

# RTL Design Sherpa

Converters Micro-Architecture Specification 1.0

**January 4, 2026**

## Table of Contents

1 Converter Mas Index.....	22
2 Document Information.....	22
2.1 Document Control.....	22
2.2 Purpose.....	22
2.3 Audience.....	22
2.4 Related Documents.....	23
2.5 Revision History.....	23
2.6 Conventions.....	23
2.6.1 Notation.....	23
2.6.2 Diagrams.....	23
2.6.3 Code Examples.....	23
3 Chapter 1: Introduction - Overview.....	24
3.1 1.1 Purpose.....	24
3.2 1.2 Problem Statement.....	24
3.2.1 1.2.1 Data Width Mismatch.....	24
3.2.2 1.2.2 Protocol Incompatibility.....	24
3.3 1.3 Solution Architecture.....	25
3.3.1 Figure 1.1: Converter Module Hierarchy.....	25
3.3.2 1.3.1 Data Width Converters.....	25
3.3.3 1.3.2 Protocol Converters.....	25
3.4 1.4 Key Design Decisions.....	26
3.4.1 1.4.1 Generic vs. Full Modules.....	26

3.4.2 1.4.2 Throughput vs. Area Trade-offs.....	26
3.4.3 1.4.3 Sideband Signal Handling.....	26
3.5 1.5 Performance Characteristics.....	27
3.5.1 1.5.1 Latency.....	27
3.5.2 1.5.2 Throughput.....	27
3.6 1.6 Scope.....	27
3.6.1 In Scope.....	27
3.6.2 Out of Scope.....	28
3.7 1.7 Target Applications.....	28
4 2.1 Generic Building Blocks.....	28
4.1 2.1.1 Module Hierarchy.....	28
4.2 2.1.2 Design Philosophy.....	29
4.2.1 Separation of Concerns.....	29
4.2.2 Interface Pattern.....	29
4.3 2.1.3 Width Ratio Calculation.....	29
4.4 2.1.4 Sideband Modes.....	30
4.4.1 Upsize Sideband Modes.....	30
4.4.2 Downsize Sideband Modes.....	30
4.5 2.1.5 Performance Comparison.....	31
4.5.1 Figure 2.1: Performance Comparison.....	31
4.6 2.1.6 Resource Utilization.....	31
4.6.1 Upsize Resources (NARROW=64, WIDE=512).....	31
4.6.2 Downsize Resources (WIDE=512, NARROW=64).....	31
4.7 2.1.7 Integration Guidelines.....	32

4.7.1 When to Use Generic Blocks.....	32
4.7.2 Composition Example.....	32
5 2.2 axi_data_upsize Module.....	32
5.1 2.2.1 Purpose and Function.....	33
5.2 2.2.2 Block Diagram.....	33
5.2.1 Figure 2.2: axi_data_upsize Architecture.....	33
5.3 2.2.3 Interface Specification.....	34
5.3.1 Parameters.....	34
5.3.2 Ports.....	34
5.4 2.2.4 Operation.....	35
5.4.1 Accumulation Cycle.....	35
5.4.2 Early LAST Handling.....	35
5.4.3 State Machine.....	35
5.5 2.2.5 Sideband Handling.....	35
5.5.1 Concatenate Mode (SB_OR_MODE=0).....	35
5.5.2 OR Mode (SB_OR_MODE=1).....	36
5.6 2.2.6 Implementation.....	36
5.6.1 Core Logic.....	36
5.7 2.2.7 Timing Characteristics.....	38
5.7.1 Latency.....	38
5.7.2 Throughput.....	38
5.7.3 Critical Paths.....	38
5.8 2.2.8 Resource Utilization.....	38
5.8.1 Typical Resources (64-bit to 512-bit).....	38

5.8.2 Scaling.....	38
5.9 2.2.9 Usage Example.....	39
5.9.1 64-bit to 512-bit Write Data.....	39
6 2.3 axi_data_dnsiz Module.....	39
6.1 2.3.1 Purpose and Function.....	39
6.2 2.3.2 Block Diagram.....	40
6.2.1 Figure 2.3: axi_data_dnsiz Single-Buffer Architecture.....	40
6.2.2 Figure 2.4: axi_data_dnsiz Dual-Buffer Architecture.....	40
6.3 2.3.3 Interface Specification.....	41
6.3.1 Parameters.....	41
6.3.2 Ports.....	41
6.4 2.3.4 Single-Buffer Mode Operation.....	42
6.4.1 Split Cycle.....	42
6.4.2 State Machine.....	42
6.4.3 Throughput Analysis.....	42
6.5 2.3.5 Dual-Buffer Mode Operation.....	43
6.5.1 Ping-Pong Operation.....	43
6.5.2 State Machine (Dual).....	43
6.5.3 100% Throughput.....	43
6.6 2.3.6 Sideband Handling.....	43
6.6.1 Slice Mode (SB_BROADCAST=0).....	43
6.6.2 Broadcast Mode (SB_BROADCAST=1).....	44
6.7 2.3.7 Burst Tracking.....	44
6.7.1 Purpose.....	44

6.7.2 Operation.....	44
6.8 2.3.8 Implementation.....	45
6.8.1 Single-Buffer Core Logic.....	45
6.9 2.3.9 Resource Utilization.....	46
6.9.1 Single-Buffer (512-bit to 64-bit).....	46
6.9.2 Dual-Buffer (512-bit to 64-bit).....	46
6.9.3 Comparison.....	46
6.10 2.3.10 Usage Example.....	47
6.10.1 512-bit to 64-bit Read Data (High Performance).....	47
7 2.4 Dual-Buffer Architecture.....	47
7.1 2.4.1 Problem Statement.....	47
7.1.1 Single-Buffer Limitation.....	47
7.1.2 High-Performance Requirements.....	48
7.2 2.4.2 Solution: Ping-Pong Buffering.....	48
7.2.1 Concept.....	48
7.2.2 Figure 2.5: Dual-Buffer Ping-Pong Operation.....	48
7.3 2.4.3 Implementation.....	48
7.3.1 Buffer State Machine.....	48
7.3.2 Output Arbiter.....	49
7.3.3 Load Controller.....	49
7.3.4 Buffer State Transitions.....	49
7.4 2.4.4 Timing Analysis.....	50
7.4.1 Continuous Operation.....	50
7.4.2 Latency.....	50

7.5 2.4.5 Edge Cases.....	51
7.5.1 Empty Pipeline Start.....	51
7.5.2 Pipeline Drain.....	51
7.5.3 Backpressure Handling.....	51
7.6 2.4.6 Resource Comparison.....	51
7.6.1 Resource Breakdown.....	51
7.6.2 Trade-off Summary.....	51
7.7 2.4.7 Configuration Guidelines.....	52
7.7.1 When to Use Dual Buffer.....	52
7.7.2 Integration Example.....	52
8 2.5 axi4_dwidth_converter_wr.....	52
8.1 2.5.1 Purpose and Function.....	52
8.2 2.5.2 Block Diagram.....	53
8.2.1 Figure 2.6: Write Converter Architecture.....	53
8.3 2.5.3 Interface Specification.....	54
8.3.1 Parameters.....	54
8.3.2 Ports.....	54
8.4 2.5.4 Burst Length Conversion.....	55
8.4.1 Ratio Calculation.....	55
8.4.2 Examples.....	55
8.4.3 Non-Aligned Bursts.....	56
8.5 2.5.5 Address Channel Handling.....	56
8.5.1 AW Passthrough with Adjustment.....	56
8.5.2 Skid Buffer for AW.....	56

8.6 2.5.6 Write Data Channel.....	57
8.6.1 Upsize Instance.....	57
8.7 2.5.7 Response Channel.....	57
8.7.1 B Channel Passthrough.....	57
8.7.2 Response Ordering.....	57
8.8 2.5.8 AW/W Synchronization.....	57
8.8.1 Challenge.....	57
8.8.2 Solution.....	58
8.9 2.5.9 Resource Utilization.....	58
8.9.1 Typical Resources (64→512, ID=4, ADDR=64).....	58
8.10 2.5.10 Timing Characteristics.....	58
8.10.1 Latency.....	58
8.10.2 Throughput.....	58
8.11 2.5.11 Usage Example.....	59
9 2.6 axi4_dwidth_converter_rd.....	59
9.1 2.6.1 Purpose and Function.....	59
9.2 2.6.2 Block Diagram.....	60
9.2.1 Figure 2.7: Read Converter Architecture.....	60
9.3 2.6.3 Interface Specification.....	60
9.3.1 Parameters.....	60
9.3.2 Ports.....	61
9.4 2.6.4 Burst Length Conversion.....	62
9.4.1 Ratio Calculation.....	62
9.4.2 Examples.....	62



9.5 2.6.5 Address Channel Handling.....	62
9.5.1 AR Passthrough with Adjustment.....	62
9.5.2 AR Information FIFO.....	62
9.6 2.6.6 Read Data Channel.....	63
9.6.1 Downsize Instance.....	63
9.7 2.6.7 RLAST Generation.....	63
9.7.1 Challenge.....	63
9.7.2 Solution: Burst Tracker.....	64
9.8 2.6.8 RID Handling.....	64
9.8.1 ID Passthrough.....	64
9.9 2.6.9 Dual-Buffer Impact.....	64
9.9.1 Performance Comparison.....	64
9.9.2 When to Use Dual Buffer.....	65
9.10 2.6.10 Resource Utilization.....	65
9.10.1 Typical Resources (512→64, ID=4, Dual Buffer).....	65
9.10.2 Single Buffer Version.....	65
9.11 2.6.11 Timing Characteristics.....	65
9.11.1 Latency.....	65
9.11.2 Throughput.....	65
9.12 2.6.12 Usage Example.....	65
10 3.1 Protocol Conversion Overview.....	66
10.1 3.1.1 Available Converters.....	66
10.1.1 AXI4 to AXI4-Lite (Protocol Downgrade).....	66
10.1.2 AXI4-Lite to AXI4 (Protocol Upgrade).....	67

10.1.3 Other Protocol Converters.....	67
10.2 3.1.2 Protocol Comparison.....	67
10.2.1 Figure 3.1: Protocol Feature Comparison.....	67
10.2.2 Feature Matrix.....	67
10.3 3.1.3 Conversion Strategies.....	68
10.3.1 AXI4 to AXI4-Lite.....	68
10.3.2 AXI4-Lite to AXI4.....	68
10.3.3 AXI4 to APB.....	68
10.4 3.1.4 Performance Characteristics.....	68
10.4.1 Key Observations.....	69
10.5 3.1.5 Use Case Guidelines.....	69
10.5.1 When to Use AXI4 to AXI4-Lite.....	69
10.5.2 When to Use AXI4-Lite to AXI4.....	69
10.5.3 When to Use AXI4 to APB.....	69
10.6 3.1.6 Integration Patterns.....	69
10.6.1 Pattern 1: CPU to Peripherals.....	69
10.6.2 Pattern 2: Simple IP in AXI4 Fabric.....	69
10.6.3 Pattern 3: Mixed System.....	70
11 3.2 AXI4 to AXI4-Lite Converter.....	70
11.1 3.2.1 Module Organization.....	70
11.1.1 Design Philosophy.....	70
11.2 3.2.2 Read Path (axi4_to_axil4_rd).....	70
11.2.1 Block Diagram.....	70
11.2.2 Figure 3.2: AXI4 to AXI4-Lite Read Path.....	70

11.2.3 Operation.....	70
11.2.4 State Machine.....	71
11.2.5 Implementation.....	71
11.3 3.2.3 Write Path (axi4_to_axil4_wr).....	72
11.3.1 Block Diagram.....	72
11.3.2 Figure 3.3: AXI4 to AXI4-Lite Write Path.....	72
11.3.3 Operation.....	72
11.3.4 AW/W Synchronization Challenge.....	72
11.3.5 Solution: Dual Accept Logic.....	73
11.3.6 Response Aggregation.....	73
11.4 3.2.4 Bidirectional Wrapper (axi4_to_axil4).....	74
11.4.1 Composition Pattern.....	74
11.5 3.2.5 Resource Utilization.....	74
11.6 3.2.6 Performance Analysis.....	75
11.6.1 Throughput.....	75
11.6.2 Latency.....	75
11.7 3.2.7 Test Coverage.....	75
12 3.3 AXI4-Lite to AXI4 Converter.....	76
12.1 3.3.1 Module Organization.....	76
12.2 3.3.2 Design Philosophy.....	76
12.3 3.3.3 Signal Mapping.....	76
12.3.1 Address Channel Signals.....	76
12.3.2 Data Channel Signals.....	77
12.4 3.3.4 Read Path (axil4_to_axi4_rd).....	77

12.4.1 Block Diagram.....	77
12.4.2 Figure 3.4: AXI4-Lite to AXI4 Read Path.....	77
12.4.3 Implementation.....	77
12.5 3.3.5 Write Path (axil4_to_axi4_wr).....	79
12.5.1 Block Diagram.....	79
12.5.2 Figure 3.5: AXI4-Lite to AXI4 Write Path.....	79
12.5.3 Implementation.....	79
12.6 3.3.6 Bidirectional Wrapper.....	80
12.7 3.3.7 Resource Utilization.....	81
12.8 3.3.8 Performance.....	81
12.9 3.3.9 Test Coverage.....	81
12.10 3.3.10 Usage Example.....	82
13 3.4 AXI4 to APB Converter.....	82
13.1 3.4.1 Purpose.....	83
13.2 3.4.2 Block Diagram.....	84
13.2.1 Figure 3.6: AXI4 to APB Converter.....	84
13.3 3.4.3 Interface Specification.....	85
13.3.1 Parameters.....	85
13.3.2 Ports.....	85
13.4 3.4.4 State Machine.....	87
13.4.1 Figure 3.7: AXI4 to APB FSM.....	87
13.4.2 States.....	88
13.4.3 Transitions.....	88
13.5 3.4.5 Burst Handling.....	88

13.5.1 Burst Decomposition.....	88
13.5.2 Address Calculation.....	89
13.6 3.4.6 Address Width Adaptation.....	89
13.6.1 64-bit to 32-bit Conversion.....	89
13.7 3.4.7 Error Response Mapping.....	89
13.7.1 Error Aggregation.....	89
13.8 3.4.8 Implementation.....	90
13.8.1 Core FSM.....	90
13.9 3.4.9 Resource Utilization.....	91
13.10 3.4.10 Performance.....	91
13.10.1 Timing Analysis.....	91
13.10.2 Throughput.....	92
13.11 3.4.11 Usage Example.....	92
14 3.5 PeakRDL Adapter.....	92
14.1 3.5.1 Purpose.....	92
14.2 3.5.2 Block Diagram.....	94
14.2.1 Figure 3.8: PeakRDL Adapter.....	94
14.3 3.5.3 Interface Specification.....	96
14.3.1 Parameters.....	96
14.3.2 Ports.....	96
14.4 3.5.4 Operation.....	97
14.4.1 Write Transaction.....	97
14.4.2 Read Transaction.....	97
14.5 3.5.5 Implementation.....	97

