hpet/rtl/apb_hpet.sv - Wrapper with CDC

HPET Generation Command:

```
.../bin/peakrdl_generate.py hpet_regs.rdl --copy-rtl ..
```

32.6 Status

- ⌚ Phase 1: Manual apb_config.sv (placeholder)

-  **Phase 2:** Create `stream_regs.rdl` (deferred to after Phase 2 RTL implementation)
-  **Phase 3:** Generate register RTL with PeakRDL
-  **Phase 4:** Update `apb_config.sv` wrapper to use generated registers

Note: Per user feedback, “The apb config will be the last thing done. They will have configs done along the lines of how they are done for the hpet.”

Last Updated: 2025-10-17 **Related Documentation:** - ./docs/ARCHITECTURAL_NOTES.md
Section 6 - APB Configuration (Deferred) - ./PRD.md Section 3.1 - APB Configuration Interface -
./././apb_hpet/peakrdl/README.md - HPET PeakRDL example