14.6 3.5.6 Resource Utilization.....	98
14.7 3.5.7 Use Cases.....	98
14.7.1 1. PeakRDL to Custom Control.....	98
14.7.2 2. Register Access Logging.....	99
14.7.3 3. Pipeline Insertion.....	99
14.8 3.5.8 Integration Example.....	99
15 4.1 Width Converter FSMs.....	100
15.1 4.1.1 Upsize FSM.....	100
15.1.1 Figure 4.1: Upsize FSM.....	101
15.1.2 States.....	101
15.1.3 Transitions.....	101
15.1.4 Implementation.....	102
15.2 4.1.2 Downsize FSM (Single Buffer).....	102
15.2.1 Figure 4.2: Downsize Single-Buffer FSM.....	103
15.2.2 States.....	105
15.2.3 Transitions.....	105
15.3 4.1.3 Downsize FSM (Dual Buffer).....	105
15.3.1 Buffer State Machine.....	105
15.3.2 Arbiter Logic.....	105
15.4 4.1.4 Full Converter FSMs.....	106
15.4.1 Write Converter (axi4_dwidth_converter_wr).....	106
15.4.2 Read Converter (axi4_dwidth_converter_rd).....	106
15.5 4.1.5 Timing Diagrams.....	107
15.5.1 Upsize Timing (8:1 ratio).....	107

15.5.2 Downsize Timing (8:1 ratio, single buffer).....	107
16 4.2 Protocol Converter FSMs.....	107
16.1 4.2.1 AXI4 to AXI4-Lite Read FSM.....	107
16.1.1 Figure 4.3: AXI4 to AXIL4 Read FSM.....	107
16.1.2 States.....	107
16.1.3 Transitions.....	108
16.1.4 Implementation.....	108
16.2 4.2.2 AXI4 to AXI4-Lite Write FSM.....	109
16.2.1 States.....	109
16.2.2 AW/W Synchronization.....	109
16.3 4.2.3 AXI4 to APB FSM.....	111
16.3.1 Figure 4.4: AXI4 to APB FSM.....	111
16.3.2 States.....	112
16.3.3 APB Protocol Phases.....	112
16.3.4 Implementation.....	112
16.4 4.2.4 Timing Analysis.....	113
16.4.1 AXI4 to AXIL4 Timing.....	113
16.4.2 AXI4 to APB Timing.....	114
17 4.3 Burst Decomposition.....	114
17.1 4.3.1 Width Converter Burst Handling.....	114
17.1.1 Burst Length Adjustment.....	114
17.1.2 Figure 4.5: Width Burst Conversion.....	114
17.1.3 Non-Aligned Bursts.....	114
17.2 4.3.2 Protocol Converter Burst Handling.....	115

17.2.1 AXI4 to AXI4-Lite Decomposition.....	115
17.2.2 Address Increment Calculation.....	115
17.3 4.3.3 Response Aggregation.....	116
17.3.1 Read Response Aggregation.....	116
17.3.2 Write Response Aggregation.....	116
17.4 4.3.4 Burst Tracking Registers.....	117
17.4.1 Required State.....	117
17.4.2 Initialization.....	117
17.5 4.3.5 Timing Impact.....	117
17.5.1 Decomposition Overhead.....	117
17.5.2 Pipeline Considerations.....	118
18 5.1 Test Strategy.....	118
18.1 5.1.1 Test Organization.....	118
18.1.1 Test Hierarchy.....	118
18.2 5.1.2 Test Levels.....	118
18.2.1 Level 1: Unit Tests.....	118
18.2.2 Level 2: Integration Tests.....	119
18.2.3 Level 3: System Tests.....	119
18.3 5.1.3 Test Categories.....	119
18.3.1 Functional Tests.....	119
18.3.2 Stress Tests.....	120
18.3.3 Error Tests.....	120
18.4 5.1.4 Testbench Architecture.....	120
18.4.1 Components.....	120



18.4.2 Driver Implementation.....	121
18.5 5.1.5 Coverage Model.....	121
18.5.1 Functional Coverage.....	121
18.5.2 Code Coverage.....	122
18.6 5.1.6 Test Execution.....	122
18.6.1 Running Tests.....	122
18.6.2 CI/CD Integration.....	122
19 5.2 Debug Guide.....	123
19.1 5.2.1 Common Issues.....	123
19.1.1 Width Converter Issues.....	123
19.1.2 Protocol Converter Issues.....	124
19.2 5.2.2 Debug Signals.....	124
19.2.1 Recommended Internal Signals.....	124
19.2.2 ILA Configuration.....	125
19.3 5.2.3 Simulation Debug.....	125
19.3.1 Waveform Analysis.....	125
19.3.2 Timing Diagram Template.....	125
19.4 5.2.4 Common Mistakes.....	126
19.4.1 Mistake 1: Wrong Width Ratio.....	126
19.4.2 Mistake 2: Missing Sideband Handling.....	126
19.4.3 Mistake 3: Incorrect LAST Timing.....	126
19.4.4 Mistake 4: Burst Length Calculation Error.....	126
19.5 5.2.5 Verification Checklist.....	127
19.6 5.2.6 Performance Validation.....	127

19.6.1 Throughput Measurement.....	127
19.6.2 Expected Throughput.....	127
20 Converters Component Specification.....	128
20.1 Document Overview.....	128
20.2 Quick Navigation.....	128
20.2.1 Chapter 1: Overview.....	128
20.2.2 Chapter 2: Data Width Converters.....	128
20.2.3 Chapter 3: Protocol Converters.....	129
20.2.4 Chapter 4: Usage Examples.....	129
20.2.5 Appendices.....	129
20.3 Component Summary.....	129
20.3.1 Data Width Converters.....	129
20.3.2 Protocol Converters.....	131
20.4 Performance Summary.....	133
20.5 Key Features.....	134
20.5.1 Data Width Conversion.....	134
20.5.2 Protocol Conversion.....	134
20.6 Documentation Assets.....	134
20.6.1 Graphviz Block Diagrams.....	134
20.6.2 PlantUML FSM Diagrams.....	134
20.7 Related Documentation.....	135
20.8 Version History.....	135
20.9 Document Conventions.....	135

## List of Figures

Figure 1.1: Converter Module Hierarchy.....	25
Figure 2.1: Performance Comparison.....	31
Figure 2.2: axi_data_upsize Architecture.....	33
Figure 2.3: axi_data_dnsz Single-Buffer Architecture.....	40
Figure 2.4: axi_data_dnsz Dual-Buffer Architecture.....	40
Figure 2.5: Dual-Buffer Ping-Pong Operation.....	48
Figure 2.6: Write Converter Architecture.....	53
Figure 2.7: Read Converter Architecture.....	60
Figure 3.1: Protocol Feature Comparison.....	67
Figure 3.2: AXI4 to AXI4-Lite Read Path.....	70
Figure 3.3: AXI4 to AXI4-Lite Write Path.....	72
Figure 3.4: AXI4-Lite to AXI4 Read Path.....	77
Figure 3.5: AXI4-Lite to AXI4 Write Path.....	79
Figure 3.6: AXI4 to APB Converter.....	84
Figure 3.7: AXI4 to APB FSM.....	87
Figure 3.8: PeakRDL Adapter.....	94
Figure 4.1: Upsize FSM.....	101
Figure 4.2: Downsize Single-Buffer FSM.....	103
Figure 4.3: AXI4 to AXI4-Lite Read FSM.....	107
Figure 4.4: AXI4 to APB FSM.....	111
Figure 4.5: Width Burst Conversion.....	114

## List of Tables

Table 1: Table 0.1: Document Control Information.....	22
Table 2: Table 0.2: Related Documents.....	23
Table 3: Table 0.3: Revision History.....	23
Table 4: Table 1.1: Common Data Width Configurations.....	24
Table 5: Table 1.2: Protocol Mismatch Examples.....	24
Table 6: Table 1.3: Throughput vs. Area Trade-offs.....	26
Table 7: Table 1.4: Sideband Handling Modes.....	26
Table 8: Table 1.5: Latency Characteristics.....	27
Table 9: Table 1.6: Throughput Characteristics.....	27
Table 10: Table 2.1: Upsize Sideband Modes.....	30
Table 11: Table 2.2: Downsize Sideband Modes.....	30
Table 12: Table 2.3: Performance Comparison.....	31
Table 13: Table 2.4: axi_data_upsize Parameters.....	34
Table 14: Table 2.5: Upsize Latency.....	38
Table 15: Table 2.6: Upsize Resource Scaling.....	38
Table 16: Table 2.7: axi_data_dnsiz Parameters.....	41
Table 17: Table 2.8: Single-Buffer Throughput by Ratio.....	43
Table 18: Table 2.9: Resource Comparison.....	46
Table 19: Table 2.10: Dual-Buffer Cycle Timing.....	50
Table 20: Table 2.11: Dual-Buffer Latency.....	50
Table 21: Table 2.12: Resource Breakdown.....	51
Table 22: Table 2.13: Mode Trade-offs.....	51
Table 23: Table 2.14: Write Converter Parameters.....	54
Table 24: Table 2.15: Burst Length Conversion Examples.....	55
Table 25: Table 2.16: Write Converter Latency.....	58
Table 26: Table 2.17: Read Converter Parameters.....	60
Table 27: Table 2.18: Read Burst Length Conversion.....	62
Table 28: Table 2.19: Read Converter Performance.....	64
Table 29: Table 2.20: Read Converter Latency.....	65
Table 30: Table 3.1: AXI4 to AXI4-Lite Converters.....	66
Table 31: Table 3.2: AXI4-Lite to AXI4 Converters.....	67
Table 32: Table 3.3: Other Protocol Converters.....	67
Table 33: Table 3.4: Protocol Feature Comparison.....	67
Table 34: Table 3.5: Protocol Converter Performance.....	68
Table 35: Table 3.6: AXI4 to AXIL4 Resources.....	74
Table 36: Table 3.7: AXI4 to AXIL4 Throughput.....	75

Table 37: Table 3.8: AXI4 to AXIL4 Latency.....	75
Table 38: Table 3.9: Test Coverage Summary.....	75
Table 39: Table 3.10: AR Channel Mapping.....	76
Table 40: Table 3.11: R Channel Mapping.....	77
Table 41: Table 3.12: AXIL4 to AXI4 Resources.....	81
Table 42: Table 3.13: AXIL4 to AXI4 Performance.....	81
Table 43: Table 3.14: Test Coverage Summary.....	82
Table 44: Table 3.15: AXI4 vs APB Comparison.....	83
Table 45: Table 3.16: AXI4 to APB Parameters.....	85
Table 46: Table 3.17: FSM Transitions.....	88
Table 47: Table 3.18: Error Mapping.....	89
Table 48: Table 3.19: APB Converter Timing.....	91
Table 49: Table 3.20: PeakRDL Adapter Parameters.....	96
Table 50: Table 4.1: Upsize FSM States.....	101
Table 51: Table 4.2: Downsize FSM States.....	105
Table 52: Table 4.3: AXI4 to AXIL4 Read FSM States.....	107
Table 53: Table 4.4: AXI4 to AXIL4 Write FSM States.....	109
Table 54: Table 4.5: AXI4 to APB FSM States.....	112
Table 55: Table 4.6: Decomposition Overhead.....	117
Table 56: Table 5.1: Functional Test Categories.....	119
Table 57: Table 5.2: Stress Test Categories.....	120
Table 58: Table 5.3: Error Test Categories.....	120
Table 59: Table 5.4: Coverage Targets.....	122
Table 60: Table 5.5: Expected Throughput.....	127

# 1 Converter Mas Index

Generated: 2026-01-04

## 2 Document Information

### 2.1 Document Control

Table 0.1: Document Control Information

Field	Value
Document Title	Converters Micro-Architecture Specification
Document Version	1.0
Component	Converters
Status	Active
Classification	Internal Technical
Last Updated	2026-01-03

### 2.2 Purpose

This Micro-Architecture Specification (MAS) provides detailed implementation guidance for the Converters component. It covers:

- Internal block architectures
- State machine designs
- Signal timing and handshaking
- Resource utilization estimates
- Debug and verification strategies

### 2.3 Audience

This document is intended for:

- RTL designers implementing or modifying converter modules
- Verification engineers creating testbenches
- Integration engineers connecting converters to systems
- Performance engineers optimizing throughput and latency

## 2.4 Related Documents

---

Table 0.2: Related Documents

Document	Purpose
Converters Spec	High-level feature specification
Component PRD	Product requirements and goals
Bridge MAS	Related crossbar micro-architecture
Stream MAS	Related datapath micro-architecture

## 2.5 Revision History

---

Table 0.3: Revision History

Version	Date	Author	Description
1.0	2026-01-03	RTL Design Sherpa	Initial MAS release

## 2.6 Conventions

---

### 2.6.1 Notation

- `signal_name` - RTL signals and parameters
- **ModuleName** - Module names and key concepts
- *Figure X.X* - Figure references

### 2.6.2 Diagrams

All diagrams use Mermaid format rendered to PNG: - Source: `assets/mermaid/*.mmd` -  
Rendered: `assets/mermaid/*.png`

### 2.6.3 Code Examples

SystemVerilog code snippets are provided for implementation guidance. These represent the intended design pattern but may differ slightly from actual RTL.

---

Next: [Chapter 1: Introduction](#)

## 3 Chapter 1: Introduction - Overview

### 3.1 1.1 Purpose

The Converters component provides essential data width conversion and protocol conversion modules that enable seamless integration between components with different data widths or communication protocols.

### 3.2 1.2 Problem Statement

Modern SoC designs frequently encounter two integration challenges:

#### 3.2.1 1.2.1 Data Width Mismatch

*Table 1.1: Common Data Width Configurations*

Component	Typical Data Width
CPU	64-bit
DDR Controller	512-bit
PCIe Endpoint	128-bit
GPU	256-bit

**Challenge:** Direct connection between mismatched widths is impossible. Width conversion is required.

#### 3.2.2 1.2.2 Protocol Incompatibility

*Table 1.2: Protocol Mismatch Examples*

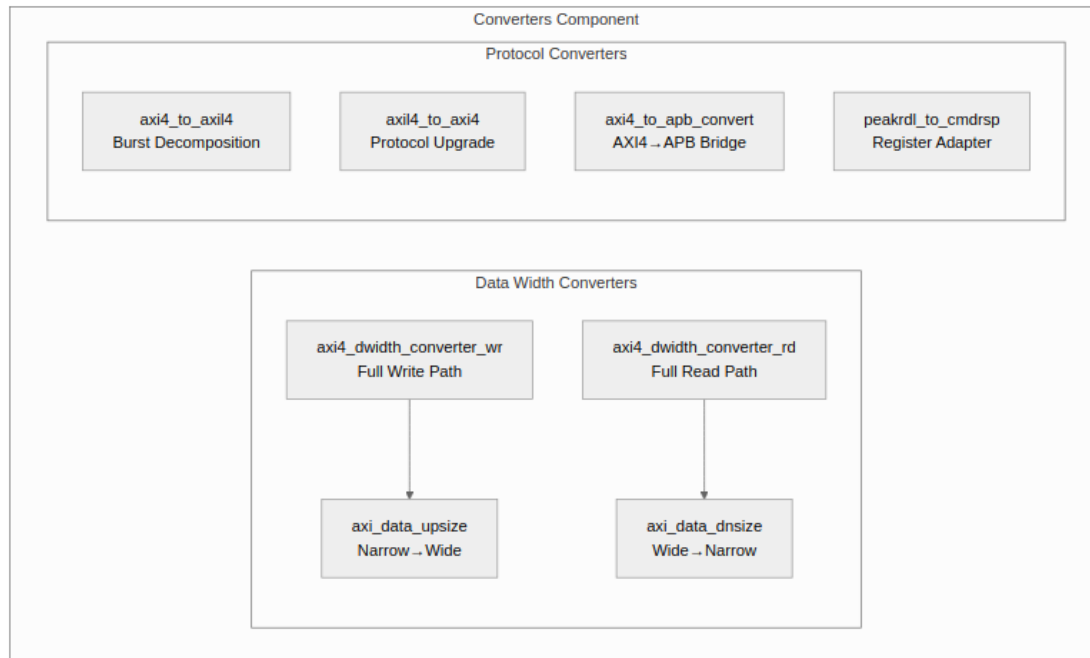
Master Type	Protocol	Slave Type	Protocol
CPU	AXI4	DDR	AXI4
DMA	AXI4	UART	APB
CPU	AXI4	GPIO	APB
Custom IP	AXI4-Lite	Fabric	AXI4

**Challenge:** Different protocols require protocol bridges for communication.



## 3.3 1.3 Solution Architecture

### 3.3.1 Figure 1.1: Converter Module Hierarchy



#### Converter Module Hierarchy

### 3.3.2 1.3.1 Data Width Converters

**Generic Building Blocks:** - **axi\_data\_upsize** - Accumulates N narrow beats into 1 wide beat - **axi\_data\_dsize** - Splits 1 wide beat into N narrow beats

**Full AXI4 Converters:** - **axi4\_dwidth\_converter\_wr** - Complete write path (AW + W + B channels) - **axi4\_dwidth\_converter\_rd** - Complete read path (AR + R channels)

### 3.3.3 1.3.2 Protocol Converters

**AXI4 to AXI4-Lite:** - **axi4\_to\_axil4\_rd** - Read path burst decomposition - **axi4\_to\_axil4\_wr** - Write path burst decomposition - **axi4\_to\_axil4** - Full bidirectional wrapper

**AXI4-Lite to AXI4:** - **axil4\_to\_axi4\_rd** - Read path protocol upgrade - **axil4\_to\_axi4\_wr** - Write path protocol upgrade - **axil4\_to\_axi4** - Full bidirectional wrapper

**Other Converters:** - **axi4\_to\_apb\_convert** - Full AXI4-to-APB bridge - **peakrdl\_to\_cmdrsp** - Register interface adapter

## 3.4 1.4 Key Design Decisions

### 3.4.1 1.4.1 Generic vs. Full Modules

The converter architecture uses a layered approach:

Layer 1: Generic Building Blocks

- `axi_data_upsize`, `axi_data_dnsize`
- Protocol-agnostic data manipulation
- Reusable across different contexts

Layer 2: Full AXI4 Converters

- `axi4_dwidth_converter_wr`, `axi4_dwidth_converter_rd`
- Compose generic blocks with AXI4 channel management
- Handle burst length adjustment, ID tracking

Layer 3: Protocol Converters

- `axi4_to_axil4`, `axi4_to_apb`, etc.
- Full protocol translation
- State machine control

### 3.4.2 1.4.2 Throughput vs. Area Trade-offs

*Table 1.3: Throughput vs. Area Trade-offs*

Configuration	Throughput	Area	Use Case
Upsize (single buffer)	100%	1x	All narrow-to-wide
Downsize (single buffer)	80%	1x	Area-constrained
Downsize (dual buffer)	100%	2x	High-performance

**Design Decision:** Single-buffer upsize is always optimal (100% throughput at minimal cost). Downsize mode is configurable based on system requirements.

### 3.4.3 1.4.3 Sideband Signal Handling

Different modes for handling sideband signals (WSTRB, RRESP, etc.):

*Table 1.4: Sideband Handling Modes*

Mode	Upsize Behavior	Downsize Behavior
Concatenate	Pack narrow strobes	Slice wide strobes
Broadcast	N/A	Repeat value

Mode	Upsize Behavior	Downsize Behavior
OR	Combine with OR	N/A

## 3.5 1.5 Performance Characteristics

### 3.5.1 1.5.1 Latency

*Table 1.5: Latency Characteristics*

Module	Single-Beat	Burst (N beats)
axi_data_upsize	0 cycles	N cycles to accumulate
axi_data_dnsiz (single)	1 cycle	N cycles + gap
axi_data_dnsiz (dual)	1 cycle	N cycles
axi4_to_axil4	0 cycles	2xN cycles
axil4_to_axi4	0 cycles	N/A (single only)

### 3.5.2 1.5.2 Throughput

*Table 1.6: Throughput Characteristics*

Module	Configuration	Peak Throughput
axi_data_upsize	Single buffer	1 beat/cycle
axi_data_dnsiz	Single buffer	0.8 beats/cycle
axi_data_dnsiz	Dual buffer	1 beat/cycle
axi4_to_axil4	Burst	0.5 beats/cycle
axil4_to_axi4	Any	1 beat/cycle

## 3.6 1.6 Scope

### 3.6.1 In Scope

- Integer width ratios (2:1, 4:1, 8:1, 16:1, etc.)
- AXI4, AXI4-Lite, and APB protocol support
- Configurable throughput vs. area trade-offs
- Generic building blocks for custom converters
- Burst-aware width conversion

### 3.6.2 Out of Scope

- Non-integer width ratios (e.g., 3:2 conversion)
- AXI4-Stream protocol (see Stream component)
- Complex buffering beyond dual-buffer
- Clock domain crossing (use separate CDC modules)
- Address translation (handled by crossbar)

## 3.7 1.7 Target Applications

---

1. **CPU-to-DDR Integration** - 64-bit CPU to 512-bit memory controller
  2. **DMA Engines** - Variable width data movers
  3. **Mixed Protocol Systems** - AXI4 fabric with APB peripheral bus
  4. **FPGA Fabric Interfaces** - Width matching for IP integration
  5. **Register Access** - PeakRDL to custom control protocols
- 

Next: [Chapter 2: Data Width Converter Blocks](#)

## 4 2.1 Generic Building Blocks

The Converters component provides two generic building blocks for data width conversion: **axi\_data\_upsize** (narrow-to-wide) and **axi\_data\_dnsize** (wide-to-narrow). These modules are protocol-agnostic and can be composed into full AXI4 converters or used directly in custom designs.

### 4.1 2.1.1 Module Hierarchy

---

Generic Building Blocks

```
|
+-- axi_data_upsize.sv      # Narrow-to-Wide accumulator
|   +-- Accumulator buffer
|   +-- Beat counter
|   +-- Sideband packing
|
+-- axi_data_dnsize.sv      # Wide-to-Narrow splitter
    +-- Data buffer (single or dual)
    +-- Beat counter
    +-- Sideband extraction
    +-- Optional burst tracking
```

## 4.2 2.1.2 Design Philosophy

---

### 4.2.1 Separation of Concerns

The generic blocks handle **only data manipulation**: - Data packing/unpacking - Sideband signal handling - Valid/ready handshaking

They do **not** handle: - Address manipulation - Burst length adjustment - ID tracking - Protocol-specific signals (ARLEN, AWLEN, etc.)

This separation enables: - Reuse in multiple contexts - Simpler verification - Cleaner composition into full converters

### 4.2.2 Interface Pattern

Both modules use a consistent valid/ready interface:

```
// Input (narrow for upsize, wide for dsize)
input logic          i_valid,
output logic         o_ready,
input logic [DATA_WIDTH-1:0] i_data,
input logic [SB_WIDTH-1:0] i_sideband,
input logic          i_last,          // Optional

// Output (wide for upsize, narrow for dsize)
output logic         o_valid,
input logic          i_ready,
output logic [DATA_WIDTH-1:0] o_data,
output logic [SB_WIDTH-1:0] o_sideband,
output logic         o_last          // Optional
```

## 4.3 2.1.3 Width Ratio Calculation

---

Both modules calculate the width ratio at elaboration time:

```
localparam int RATIO = WIDE_WIDTH / NARROW_WIDTH;
localparam int RATIO_LOG2 = $clog2(RATIO);
```

```
// Example: 64-bit to 512-bit
// RATIO = 512 / 64 = 8
// RATIO_LOG2 = 3
```

**Constraints:** - WIDE\_WIDTH must be an integer multiple of NARROW\_WIDTH - Minimum ratio: 2 - Maximum ratio: Typically 16 (limited by timing)

## 4.4 2.1.4 Sideband Modes

### 4.4.1 Upsize Sideband Modes

Table 2.1: Upsize Sideband Modes

Mode	Parameter	Behavior	Use Case
Concatenate	SB_OR_MODE=0	Pack N narrow sidebands	WSTRB packing
OR	SB_OR_MODE=1	OR all narrow sidebands	Error aggregation

#### Concatenate Example (WSTRB):

8 beats of 8-bit WSTRB → 1 beat of 64-bit WSTRB

[0]: 0xFF → output[7:0] = 0xFF

[1]: 0xF0 → output[15:8] = 0xF0

...

[7]: 0x0F → output[63:56] = 0x0F

#### OR Example (Error flags):

8 beats of error flags → 1 beat with any error

[0]: 0 → output = 0

[1]: 1 → output = 1 (error detected)

...

[7]: 0 → output remains 1

### 4.4.2 Downsize Sideband Modes

Table 2.2: Downsize Sideband Modes

Mode	Parameter	Behavior	Use Case
Slice	SB_BROADCAST=0	Extract slice per beat	WSTRB extraction
Broadcast	SB_BROADCAST=1	Repeat full value	RRESP broadcast

#### Slice Example (WSTRB):

1 beat of 64-bit WSTRB → 8 beats of 8-bit WSTRB

input = 0xFF\_F0...\_0F

[0]: output = 0xFF (bits [7:0])

[1]: output = 0xF0 (bits [15:8])

...

[7]: output = 0x0F (bits [63:56])

### Broadcast Example (RRESP):

1 beat of 2-bit RRESP → 8 beats of 2-bit RRESP  
input = OKAY (2'b00)  
[0-7]: output = OKAY (all beats get same response)

## 4.5 2.1.5 Performance Comparison

### 4.5.1 Figure 2.1: Performance Comparison

#### Performance Comparison

*Performance Comparison*

*Table 2.3: Performance Comparison*

Module	Mode	Throughput	Latency	Area
axi_data_u psize	Single	100%	N cycles	1x
axi_data_d nsize	Single	80%	1 cycle	1x
axi_data_d nsize	Dual	100%	1 cycle	2x

**Why 80% for single-buffer downsize?** - Single buffer requires one cycle gap between wide beats  
- Wide beat loaded → N narrow beats output → gap → next wide beat - Gap cycle =  $1/(N+1)$   
throughput loss - For large N, approaches 100% but never reaches it

**Why 100% for dual-buffer downsize?** - Ping-pong between two buffers - While one buffer outputs, other loads - No gap cycles required

## 4.6 2.1.6 Resource Utilization

### 4.6.1 Upsize Resources (NARROW=64, WIDE=512)

Accumulator buffer: 512 bits (output register)  
Beat counter: 3 bits ( $\text{clog}_2(8)$ )  
Sideband accumulator: 64 bits (WSTRB)  
Control logic: ~50 LEs

Total: ~70-100 LEs, ~580 registers

### 4.6.2 Downsize Resources (WIDE=512, NARROW=64)

**Single Buffer:**

Data buffer: 512 bits  
Beat counter: 3 bits  
Sideband logic: ~20 LEs  
Control logic: ~50 LEs

Total: ~70-100 LEs, ~520 registers

#### Dual Buffer:

Data buffers (2x): 1024 bits  
Beat counters (2x): 6 bits  
Sideband logic: ~40 LEs  
Control logic: ~100 LEs (ping-pong FSM)

Total: ~140-180 LEs, ~1040 registers

## 4.7 2.1.7 Integration Guidelines

---

### 4.7.1 When to Use Generic Blocks

**Use directly when:** - Building custom data pipelines - Data width conversion without AXI4 protocol - Simple valid/ready streaming interfaces

**Use full converters when:** - Need AXI4 channel management (AW, W, B, AR, R) - Burst length adjustment required - ID tracking needed

### 4.7.2 Composition Example

```
// Full AXI4 write path composition
axi4_dwidth_converter_wr #(.S_DATA_WIDTH(64), .M_DATA_WIDTH(512)) u_wr
(
    // This module internally instantiates:
    // 1. Address phase skid buffer
    // 2. axi_data_upsize for write data
    // 3. Response path passthrough
    // 4. Burst length adjustment logic
);
```

---

Next: [axi\\_data\\_upsize Module](#)

## 5 2.2 axi\_data\_upsize Module

The **axi\_data\_upsize** module accumulates N narrow beats into 1 wide beat. It is the core building block for narrow-to-wide data width conversion.



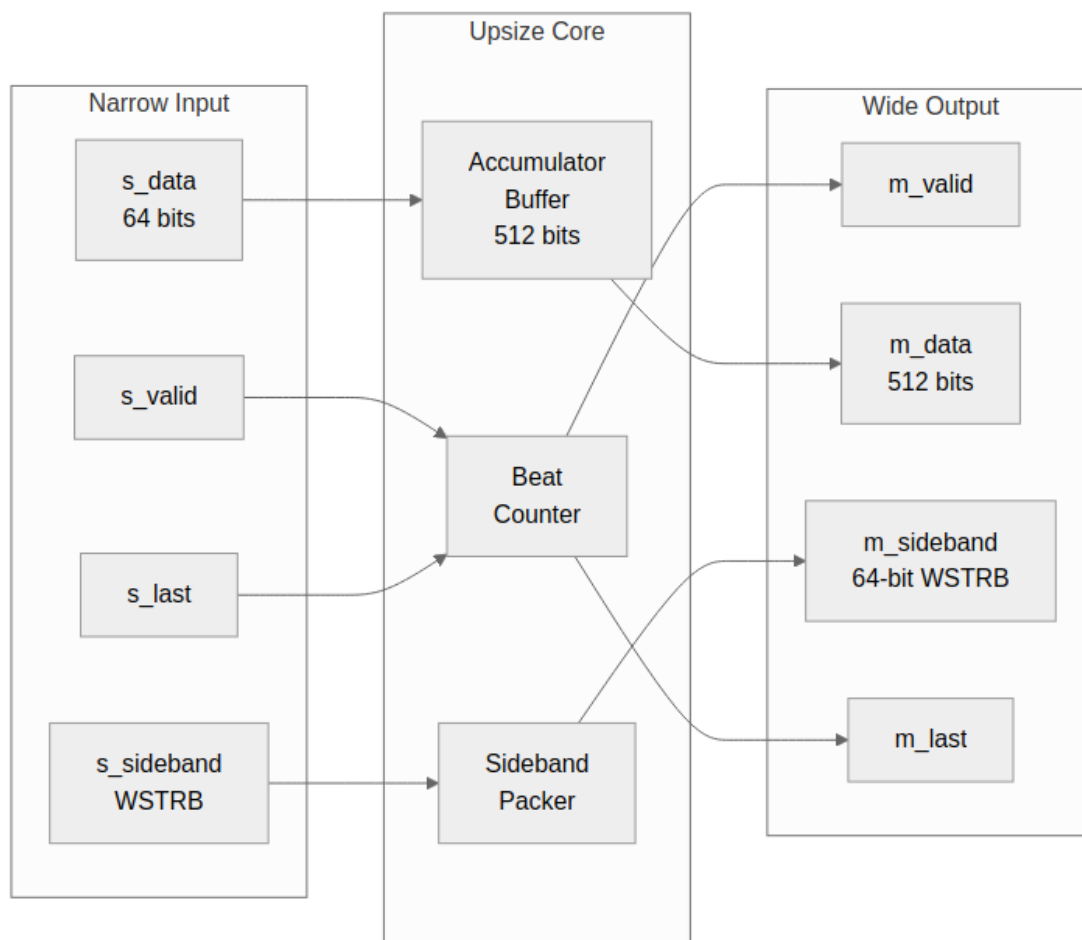
## 5.1 2.2.1 Purpose and Function

The upsize module serves several critical functions:

1. **Data Accumulation:** Collects N narrow data beats into accumulator buffer
2. **Sideband Packing:** Concatenates or ORs sideband signals (WSTRB, etc.)
3. **Flow Control:** Manages valid/ready handshaking with single-cycle latency
4. **LAST Tracking:** Detects input LAST to generate output LAST

## 5.2 2.2.2 Block Diagram

5.2.1 Figure 2.2: axi\_data\_upsize Architecture



*axi\_data\_upsize* Architecture

## 5.3 2.2.3 Interface Specification

### 5.3.1 Parameters

Table 2.4: *axi\_data\_upsize* Parameters

Parameter	Type	Default	Description
NARROW_WIDTH	int	64	Input data width (bits)
WIDE_WIDTH	int	512	Output data width (bits)
NARROW_SB_WIDTH	int	8	Input sideband width (bits)
WIDE_SB_WIDTH	int	64	Output sideband width
SB_OR_MODE	bit	0	0=concatenate, 1=OR sidebands
USE_LAST	bit	1	Enable LAST signal tracking

### 5.3.2 Ports

```
module axi_data_upsize #(
    parameter int NARROW_WIDTH    = 64,
    parameter int WIDE_WIDTH      = 512,
    parameter int NARROW_SB_WIDTH = 8,
    parameter int WIDE_SB_WIDTH  = 64,
    parameter bit SB_OR_MODE     = 0,
    parameter bit USE_LAST       = 1
) (
    input logic          clk,
    input logic          rst_n,

    // Narrow input interface
    input logic          s_valid,
    output logic         s_ready,
    input logic [NARROW_WIDTH-1:0] s_data,
    input logic [NARROW_SB_WIDTH-1:0] s_sideband,
    input logic          s_last,

    // Wide output interface
```

```

        output logic [WIDE_WIDTH-1:0] m_valid,
        input logic m_ready,
        output logic [WIDE_SB_WIDTH-1:0] m_data,
        output logic [WIDE_SB_WIDTH-1:0] m_sideband,
        output logic m_last
    );

```

## 5.4 2.2.4 Operation

---

### 5.4.1 Accumulation Cycle

Cycle 0: s\_data[0] → buffer[63:0], count = 0 → 1  
 Cycle 1: s\_data[1] → buffer[127:64], count = 1 → 2  
 ...  
 Cycle 7: s\_data[7] → buffer[511:448], count = 7 → 0, m\_valid = 1  
 Cycle 8: m\_ready handshake, output complete

### 5.4.2 Early LAST Handling

If s\_last arrives before buffer is full:

Cycle 0: s\_data[0] → buffer[63:0], count = 0 → 1  
 Cycle 1: s\_data[1] + s\_last → buffer[127:64], count = 1 → 0  
         m\_valid = 1, m\_last = 1  
         Remaining bytes = don't care (masked by WSTRB)

### 5.4.3 State Machine

IDLE (count=0):  
 - s\_valid=1 → load beat, increment count  
 - count < RATIO-1 → stay in IDLE  
 - count = RATIO-1 OR s\_last → OUTPUT

OUTPUT (m\_valid=1):  
 - m\_ready=1 → clear buffer, → IDLE  
 - m\_ready=0 → hold output

## 5.5 2.2.5 Sideband Handling

---

### 5.5.1 Concatenate Mode (SB\_OR\_MODE=0)

Used for WSTRB packing:

```

// Pack narrow sidebands into wide sideband
always_ff @(posedge clk) begin
    if (s_valid && s_ready) begin
        r_sideband[r_count * NARROW_SB_WIDTH +: NARROW_SB_WIDTH] <=
s_sideband;
    end
end

```

**end**  
**end**

**Example:** 8 beats of 8-bit WSTRB to 64-bit WSTRB

Beat 0: WSTRB = 0xFF → output[7:0] = 0xFF  
Beat 1: WSTRB = 0xF0 → output[15:8] = 0xF0  
Beat 2: WSTRB = 0x0F → output[23:16] = 0x0F  
...  
Beat 7: WSTRB = 0xAA → output[63:56] = 0xAA  
Final: output = 0xAA...\_0F\_F0\_FF

## 5.5.2 OR Mode (SB\_OR\_MODE=1)

Used for error flag aggregation:

```
// OR narrow sidebands together
always_ff @(posedge clk) begin
    if (s_valid && s_ready) begin
        if (r_count == 0)
            r_sideband <= {{(WIDE_SB_WIDTH-NARROW_SB_WIDTH)}{1'b0}},
s_sideband};
        else
            r_sideband <= r_sideband | s_sideband;
    end
end
```

**Example:** Any error in burst propagates

Beat 0: error = 0 → output = 0  
Beat 1: error = 0 → output = 0  
Beat 2: error = 1 → output = 1 (error detected)  
Beat 3: error = 0 → output = 1 (remains set)  
...  
Final: output = 1 (error occurred in burst)

## 5.6 2.2.6 Implementation

---

### 5.6.1 Core Logic

```
// Beat counter
logic [$clog2(RATIO)-1:0] r_count;

// Accumulator buffer
logic [WIDE_WIDTH-1:0] r_data;
logic [WIDE_SB_WIDTH-1:0] r_sideband;
logic r_last;

// Output valid when buffer full or early LAST
```

```

logic w_output_valid;
assign w_output_valid = (r_count == RATIO - 1) || r_last;

// Ready when not outputting or downstream ready
assign s_ready = !w_output_valid || m_ready;

// Main accumulation logic
always_ff @(posedge clk or negedge rst_n) begin
    if (!rst_n) begin
        r_count <= '0;
        r_data <= '0;
        r_sideband <= '0;
        r_last <= 1'b0;
    end else if (s_valid && s_ready) begin
        // Pack data into buffer
        r_data[r_count * NARROW_WIDTH +: NARROW_WIDTH] <= s_data;

        // Handle sideband based on mode
        if (SB_OR_MODE)
            r_sideband <= (r_count == 0) ? s_sideband : (r_sideband |
s_sideband);
        else
            r_sideband[r_count * NARROW_SB_WIDTH +: NARROW_SB_WIDTH]
<= s_sideband;

        // Track LAST
        r_last <= s_last;

        // Update counter
        if (s_last || r_count == RATIO - 1)
            r_count <= '0;
        else
            r_count <= r_count + 1'b1;
    end else if (m_valid && m_ready) begin
        r_last <= 1'b0;
    end
end

// Output assignments
assign m_valid = w_output_valid;
assign m_data = r_data;
assign m_sideband = r_sideband;
assign m_last = r_last;

```

## 5.7 2.2.7 Timing Characteristics

### 5.7.1 Latency

Table 2.5: Upsize Latency

Scenario	Latency
Full buffer (N beats)	N cycles
Early LAST (M beats)	M cycles
Output handshake	0-1 cycles

### 5.7.2 Throughput

**100% throughput** - No gaps required between input beats.

The accumulator accepts one beat per cycle continuously. When the output buffer is ready, it completes the handshake and immediately starts accumulating the next wide beat.

### 5.7.3 Critical Paths

Typical critical paths: - s\_data → accumulator buffer → m\_data - r\_count → comparison → s\_ready

**Timing closure:** The module is designed for single-cycle operation with combinatorial paths only within registered stages.

## 5.8 2.2.8 Resource Utilization

### 5.8.1 Typical Resources (64-bit to 512-bit)

Accumulator buffer: 512 flip-flops  
Sideband buffer: 64 flip-flops (WSTRB)  
Beat counter: 3 flip-flops  
Control logic: ~20 flip-flops  
~50-70 LUTs

Total: ~600 flip-flops, ~50-70 LUTs

### 5.8.2 Scaling

Table 2.6: Upsize Resource Scaling

Configuration	Registers	LUTs
32 → 128 (4:1)	~170	~30
64 → 256 (4:1)	~330	~40
64 → 512 (8:1)	~600	~60

Configuration	Registers	LUTs
128 → 1024 (8:1)	~1150	~80

## 5.9 2.2.9 Usage Example

### 5.9.1 64-bit to 512-bit Write Data

```
axi_data_upsize #(
    .NARROW_WIDTH(64),
    .WIDE_WIDTH(512),
    .NARROW_SB_WIDTH(8),    // WSTRB
    .WIDE_SB_WIDTH(64),
    .SB_OR_MODE(0),        // Concatenate WSTRB
    .USE_LAST(1)
) u_wdata_upsize (
    .clk          (aclk),
    .rst_n        (aresetn),
    .s_valid      (s_wvalid),
    .s_ready      (s_wready),
    .s_data       (s_wdata),
    .s_sideband   (s_wstrb),
    .s_last       (s_wlast),
    .m_valid      (m_wvalid),
    .m_ready      (m_wready),
    .m_data       (m_wdata),
    .m_sideband   (m_wstrb),
    .m_last       (m_wlast)
);
```

Next: [axi\\_data\\_dnsz Module](#)

## 6 2.3 axi\_data\_dnsz Module

The **axi\_data\_dnsz** module splits 1 wide beat into N narrow beats. It supports both single-buffer (80% throughput) and dual-buffer (100% throughput) modes.

### 6.1 2.3.1 Purpose and Function

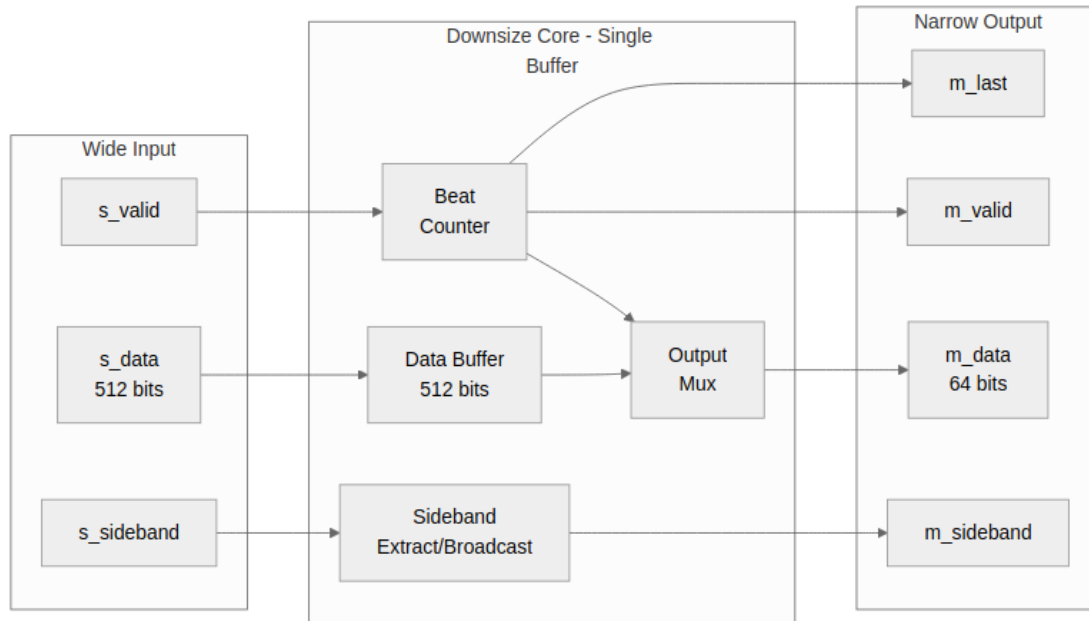
The downsize module serves several critical functions:

1. **Data Splitting:** Extracts N narrow beats from one wide beat
2. **Sideband Extraction:** Slices or broadcasts sideband signals
3. **Dual-Buffer Mode:** Optional ping-pong buffering for 100% throughput

4. **Burst Tracking:** Optional LAST signal generation based on burst length

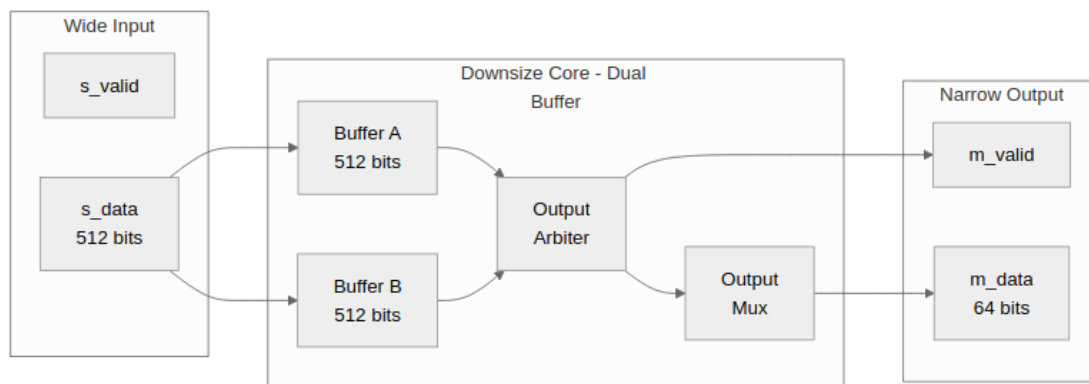
## 6.2 2.3.2 Block Diagram

6.2.1 Figure 2.3: axi\_data\_dnsz Single-Buffer Architecture



*axi\_data\_dnsz Single Buffer*

6.2.2 Figure 2.4: axi\_data\_dnsz Dual-Buffer Architecture



*axi\_data\_dnsz Dual Buffer*



## 6.3 2.3.3 Interface Specification

### 6.3.1 Parameters

Table 2.7: *axi\_data\_dnsiz*e Parameters

Parameter	Type	Default	Description
WIDE_WIDTH	int	512	Input data width (bits)
NARROW_WIDTH	int	64	Output data width (bits)
WIDE_SB_WIDTH	int	2	Input sideband width
NARROW_SB_WIDTH	int	2	Output sideband width
SB_BROADCAST	bit	1	0=slice, 1=broadcast sidebands
DUAL_BUFFER	bit	0	0=single, 1=dual buffer mode
USE_BURST_TRACKER	bit	0	Enable burst-aware LAST generation
BURST_LEN_WIDTH	int	8	Width of burst length input

### 6.3.2 Ports

```
module axi_data_dnsiz #(
    parameter int WIDE_WIDTH      = 512,
    parameter int NARROW_WIDTH    = 64,
    parameter int WIDE_SB_WIDTH   = 2,
    parameter int NARROW_SB_WIDTH = 2,
    parameter bit SB_BROADCAST    = 1,
    parameter bit DUAL_BUFFER     = 0,
    parameter bit USE_BURST_TRACKER = 0,
    parameter int BURST_LEN_WIDTH = 8
) (
    input logic                clk,
```

```

input logic                                rst_n,

// Wide input interface
input logic                                s_valid,
output logic                               s_ready,
input logic [WIDE_WIDTH-1:0]              s_data,
input logic [WIDE_SB_WIDTH-1:0]           s_sideband,
input logic                                s_last,

// Burst length input (optional)
input logic [BURST_LEN_WIDTH-1:0] burst_len,

// Narrow output interface
output logic                               m_valid,
input logic                               m_ready,
output logic [NARROW_WIDTH-1:0]           m_data,
output logic [NARROW_SB_WIDTH-1:0] m_sideband,
output logic                               m_last
);

```

## 6.4 2.3.4 Single-Buffer Mode Operation

---

### 6.4.1 Split Cycle

Cycle 0: s\_data loaded → buffer, s\_ready = 0  
 Cycle 1: m\_data = buffer[63:0], count = 0, m\_valid = 1  
 Cycle 2: m\_data = buffer[127:64], count = 1  
 ...  
 Cycle 8: m\_data = buffer[511:448], count = 7, m\_last possible  
 Cycle 9: s\_ready = 1, gap cycle (80% throughput loss)

### 6.4.2 State Machine

IDLE:  
 - s\_valid=1 → load buffer → SPLITTING

SPLITTING:  
 - Output beats 0 to RATIO-1  
 - m\_ready=1 → increment count  
 - count=RATIO-1 AND m\_ready → IDLE

### 6.4.3 Throughput Analysis

#### Why 80%?

For ratio N, the cycle utilization is: - N cycles outputting narrow beats - 1 cycle loading next wide beat

Throughput =  $N / (N + 1)$

*Table 2.8: Single-Buffer Throughput by Ratio*

Ratio	Cycles Active	Cycles Total	Throughput
2:1	2	3	66.7%
4:1	4	5	80.0%
8:1	8	9	88.9%
16:1	16	17	94.1%

## 6.5 2.3.5 Dual-Buffer Mode Operation

### 6.5.1 Ping-Pong Operation

Buffer A	Buffer B	Output
-----	-----	-----
Load beat 0	(empty)	(idle)
Outputting 0	Load beat 1	beat 0[0]
Outputting 0	Outputting 1	beat 0[1]
...	...	...
Outputting 0	Outputting 1	beat 0[N-1]
Load beat 2	Outputting 1	beat 1[0]
Outputting 2	Outputting 1	beat 1[1]
...		

### 6.5.2 State Machine (Dual)

Buffer A State: LOADING | OUTPUTTING | DONE  
 Buffer B State: LOADING | OUTPUTTING | DONE

Arbiter selects active output buffer  
 When output complete, swap buffers

### 6.5.3 100% Throughput

Dual-buffer achieves 100% throughput because: - While buffer A outputs, buffer B loads - While buffer B outputs, buffer A loads - No gap cycles required

**Trade-off:** 2x register resources

## 6.6 2.3.6 Sideband Handling

### 6.6.1 Slice Mode (SB\_BROADCAST=0)

Used for WSTRB extraction:

```
// Extract sideband slice per beat
assign m_sideband = r_sideband[r_count * NARROW_SB_WIDTH +:
NARROW_SB_WIDTH];
```

**Example:** 64-bit WSTRB to 8-bit WSTRB

Input: 0xAA\_BB\_CC\_DD\_EE\_FF\_00\_11

Beat 0: output WSTRB = 0x11 (bits [7:0])  
Beat 1: output WSTRB = 0x00 (bits [15:8])  
Beat 2: output WSTRB = 0xFF (bits [23:16])  
...  
Beat 7: output WSTRB = 0xAA (bits [63:56])

## 6.6.2 Broadcast Mode (SB\_BROADCAST=1)

Used for RRESP:

```
// Broadcast same sideband to all beats
assign m_sideband = r_sideband[NARROW_SB_WIDTH-1:0];
```

**Example:** RRESP = OKAY for all beats

Input: RRESP = 2'b00 (OKAY)

Beat 0: output RRESP = 2'b00  
Beat 1: output RRESP = 2'b00  
...  
Beat 7: output RRESP = 2'b00

## 6.7 2.3.7 Burst Tracking

---

### 6.7.1 Purpose

When USE\_BURST\_TRACKER=1, the module generates correct m\_last based on AXI4 burst length instead of relying on input s\_last.

### 6.7.2 Operation

```
// Track narrow beats across burst
logic [BURST_LEN_WIDTH+RATIO_LOG2-1:0] r_burst_beats_remaining;

// Initialize on first beat
if (first_beat)
    r_burst_beats_remaining <= (burst_len + 1) * RATIO - 1;

// Decrement on each output
if (m_valid && m_ready)
    r_burst_beats_remaining <= r_burst_beats_remaining - 1;
```

```
// Generate LAST
assign m_last = (r_burst_beats_remaining == 0);
```

**Example:** ARLEN=3 (4 beats), ratio 8:1

Total narrow beats = 4 \* 8 = 32  
 Beat 0-30: m\_last = 0  
 Beat 31: m\_last = 1

## 6.8 2.3.8 Implementation

---

### 6.8.1 Single-Buffer Core Logic

```
// Beat counter
logic [$clog2(RATIO)-1:0] r_count;
logic r_active;

// Data buffer
logic [WIDE_WIDTH-1:0] r_data;
logic [WIDE_SB_WIDTH-1:0] r_sideband;
logic r_last_wide;

// Load/output control
always_ff @(posedge clk or negedge rst_n) begin
    if (!rst_n) begin
        r_active <= 1'b0;
        r_count <= '0;
    end else begin
        if (!r_active && s_valid) begin
            // Load new wide beat
            r_data <= s_data;
            r_sideband <= s_sideband;
            r_last_wide <= s_last;
            r_active <= 1'b1;
            r_count <= '0;
        end else if (r_active && m_ready) begin
            if (r_count == RATIO - 1) begin
                r_active <= 1'b0; // Done with this beat
            end else begin
                r_count <= r_count + 1'b1;
            end
        end
    end
end

// Output data slice
assign m_data = r_data[r_count * NARROW_WIDTH +: NARROW_WIDTH];
```

```

// Sideband (slice or broadcast)
assign m_sideband = SB_BROADCAST ?
    r_sideband[NARROW_SB_WIDTH-1:0] :
    r_sideband[r_count * NARROW_SB_WIDTH +: NARROW_SB_WIDTH];

// Control signals
assign m_valid = r_active;
assign s_ready = !r_active;
assign m_last = r_last_wide && (r_count == RATIO - 1);

```

## 6.9 2.3.9 Resource Utilization

---

### 6.9.1 Single-Buffer (512-bit to 64-bit)

Data buffer: 512 flip-flops  
 Sideband buffer: 64 flip-flops  
 Beat counter: 3 flip-flops  
 Control logic: ~10 flip-flops  
                   ~30-50 LUTs

Total: ~590 flip-flops, ~30-50 LUTs

### 6.9.2 Dual-Buffer (512-bit to 64-bit)

Data buffers (2x): 1024 flip-flops  
 Sideband buffers: 128 flip-flops  
 Beat counters (2x): 6 flip-flops  
 Control logic: ~30 flip-flops  
 Ping-pong FSM: ~50 LUTs

Total: ~1190 flip-flops, ~80-100 LUTs

### 6.9.3 Comparison

*Table 2.9: Resource Comparison*

Mode	Registers	LUTs	Throughput
Single	590	40	80%
Dual	1190	90	100%

**Decision Guide:** - Area-constrained: Use single buffer, accept 80% throughput - Performance-critical: Use dual buffer, accept 2x resources

## 6.10 2.3.10 Usage Example

---

### 6.10.1 512-bit to 64-bit Read Data (High Performance)

```
axi_data_dnsiz #(  
    .WIDE_WIDTH(512),  
    .NARROW_WIDTH(64),  
    .WIDE_SB_WIDTH(2),           // RRESP  
    .NARROW_SB_WIDTH(2),  
    .SB_BROADCAST(1),           // Broadcast RRESP  
    .DUAL_BUFFER(1),            // 100% throughput  
    .USE_BURST_TRACKER(1),       // Generate RLAST  
    .BURST_LEN_WIDTH(8)  
) u_rdata_dnsiz (  
    .clk      (aclk),  
    .rst_n    (aresetn),  
    .s_valid  (s_rvalid),  
    .s_ready  (s_rready),  
    .s_data   (s_rdata),  
    .s_sideband (s_rresp),  
    .s_last   (s_rlast),  
    .burst_len (ar_len),  
    .m_valid  (m_rvalid),  
    .m_ready  (m_rready),  
    .m_data   (m_rdata),  
    .m_sideband (m_rresp),  
    .m_last   (m_rlast)  
);
```

---

Next: [Dual-Buffer Architecture](#)

## 7 2.4 Dual-Buffer Architecture

The dual-buffer mode for **axi\_data\_dnsiz** achieves 100% throughput by using ping-pong buffering to eliminate the gap cycle between wide beat loads.

### 7.1 2.4.1 Problem Statement

---

#### 7.1.1 Single-Buffer Limitation

In single-buffer mode, there is an unavoidable gap cycle between completing output of one wide beat and starting output of the next:

Cycle N: Output last narrow beat of wide beat A  
 Cycle N+1: Load wide beat B (gap - no output)  
 Cycle N+2: Output first narrow beat of wide beat B

This gap cycle reduces throughput to  $N/(N+1)$ , or approximately 80-90% depending on the width ratio.

### 7.1.2 High-Performance Requirements

Some applications require continuous 100% throughput: - DDR memory controllers with sustained bandwidth - DMA engines with continuous data flows - Real-time video/audio processing

## 7.2 2.4.2 Solution: Ping-Pong Buffering

### 7.2.1 Concept

Use two buffers that alternate between loading and outputting:

Time:	0	1	2	3	4	5	6	7	8	9	10	11
Buffer A:	LD	00	01	02	03	04	05	06	07	LD	00	01
Buffer B:		LD	--	--	--	--	--	--	--	LD	--	--
Output:	--	A0	A1	A2	A3	A4	A5	A6	A7	B0	B1	B2

LD = Loading, On = Outputting beat n

While buffer A outputs its 8 narrow beats, buffer B loads the next wide beat. When buffer A completes, buffer B immediately starts outputting while buffer A loads.

### 7.2.2 Figure 2.5: Dual-Buffer Ping-Pong Operation



## 7.3 2.4.3 Implementation

### 7.3.1 Buffer State Machine

Each buffer has three states:

```
typedef enum logic [1:0] {
    BUF_IDLE      = 2'b00, // Empty, ready to load
    BUF_LOADED    = 2'b01, // Full, waiting to output
    BUF_OUTPUTTING = 2'b10 // Active output
} buf_state_t;

buf_state_t r_buf_a_state, r_buf_b_state;
```



### 7.3.2 Output Arbiter

```
// Select which buffer outputs
logic w_output_sel; // 0 = buffer A, 1 = buffer B

always_comb begin
    if (r_buf_a_state == BUF_OUTPUTTING)
        w_output_sel = 1'b0;
    else if (r_buf_b_state == BUF_OUTPUTTING)
        w_output_sel = 1'b1;
    else if (r_buf_a_state == BUF_LOADED)
        w_output_sel = 1'b0; // A ready first
    else if (r_buf_b_state == BUF_LOADED)
        w_output_sel = 1'b1; // B ready
    else
        w_output_sel = 1'b0; // Default
end
```

### 7.3.3 Load Controller

```
// Determine which buffer can accept new data
logic w_load_sel; // 0 = buffer A, 1 = buffer B

always_comb begin
    if (r_buf_a_state == BUF_IDLE)
        w_load_sel = 1'b0;
    else if (r_buf_b_state == BUF_IDLE)
        w_load_sel = 1'b1;
    else
        w_load_sel = 1'b0; // No buffer available
end

assign s_ready = (r_buf_a_state == BUF_IDLE) || (r_buf_b_state ==
BUF_IDLE);
```

### 7.3.4 Buffer State Transitions

```
// Buffer A state machine
always_ff @(posedge clk or negedge rst_n) begin
    if (!rst_n) begin
        r_buf_a_state <= BUF_IDLE;
    end else begin
        case (r_buf_a_state)
            BUF_IDLE: begin
                if (s_valid && s_ready && !w_load_sel) begin
                    r_buf_a_data <= s_data;
                    r_buf_a_state <= BUF_LOADED;
                end
            end
        end
    end
```

```

    BUF_LOADED: begin
        if (!w_output_sel || r_buf_b_state != BUF_OUTPUTTING)
            r_buf_a_state <= BUF_OUTPUTTING;
        end

    BUF_OUTPUTTING: begin
        if (m_ready && r_buf_a_count == RATIO - 1)
            r_buf_a_state <= BUF_IDLE;
        end
    endcase
end
end

// Buffer B follows same pattern with opposite selection

```

## 7.4 2.4.4 Timing Analysis

### 7.4.1 Continuous Operation

With dual buffering, the output is continuous:

*Table 2.10: Dual-Buffer Cycle Timing*

Cycle	Buffer A	Buffer B	Output
0	Load W0	Idle	-
1	Output W0[0]	Load W1	W0[0]
2	Output W0[1]	Wait	W0[1]
...	...	...	...
8	Output W0[7]	Wait	W0[7]
9	Load W2	Output W1[0]	W1[0]
10	Wait	Output W1[1]	W1[1]

### 7.4.2 Latency

*Table 2.11: Dual-Buffer Latency*

Metric	Value
First beat latency	1 cycle (load)
Sustained throughput	100%
Buffer switch latency	0 cycles

## 7.5 2.4.5 Edge Cases

---

### 7.5.1 Empty Pipeline Start

When starting from empty state: 1. First wide beat loads into buffer A 2. Buffer A starts outputting 3. Buffer B loads next wide beat 4. Continuous operation begins

### 7.5.2 Pipeline Drain

When input stops (last beat processed): 1. Currently outputting buffer completes 2. Other buffer (if loaded) takes over 3. Pipeline drains to empty

### 7.5.3 Backpressure Handling

If `m_ready` goes low: 1. Output stalls on current beat 2. Loading continues if buffer available 3. If both buffers full, `s_ready` goes low 4. Resumes when `m_ready` returns high

## 7.6 2.4.6 Resource Comparison

---

### 7.6.1 Resource Breakdown

*Table 2.12: Resource Breakdown*

Component	Single Buffer	Dual Buffer
Data registers	512 bits	1024 bits
Sideband registers	64 bits	128 bits
Beat counters	3 bits	6 bits
State machines	1	2 + arbiter
Control logic	~40 LEs	~90 LEs

### 7.6.2 Trade-off Summary

*Table 2.13: Mode Trade-offs*

Mode	Area	Throughput	Use Case
Single	1x	80-90%	Area-constrained, bursty traffic
Dual	2x	100%	Performance-critical, sustained traffic

## 7.7 2.4.7 Configuration Guidelines

---

### 7.7.1 When to Use Dual Buffer

**Recommended:** - DDR/HBM memory interfaces - DMA engines with sustained transfers - Video/audio streaming paths - Any path where 10-20% throughput loss is unacceptable

**Not Recommended:** - Area-constrained designs - Bursty traffic with gaps - Control paths (low bandwidth) - When latency is more critical than throughput

### 7.7.2 Integration Example

```
// High-performance read path for DDR controller
axi_data_dsize #(
    .WIDE_WIDTH(512),
    .NARROW_WIDTH(64),
    .DUAL_BUFFER(1),           // Enable dual buffer
    .USE_BURST_TRACKER(1),
    .SB_BROADCAST(1)
) u_rdata_dsize (
    // ... connections
);

// Low-bandwidth control path (save area)
axi_data_dsize #(
    .WIDE_WIDTH(128),
    .NARROW_WIDTH(32),
    .DUAL_BUFFER(0),          // Single buffer OK
    .USE_BURST_TRACKER(0),
    .SB_BROADCAST(1)
) u_ctrl_dsize (
    // ... connections
);
```

---

Next: [axi4\\_dwidth\\_converter\\_wr](#)

## 8 2.5 axi4\_dwidth\_converter\_wr

The **axi4\_dwidth\_converter\_wr** module provides complete AXI4 write path conversion, handling AW, W, and B channels with burst length adjustment and proper protocol compliance.

### 8.1 2.5.1 Purpose and Function

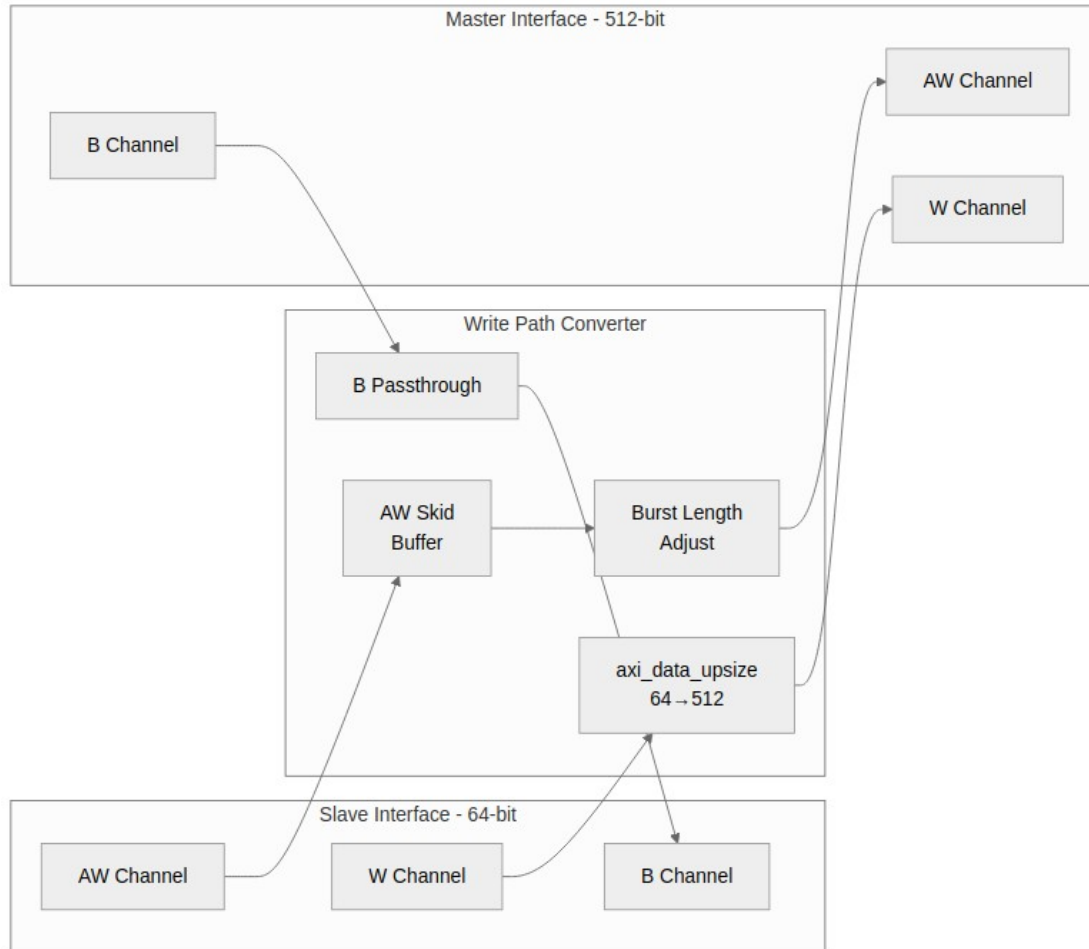
---

The write converter combines the generic `axi_data_upsize` with AXI4 protocol handling:

1. **Address Channel (AW):** Passes through with burst length adjustment
2. **Write Data Channel (W):** Uses axi\_data\_upsize for data packing
3. **Response Channel (B):** Passes through unchanged
4. **Burst Length Adjustment:** Converts AWLEN based on width ratio

## 8.2 2.5.2 Block Diagram

8.2.1 Figure 2.6: Write Converter Architecture



### Write Converter Architecture

## 8.3 2.5.3 Interface Specification

### 8.3.1 Parameters

Table 2.14: Write Converter Parameters

Parameter	Type	Default	Description
S_DATA_WIDTH	int	64	Slave-side (narrow) data width
M_DATA_WIDTH	int	512	Master-side (wide) data width
ADDR_WIDTH	int	64	Address width
ID_WIDTH	int	4	Transaction ID width
SKID_DEPTH	int	2	Pipeline buffer depth

### 8.3.2 Ports

```
module axi4_dwidth_converter_wr #(
    parameter int S_DATA_WIDTH = 64,
    parameter int M_DATA_WIDTH = 512,
    parameter int ADDR_WIDTH   = 64,
    parameter int ID_WIDTH     = 4,
    parameter int SKID_DEPTH   = 2
) (
    input  logic clk,
    input  logic rst_n,

    // Slave interface (narrow, from master)
    input  logic          s_awvalid,
    output logic          s_awready,
    input  logic [ADDR_WIDTH-1:0] s_awaddr,
    input  logic [7:0] s_awlen,
    input  logic [2:0] s_awsz,
    input  logic [1:0] s_awburst,
    input  logic [ID_WIDTH-1:0] s_awid,
    // ... other AW signals

    input  logic          s_wvalid,
    output logic          s_wready,
    input  logic [S_DATA_WIDTH-1:0] s_wdata,
```

```

    input logic [S_DATA_WIDTH/8-1:0] s_wstrb,
    input logic                                s_wlast,

    output logic                                s_bvalid,
    input logic                                s_bready,
    output logic [ID_WIDTH-1:0]                s_bid,
    output logic [1:0]                          s_bresp,

    // Master interface (wide, to slave)
    output logic                                m_awvalid,
    input logic                                m_awready,
    output logic [ADDR_WIDTH-1:0]              m_awaddr,
    output logic [7:0]                          m_awlen,
    output logic [2:0]                          m_awsz,
    output logic [1:0]                          m_awburst,
    output logic [ID_WIDTH-1:0]                m_awid,
    // ... other AW signals

    output logic                                m_wvalid,
    input logic                                m_wready,
    output logic [M_DATA_WIDTH-1:0]            m_wdata,
    output logic [M_DATA_WIDTH/8-1:0]          m_wstrb,
    output logic                                m_wlast,

    input logic                                m_bvalid,
    output logic                                m_bready,
    input logic [ID_WIDTH-1:0]                m_bid,
    input logic [1:0]                          m_bresp
);

```

## 8.4 2.5.4 Burst Length Conversion

### 8.4.1 Ratio Calculation

```

localparam int RATIO = M_DATA_WIDTH / S_DATA_WIDTH;
localparam int RATIO_LOG2 = $clog2(RATIO);

```

```

// New AWLEN = (original AWLEN + 1) / RATIO - 1
// = (AWLEN + 1) >> RATIO_LOG2 - 1

```

### 8.4.2 Examples

Table 2.15: Burst Length Conversion Examples

S_DATA	M_DATA	Ratio	S_AWLE N	S_beats	M_AWLE N	M_beats
64	512	8	7	8	0	1

S_DATA	M_DATA	Ratio	S_AWLE		M_AWLE	
			N	S_beats	N	M_beats
64	512	8	15	16	1	2
64	256	4	3	4	0	1
64	256	4	7	8	1	2

### 8.4.3 Non-Aligned Bursts

When burst length is not a multiple of ratio:

S\_AWLEN = 5 (6 beats), RATIO = 8

M\_AWLEN = 0 (1 beat)

The 6 narrow beats pack into 1 wide beat.

Remaining 2 positions have WSTRB = 0 (no write).

## 8.5 2.5.5 Address Channel Handling

### 8.5.1 AW Passthrough with Adjustment

```
// Burst length adjustment
logic [7:0] w_adjusted_awlen;
assign w_adjusted_awlen = ((s_awlen + 1) >> RATIO_LOG2) - 1;

// Size adjustment (wider data = larger size)
logic [2:0] w_adjusted_awsz;
assign w_adjusted_awsz = s_awsz + RATIO_LOG2;

// Address alignment check
logic w_aligned;
assign w_aligned = (s_awaddr[RATIO_LOG2+2:0] == '0);
```

### 8.5.2 Skid Buffer for AW

```
axi_skid_buffer #(
    .DATA_WIDTH(AW_CHANNEL_WIDTH)
) u_aw_skid (
    .clk      (clk),
    .rst_n    (rst_n),
    .s_valid  (s_awvalid),
    .s_ready  (s_awready),
    .s_data   ({s_awid, s_awaddr, w_adjusted_awlen, ...}),
    .m_valid  (m_awvalid),
    .m_ready  (m_awready),
    .m_data   ({m_awid, m_awaddr, m_awlen, ...})
);
```



## 8.6 2.5.6 Write Data Channel

---

### 8.6.1 Upsize Instance

```
axi_data_upsize #(
    .NARROW_WIDTH(S_DATA_WIDTH),
    .WIDE_WIDTH(M_DATA_WIDTH),
    .NARROW_SB_WIDTH(S_DATA_WIDTH/8), // WSTRB
    .WIDE_SB_WIDTH(M_DATA_WIDTH/8),
    .SB_OR_MODE(0), // Concatenate WSTRB
    .USE_LAST(1)
) u_wdata_upsize (
    .clk      (clk),
    .rst_n    (rst_n),
    .s_valid  (s_wvalid),
    .s_ready  (s_wready),
    .s_data   (s_wdata),
    .s_sideband (s_wstrb),
    .s_last   (s_wlast),
    .m_valid  (m_wvalid),
    .m_ready  (m_wready),
    .m_data   (m_wdata),
    .m_sideband (m_wstrb),
    .m_last   (m_wlast)
);
```

## 8.7 2.5.7 Response Channel

---

### 8.7.1 B Channel Passthrough

The response channel passes through unchanged:

```
// Simple passthrough (or via skid buffer)
assign s_bvalid = m_bvalid;
assign m_bready = s_bready;
assign s_bid    = m_bid;
assign s_bresp  = m_bresp;
```

### 8.7.2 Response Ordering

Responses return in order because: - Single outstanding transaction per ID - Upsize doesn't reorder data - B response generated after all W beats accepted

## 8.8 2.5.8 AW/W Synchronization

---

### 8.8.1 Challenge

AXI4 allows AW to arrive before, with, or after W data. The converter must handle all cases:

1. **AW before W:** Normal pipelining
2. **W before AW:** Data buffered until AW arrives
3. **Interleaved:** Multiple transactions in flight

### 8.8.2 Solution

Use FIFO for AW information needed by upsize logic:

```
// FIFO stores AWLEN for burst tracking
fifo_sync #(.WIDTH(8), .DEPTH(4)) u_aw_info_fifo (
    .clk      (clk),
    .rst_n    (rst_n),
    .wr_en    (s_awvalid && s_awready),
    .wr_data  (s_awlen),
    .rd_en    (m_wvalid && m_wready && m_wlast),
    .rd_data  (current_awlen),
    .full     (aw_fifo_full),
    .empty    (aw_fifo_empty)
);
```

## 8.9 2.5.9 Resource Utilization

### 8.9.1 Typical Resources (64→512, ID=4, ADDR=64)

AW skid buffer: ~200 flip-flops  
W upsize: ~600 flip-flops, ~60 LUTs  
B skid buffer: ~20 flip-flops  
AW info FIFO: ~50 flip-flops  
Control logic: ~100 LUTs

Total: ~870 flip-flops, ~160 LUTs

## 8.10 2.5.10 Timing Characteristics

### 8.10.1 Latency

Table 2.16: Write Converter Latency

Path	Latency
AW passthrough	1-2 cycles (skid)
W upsize	N cycles (accumulation)
B passthrough	1 cycle (skid)

### 8.10.2 Throughput

- AW channel: 1 transaction/cycle

- W channel: 100% (upsized is 100%)
- B channel: 1 response/cycle

## 8.11 2.5.11 Usage Example

---

```
axi4_dwidth_converter_wr #(
    .S_DATA_WIDTH(64),
    .M_DATA_WIDTH(512),
    .ADDR_WIDTH(64),
    .ID_WIDTH(4),
    .SKID_DEPTH(2)
) u_wr_converter (
    .clk      (aclk),
    .rst_n    (aresetn),

    // 64-bit slave interface (from CPU)
    .s_awvalid (cpu_awvalid),
    .s_awready (cpu_awready),
    .s_awaddr  (cpu_awaddr),
    .s_awlen   (cpu_awlen),
    // ... other s_* signals

    // 512-bit master interface (to DDR)
    .m_awvalid (ddr_awvalid),
    .m_awready (ddr_awready),
    .m_awaddr  (ddr_awaddr),
    .m_awlen   (ddr_awlen),
    // ... other m_* signals
);
```

---

Next: [axi4\\_dwidth\\_converter\\_rd](#)

## 9 2.6 axi4\_dwidth\_converter\_rd

The **axi4\_dwidth\_converter\_rd** module provides complete AXI4 read path conversion, handling AR and R channels with burst length adjustment and burst-aware RLAST generation.

### 9.1 2.6.1 Purpose and Function

---

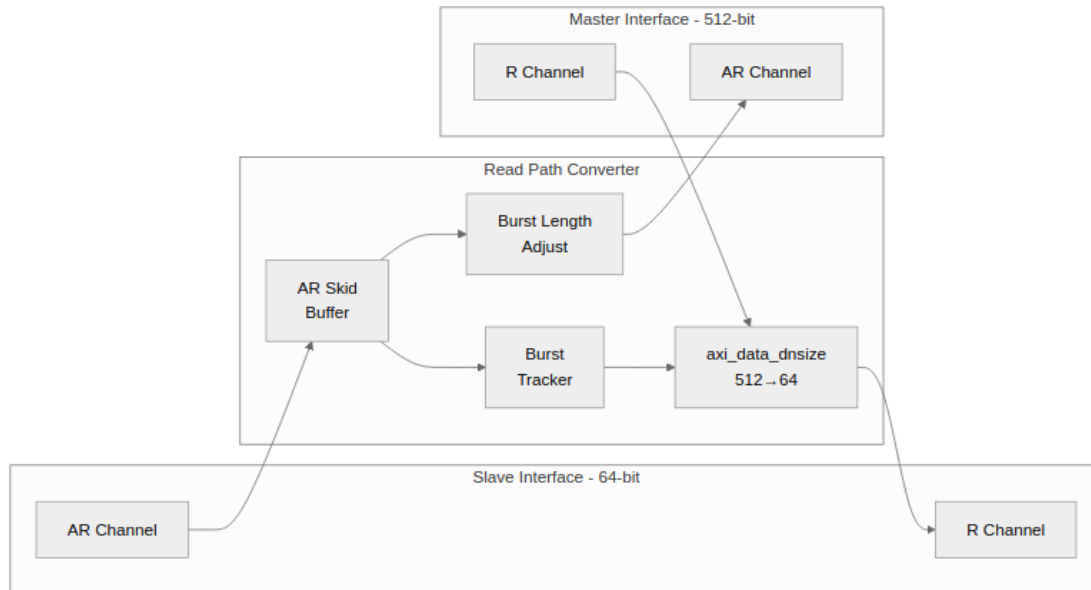
The read converter combines the generic `axi_data_dsize` with AXI4 protocol handling:

1. **Address Channel (AR):** Passes through with burst length adjustment
2. **Read Data Channel (R):** Uses `axi_data_dsize` for data splitting

3. **Burst Tracking:** Generates correct RLAST based on original ARLEN
4. **Response Broadcasting:** Propagates RRESP to all narrow beats

## 9.2 2.6.2 Block Diagram

### 9.2.1 Figure 2.7: Read Converter Architecture



*Read Converter Architecture*

## 9.3 2.6.3 Interface Specification

### 9.3.1 Parameters

*Table 2.17: Read Converter Parameters*

Parameter	Type	Default	Description
S_DATA_WIDTH	int	64	Slave-side (narrow) data width
M_DATA_WIDTH	int	512	Master-side (wide) data width
ADDR_WIDTH	int	64	Address width
ID_WIDTH	int	4	Transaction ID width

Parameter	Type	Default	Description
DUAL_BUFFER	bit	1	Enable dual-buffer for 100% throughput
SKID_DEPTH	int	2	Pipeline buffer depth

### 9.3.2 Ports

```

module axi4_dwidth_converter_rd #(
    parameter int S_DATA_WIDTH = 64,
    parameter int M_DATA_WIDTH = 512,
    parameter int ADDR_WIDTH   = 64,
    parameter int ID_WIDTH    = 4,
    parameter bit DUAL_BUFFER = 1,
    parameter int SKID_DEPTH  = 2
) (
    input   logic clk,
    input   logic rst_n,

    // Slave interface (narrow, to master)
    input   logic s_arvalid,
    output  logic s_arready,
    input   logic [ADDR_WIDTH-1:0] s_araddr,
    input   logic [7:0] s_arlen,
    input   logic [2:0] s_arsize,
    input   logic [1:0] s_arburst,
    input   logic [ID_WIDTH-1:0] s_arid,

    output  logic s_rvalid,
    input   logic s_rready,
    output  logic [S_DATA_WIDTH-1:0] s_rdata,
    output  logic [ID_WIDTH-1:0] s_rid,
    output  logic [1:0] s_rresp,
    output  logic s_rlast,

    // Master interface (wide, from slave)
    output  logic m_arvalid,
    input   logic m_arready,
    output  logic [ADDR_WIDTH-1:0] m_araddr,
    output  logic [7:0] m_arlen,
    output  logic [2:0] m_arsize,
    output  logic [1:0] m_arburst,
    output  logic [ID_WIDTH-1:0] m_arid,

```

```

    input  logic                                     m_rvalid,
    output logic                                     m_rready,
    input  logic [M_DATA_WIDTH-1:0]                 m_rdata,
    input  logic [ID_WIDTH-1:0]                     m_rid,
    input  logic [1:0]                             m_rresp,
    input  logic                                     m_rlast
);

```

## 9.4 2.6.4 Burst Length Conversion

### 9.4.1 Ratio Calculation

Same as write converter:

```

localparam int RATIO = M_DATA_WIDTH / S_DATA_WIDTH;
localparam int RATIO_LOG2 = $clog2(RATIO);

```

```

// New ARLEN = (original ARLEN + 1) / RATIO - 1

```

### 9.4.2 Examples

Table 2.18: Read Burst Length Conversion

S_DATA	M_DATA	Ratio	S_ARLE	S_beats	M_ARLE	M_beats
			N		N	
64	512	8	7	8	0	1
64	512	8	15	16	1	2
64	512	8	31	32	3	4

## 9.5 2.6.5 Address Channel Handling

### 9.5.1 AR Passthrough with Adjustment

```

// Burst length adjustment
logic [7:0] w_adjusted_arlen;
assign w_adjusted_arlen = ((s_arlen + 1) >> RATIO_LOG2) - 1;

```

```

// Size adjustment
logic [2:0] w_adjusted_arsize;
assign w_adjusted_arsize = s_arsize + RATIO_LOG2;

```

### 9.5.2 AR Information FIFO

Store original ARLEN for RLAST generation:

```
// FIFO to track original burst length
fifo_sync #(.WIDTH(8+ID_WIDTH), .DEPTH(4)) u_ar_info_fifo (
    .clk      (clk),
    .rst_n    (rst_n),
    .wr_en     (s_arvalid && s_arready),
    .wr_data   ({s_arid, s_arlen}),
    .rd_en     (s_rvalid && s_rready && s_rlast),
    .rd_data   ({current_arid, current_arlen}),
    .full      (ar_fifo_full),
    .empty     (ar_fifo_empty)
);
```

## 9.6 2.6.6 Read Data Channel

---

### 9.6.1 Downsize Instance

```
axi_data_dsize #(
    .WIDE_WIDTH(M_DATA_WIDTH),
    .NARROW_WIDTH(S_DATA_WIDTH),
    .WIDE_SB_WIDTH(2),           // RRESP
    .NARROW_SB_WIDTH(2),
    .SB_BROADCAST(1),           // Broadcast RRESP
    .DUAL_BUFFER(DUAL_BUFFER),
    .USE_BURST_TRACKER(1),
    .BURST_LEN_WIDTH(8)
) u_rdata_dsize (
    .clk      (clk),
    .rst_n    (rst_n),
    .s_valid   (m_rvalid),
    .s_ready   (m_rready),
    .s_data    (m_rdata),
    .s_sideband (m_rresp),
    .s_last    (m_rlast),
    .burst_len (current_arlen),
    .m_valid    (s_rvalid),
    .m_ready    (s_rready),
    .m_data     (s_rdata),
    .m_sideband (s_rresp),
    .m_last     (s_rlast)
);
```

## 9.7 2.6.7 RLAST Generation

---

### 9.7.1 Challenge

The wide interface generates RLAST based on M\_ARLEN, but the narrow interface needs RLAST based on S\_ARLEN:

Wide RLAST: Asserted on beat (M\_ARLEN + 1)  
Narrow RLAST: Asserted on beat (S\_ARLEN + 1) = (M\_ARLEN + 1) \* RATIO

### 9.7.2 Solution: Burst Tracker

```
// Track narrow beats within burst
logic [15:0] r_beat_count;
logic [7:0] r_total_narrow_beats;

always_ff @(posedge clk or negedge rst_n) begin
    if (!rst_n) begin
        r_beat_count <= '0;
        r_total_narrow_beats <= '0;
    end else begin
        if (new_burst_start) begin
            r_beat_count <= '0;
            r_total_narrow_beats <= (current_arlen + 1) * RATIO - 1;
        end else if (s_rvalid && s_rready) begin
            r_beat_count <= r_beat_count + 1;
        end
    end
end

assign s_rlast = (r_beat_count == r_total_narrow_beats);
```

## 9.8 2.6.8 RID Handling

### 9.8.1 ID Passthrough

RID passes through unchanged:

```
// RID from wide interface propagates to all narrow beats
assign s_rid = m_rid;

// Or use tracked ID from FIFO
assign s_rid = current_arid;
```

## 9.9 2.6.9 Dual-Buffer Impact

### 9.9.1 Performance Comparison

Table 2.19: Read Converter Performance

Mode	Throughput	Latency	Resources
Single	80-90%	1 cycle	1x
Dual	100%	1 cycle	2x



### 9.9.2 When to Use Dual Buffer

Use **dual buffer** for: - DDR read paths with high bandwidth - Streaming read applications - DMA read operations

Use **single buffer** for: - Control register reads - Low-bandwidth paths - Area-constrained designs

## 9.10 2.6.10 Resource Utilization

---

### 9.10.1 Typical Resources (512→64, ID=4, Dual Buffer)

AR skid buffer: ~150 flip-flops  
R downsize (dual): ~1200 flip-flops, ~100 LUTs  
AR info FIFO: ~100 flip-flops  
Burst tracker: ~30 flip-flops, ~20 LUTs  
Control logic: ~80 LUTs

Total: ~1480 flip-flops, ~200 LUTs

### 9.10.2 Single Buffer Version

R downsize (single): ~600 flip-flops, ~50 LUTs

Total: ~880 flip-flops, ~120 LUTs

## 9.11 2.6.11 Timing Characteristics

---

### 9.11.1 Latency

*Table 2.20: Read Converter Latency*

Path	Latency
AR passthrough	1-2 cycles (skid)
First R beat	1 cycle (load buffer)
Subsequent R beats	1 beat/cycle

### 9.11.2 Throughput

- AR channel: 1 transaction/cycle
- R channel: 80% (single) or 100% (dual)

## 9.12 2.6.12 Usage Example

---

```
axi4_dwidth_converter_rd #(  
    .S_DATA_WIDTH(64),  
    .M_DATA_WIDTH(512),  
    .ADDR_WIDTH(64),
```

```

        .ID_WIDTH(4),
        .DUAL_BUFFER(1),          // High-performance mode
        .SKID_DEPTH(2)
    ) u_rd_converter (
        .clk      (aclk),
        .rst_n    (aresetn),

        // 64-bit slave interface (to CPU)
        .s_arvalid (cpu_arvalid),
        .s_arready (cpu_arready),
        .s_araddr  (cpu_araddr),
        .s_arlen   (cpu_arlen),
        .s_rvalid  (cpu_rvalid),
        .s_rready  (cpu_rready),
        .s_rdata   (cpu_rdata),
        // ... other s_* signals

        // 512-bit master interface (from DDR)
        .m_arvalid (ddr_arvalid),
        .m_arready (ddr_arready),
        .m_araddr  (ddr_araddr),
        .m_arlen   (ddr_arlen),
        .m_rvalid  (ddr_rvalid),
        .m_rready  (ddr_rready),
        .m_rdata   (ddr_rdata),
        // ... other m_* signals
    );

```

---

Next: [Chapter 3: Protocol Converter Blocks](#)

## 10 3.1 Protocol Conversion Overview

Protocol converters enable communication between components using different communication protocols, essential for integrating diverse IP blocks in complex SoC designs.

### 10.1 3.1.1 Available Converters

#### 10.1.1 AXI4 to AXI4-Lite (Protocol Downgrade)

Table 3.1: AXI4 to AXI4-Lite Converters

Module	Function	Test Status
axi4_to_axil4_rd	Read burst decomposition	14/14 passing

Module	Function	Test Status
axi4_to_axil4_wr	Write burst decomposition	14/14 passing
axi4_to_axil4	Full bidirectional wrapper	Composed

### 10.1.2 AXI4-Lite to AXI4 (Protocol Upgrade)

*Table 3.2: AXI4-Lite to AXI4 Converters*

Module	Function	Test Status
axil4_to_axi4_rd	Read protocol upgrade	7/7 passing
axil4_to_axi4_wr	Write protocol upgrade	7/7 passing
axil4_to_axi4	Full bidirectional wrapper	Composed

### 10.1.3 Other Protocol Converters

*Table 3.3: Other Protocol Converters*

Module	Function	Status
axi4_to_apb_convert	Full AXI4-to-APB bridge	Production
peakrdl_to_cmdrsp	Register interface adapter	Production
uart_axil_bridge	UART to AXI4-Lite	Planned

## 10.2 3.1.2 Protocol Comparison

### 10.2.1 Figure 3.1: Protocol Feature Comparison

Protocol Comparison
<i>Protocol Comparison</i>

### 10.2.2 Feature Matrix

*Table 3.4: Protocol Feature Comparison*

Feature	AXI4	AXI4-Lite	APB
Channels	5 (AW, W, B,	5 (simplified)	1 (combined)

Feature	AXI4	AXI4-Lite	APB
	AR, R)		
Bursts	Up to 256 beats	Single beat only	Single beat
Out-of-order	Yes (ID-based)	No	No
Pipelining	Yes	Optional	No (2-phase)
Data widths	8-1024 bits	32/64 bits	8-32 bits

### 10.3 3.1.3 Conversion Strategies

#### 10.3.1 AXI4 to AXI4-Lite

**Challenge:** Decompose multi-beat bursts into sequential single beats.

**Strategy:** 1. Accept AXI4 burst transaction 2. Issue N single-beat AXI4-Lite transactions 3. Aggregate responses 4. Return combined response to AXI4 master

**Complexity:** Medium (FSM-based decomposition)

#### 10.3.2 AXI4-Lite to AXI4

**Challenge:** Add AXI4 burst signals with appropriate defaults.

**Strategy:** 1. Pass through all AXI4-Lite signals 2. Add default values for burst signals (LEN=0, SIZE=2, BURST=INCR) 3. Add default IDs (configurable)

**Complexity:** Very low (combinational only)

#### 10.3.3 AXI4 to APB

**Challenge:** Bridge 5-channel AXI4 to 2-phase APB protocol.

**Strategy:** 1. Accept AXI4 transaction 2. Execute APB setup phase 3. Execute APB access phase 4. Return AXI4 response

**Complexity:** High (full protocol FSM)

### 10.4 3.1.4 Performance Characteristics

*Table 3.5: Protocol Converter Performance*

Converter	Single-Beat	Burst (N)	Area
axi4_to_axil4	0 cycles	2N cycles	~450 LUTs
axil4_to_axi4	0 cycles	N/A	~110 LUTs



### 10.6.3 Pattern 3: Mixed System

CPU (AXI4) → AXI4 Crossbar → DDR (AXI4)  
→ AXI4-to-AXIL4 → Config Regs (AXIL4)  
→ AXI4-to-APB → UART, GPIO (APB)

---

Next: [AXI4 to AXI4-Lite](#)

## 11 3.2 AXI4 to AXI4-Lite Converter

The **axi4\_to\_axil4** converter family decomposes AXI4 burst transactions into sequential AXI4-Lite single-beat transactions.

### 11.1 3.2.1 Module Organization

```
axi4_to_axil4.sv      # Full bidirectional wrapper
├── axi4_to_axil4_rd.sv # Read path converter
└── axi4_to_axil4_wr.sv # Write path converter
```

#### 11.1.1 Design Philosophy

Separate read and write paths enable: - Independent optimization - Selective instantiation (read-only, write-only, or both) - Simpler verification (test paths independently)

### 11.2 3.2.2 Read Path (axi4\_to\_axil4\_rd)

#### 11.2.1 Block Diagram

#### 11.2.2 Figure 3.2: AXI4 to AXI4-Lite Read Path

AXI4 to AXIL4 Read

*AXI4 to AXIL4 Read*

#### 11.2.3 Operation

**Single-Beat (ARLEN=0):**

Cycle 0: AR accepted (passthrough)  
Cycle 1: R returned (passthrough)  
Total: 0 extra cycles (pure passthrough)

**Multi-Beat (ARLEN=N-1):**

Cycle 0: AR[0] issued to AXIL4  
Cycle 1: R[0] received, AR[1] issued  
Cycle 2: R[1] received, AR[2] issued

```

...
Cycle 2N-1: R[N-1] received (RLAST)
Total: 2N cycles (1 cycle per AR + 1 cycle per R)

```

#### 11.2.4 State Machine

```

typedef enum logic [1:0] {
    IDLE      = 2'b00,  // Wait for AR
    DECOMPOSE = 2'b01,  // Issuing single beats
    WAIT_R    = 2'b10   // Wait for last R
} rd_state_t;

```

#### 11.2.5 Implementation

```

// Beat counter and address tracking
logic [7:0] r_beat_count;
logic [7:0] r_arlen_saved;
logic [ADDR_WIDTH-1:0] r_current_addr;

// State machine
always_ff @(posedge clk or negedge rst_n) begin
    if (!rst_n) begin
        r_state <= IDLE;
        r_beat_count <= '0;
    end else begin
        case (r_state)
            IDLE: begin
                if (s_arvalid && s_arready) begin
                    r_arlen_saved <= s_arlen;
                    r_current_addr <= s_araddr;
                    if (s_arlen == 0) begin
                        // Single beat - passthrough
                        r_state <= WAIT_R;
                    end else begin
                        r_state <= DECOMPOSE;
                        r_beat_count <= 8'd1;
                    end
                end
            end
        end

        DECOMPOSE: begin
            if (m_arvalid && m_arready) begin
                r_current_addr <= r_current_addr + (1 <<
s_arsize);
                r_beat_count <= r_beat_count + 1;
                if (r_beat_count == r_arlen_saved) begin
                    r_state <= WAIT_R;
                end
            end
        end
    end

```

```

        end

        WAIT_R: begin
            if (s_rvalid && s_rready && s_rlast) begin
                r_state <= IDLE;
            end
        end
    end
endcase
end
end
end

// AXIL4 AR generation
assign m_arvalid = (r_state == DECOMPOSE) || (r_state == IDLE &&
s_arvalid);
assign m_araddr = r_current_addr;

// R aggregation
assign s_rlast = (r_beat_count == r_arlen_saved) || (r_arlen_saved ==
0);

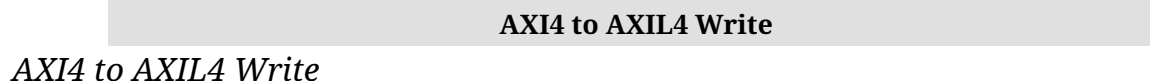
```

### 11.3 3.2.3 Write Path (axi4\_to\_axil4\_wr)

---

#### 11.3.1 Block Diagram

#### 11.3.2 Figure 3.3: AXI4 to AXI4-Lite Write Path



#### 11.3.3 Operation

##### Single-Beat (AWLEN=0):

Cycle 0: AW+W accepted (passthrough)  
 Cycle 1: B returned (passthrough)  
 Total: 0 extra cycles

##### Multi-Beat (AWLEN=N-1):

Cycle 0: AW[0] + W[0] issued  
 Cycle 1: B[0] received, AW[1] + W[1] issued  
 ...  
 Cycle 2N-1: B[N-1] received  
 Total: 2N cycles

#### 11.3.4 AW/W Synchronization Challenge

AXI4 allows AW and W to arrive in any order: - AW before W - W before AW - Interleaved



### 11.3.5 Solution: Dual Accept Logic

```
// Track AW and W arrival independently
logic r_aw_accepted;
logic r_w_accepted;

// Accept both when ready to issue
always_ff @(posedge clk) begin
    if (w_issue_axil4) begin
        r_aw_accepted <= 1'b0;
        r_w_accepted <= 1'b0;
    end else begin
        if (s_awvalid && s_awready)
            r_aw_accepted <= 1'b1;
        if (s_wvalid && s_wready)
            r_w_accepted <= 1'b1;
    end
end

// Issue AXIL4 when both available
assign w_issue_axil4 = (r_aw_accepted || s_awvalid) &&
                        (r_w_accepted || s_wvalid) &&
                        m_ready;
```

### 11.3.6 Response Aggregation

```
// Track worst response in burst
logic [1:0] r_worst_bresp;

always_ff @(posedge clk) begin
    if (r_state == IDLE)
        r_worst_bresp <= 2'b00; // OKAY
    else if (m_bvalid && m_bready)
        r_worst_bresp <= (m_bresp > r_worst_bresp) ? m_bresp :
r_worst_bresp;
end

// Return worst response on final beat
assign s_bresp = (r_beat_count == r_awlen_saved) ?
                  ((m_bresp > r_worst_bresp) ? m_bresp : r_worst_bresp)
:
                  r_worst_bresp;
```

## 11.4 3.2.4 Bidirectional Wrapper (axi4\_to\_axil4)

---

### 11.4.1 Composition Pattern

```
module axi4_to_axil4 #(
    parameter int DATA_WIDTH = 32,
    parameter int ADDR_WIDTH = 32,
    parameter int ID_WIDTH    = 4
) (
    // ... ports
);

    // Instantiate read path
    axi4_to_axil4_rd #(
        .DATA_WIDTH(DATA_WIDTH),
        .ADDR_WIDTH(ADDR_WIDTH),
        .ID_WIDTH(ID_WIDTH)
    ) u_rd (
        // ... read channel connections
    );

    // Instantiate write path
    axi4_to_axil4_wr #(
        .DATA_WIDTH(DATA_WIDTH),
        .ADDR_WIDTH(ADDR_WIDTH),
        .ID_WIDTH(ID_WIDTH)
    ) u_wr (
        // ... write channel connections
    );

endmodule
```

## 11.5 3.2.5 Resource Utilization

---

Table 3.6: AXI4 to AXIL4 Resources

Module	Registers	LUTs	BRAM
axi4_to_axil4_r d	~120	~180	0
axi4_to_axil4_ wr	~150	~220	0
axi4_to_axil4 (combined)	~270	~400	0

## 11.6 3.2.6 Performance Analysis

---

### 11.6.1 Throughput

*Table 3.7: AXI4 to AXIL4 Throughput*

Transaction Type	Throughput
Single-beat	100% (passthrough)
2-beat burst	50%
4-beat burst	50%
N-beat burst	~50%

### 11.6.2 Latency

*Table 3.8: AXI4 to AXIL4 Latency*

Transaction Type	Latency
Single-beat	0 extra cycles
N-beat burst	2N - 1 cycles

## 11.7 3.2.7 Test Coverage

---

**Test Suite:** 42 tests passing

*Table 3.9: Test Coverage Summary*

Test Category	Tests	Status
Single-beat read	4	Pass
Multi-beat read	6	Pass
Single-beat write	4	Pass
Multi-beat write	6	Pass
Mixed traffic	8	Pass
Error injection	6	Pass
Edge cases	8	Pass

**Next:** [AXI4-Lite to AXI4](#)

## 12 3.3 AXI4-Lite to AXI4 Converter

The `axil4_to_axi4` converter family upgrades AXI4-Lite single-beat transactions to full AXI4 protocol by adding default burst signals.

### 12.1 3.3.1 Module Organization

```
axil4_to_axi4.sv      # Full bidirectional wrapper
├─ axil4_to_axi4_rd.sv # Read path converter
└─ axil4_to_axi4_wr.sv # Write path converter
```

### 12.2 3.3.2 Design Philosophy

**Zero-Overhead Upgrade:** - Purely combinational logic - No state machines or buffers - Adds default values for missing AXI4 signals

**Why This Works:** - AXI4-Lite is a subset of AXI4 - All AXI4-Lite transactions are single-beat - Missing AXI4 signals have well-defined defaults

### 12.3 3.3.3 Signal Mapping

#### 12.3.1 Address Channel Signals

Table 3.10: AR Channel Mapping

AXI4-Lite Signal	AXI4 Signal	Default/Mapping
ARADDR	ARADDR	Passthrough
ARPROT	ARPROT	Passthrough
ARVALID	ARVALID	Passthrough
ARREADY	ARREADY	Passthrough
-	ARLEN	8'h00 (single beat)
-	ARSIZE	$\$clog2(DATA\_WIDT$ H/8)
-	ARBURST	2'b01 (INCR)
-	ARLOCK	1'b0 (normal)
-	ARCACHE	4'b0000 (non- cacheable)
-	ARQOS	4'b0000 (no QoS)
-	ARID	Configurable default

## 12.3.2 Data Channel Signals

Table 3.11: R Channel Mapping

AXI4-Lite Signal	AXI4 Signal	Default/Mapping
RDATA	RDATA	Passthrough
RRESP	RRESP	Passthrough
RVALID	RVALID	Passthrough
RREADY	RREADY	Passthrough
-	RLAST	1'b1 (always last)
-	RID	Matches ARID

## 12.4 3.3.4 Read Path (axil4\_to\_axi4\_rd)

### 12.4.1 Block Diagram

### 12.4.2 Figure 3.4: AXI4-Lite to AXI4 Read Path

#### AXIL4 to AXI4 Read

*AXIL4 to AXI4 Read*

### 12.4.3 Implementation

```
module axil4_to_axi4_rd #(
    parameter int DATA_WIDTH = 32,
    parameter int ADDR_WIDTH = 32,
    parameter int ID_WIDTH = 4,
    parameter logic [ID_WIDTH-1:0] DEFAULT_ARID = '0
) (
    // AXI4-Lite slave interface (input)
    input logic s_arvalid,
    output logic s_arready,
    input logic [ADDR_WIDTH-1:0] s_araddr,
    input logic [2:0] s_arprot,

    output logic s_rvalid,
    input logic s_rready,
    output logic [DATA_WIDTH-1:0] s_rdata,
    output logic [1:0] s_rresp,

    // AXI4 master interface (output)
    output logic m_arvalid,
    input logic m_arready,
    output logic [ADDR_WIDTH-1:0] m_araddr,
    output logic [7:0] m_arlen,
```

```

output logic [2:0]          m_arsize,
output logic [1:0]          m_arburst,
output logic                m_arlock,
output logic [3:0]          m_arcache,
output logic [2:0]          m_arprot,
output logic [3:0]          m_arqos,
output logic [ID_WIDTH-1:0] m_arid,

input  logic                m_rvalid,
output logic                m_rready,
input  logic [DATA_WIDTH-1:0] m_rdata,
input  logic [1:0]          m_rresp,
input  logic                m_rlast,
input  logic [ID_WIDTH-1:0] m_rid
);

// AR channel - passthrough with defaults
assign m_arvalid = s_arvalid;
assign s_arready = m_arready;
assign m_araddr  = s_araddr;
assign m_arprot  = s_arprot;

// AXI4 burst defaults
assign m_arlen   = 8'h00;           // Single beat
assign m_arsize  = $clog2(DATA_WIDTH/8); // Full width
assign m_arburst = 2'b01;           // INCR
assign m_arlock  = 1'b0;           // Normal access
assign m_arcache = 4'b0000;        // Non-cacheable
assign m_arqos   = 4'b0000;        // No QoS
assign m_arid    = DEFAULT_ARID;    // Configurable ID

// R channel - passthrough (ignore RLAST and RID)
assign s_rvalid  = m_rvalid;
assign m_rready  = s_rready;
assign s_rdata   = m_rdata;
assign s_rresp   = m_rresp;

endmodule

```

**Key Points:** - No registers - purely combinational - RLAST from AXI4 is ignored (always 1 for AXIL4) - RID from AXI4 is ignored (no ID tracking in AXIL4)

## 12.5 3.3.5 Write Path (axil4\_to\_axi4\_wr)

### 12.5.1 Block Diagram

### 12.5.2 Figure 3.5: AXI4-Lite to AXI4 Write Path

#### AXIL4 to AXI4 Write

*AXIL4 to AXI4 Write*

### 12.5.3 Implementation

```
module axil4_to_axi4_wr #(
    parameter int DATA_WIDTH = 32,
    parameter int ADDR_WIDTH = 32,
    parameter int ID_WIDTH = 4,
    parameter logic [ID_WIDTH-1:0] DEFAULT_AWID = '0
) (
    // AXI4-Lite slave interface (input)
    input logic s_awvalid,
    output logic s_awready,
    input logic [ADDR_WIDTH-1:0] s_awaddr,
    input logic [2:0] s_awprot,

    input logic s_wvalid,
    output logic s_wready,
    input logic [DATA_WIDTH-1:0] s_wdata,
    input logic [DATA_WIDTH/8-1:0] s_wstrb,

    output logic s_bvalid,
    input logic s_bready,
    output logic [1:0] s_bresp,

    // AXI4 master interface (output)
    output logic m_awvalid,
    input logic m_awready,
    output logic [ADDR_WIDTH-1:0] m_awaddr,
    output logic [7:0] m_awlen,
    output logic [2:0] m_awsz,
    output logic [1:0] m_awburst,
    output logic m_awlock,
    output logic [3:0] m_awcache,
    output logic [2:0] m_awprot,
    output logic [3:0] m_awqos,
    output logic [ID_WIDTH-1:0] m_awid,

    output logic m_wvalid,
    input logic m_wready,
```

```

    output logic [DATA_WIDTH-1:0] m_wdata,
    output logic [DATA_WIDTH/8-1:0] m_wstrb,
    output logic m_wlast,

    input logic m_bvalid,
    output logic m_bready,
    input logic [1:0] m_bresp,
    input logic [ID_WIDTH-1:0] m_bid
);

// AW channel
assign m_awvalid = s_awvalid;
assign s_awready = m_awready;
assign m_awaddr = s_awaddr;
assign m_awprot = s_awprot;
assign m_awlen = 8'h00;
assign m_awsz = $clog2(DATA_WIDTH/8);
assign m_awburst = 2'b01;
assign m_awlock = 1'b0;
assign m_awcache = 4'b0000;
assign m_awqos = 4'b0000;
assign m_awid = DEFAULT_AWID;

// W channel
assign m_wvalid = s_wvalid;
assign s_wready = m_wready;
assign m_wdata = s_wdata;
assign m_wstrb = s_wstrb;
assign m_wlast = 1'b1; // Always last (single beat)

// B channel
assign s_bvalid = m_bvalid;
assign m_bready = s_bready;
assign s_bresp = m_bresp;

endmodule

```

## 12.6 3.3.6 Bidirectional Wrapper

---

```

module axi4_to_axi4 #(
    parameter int DATA_WIDTH = 32,
    parameter int ADDR_WIDTH = 32,
    parameter int ID_WIDTH = 4,
    parameter logic [ID_WIDTH-1:0] DEFAULT_ID = '0
) (
    // ... all port declarations
);

```



```

axil4_to_axi4_rd #(
    .DATA_WIDTH(DATA_WIDTH),
    .ADDR_WIDTH(ADDR_WIDTH),
    .ID_WIDTH(ID_WIDTH),
    .DEFAULT_ARID(DEFAULT_ID)
) u_rd (/* connections */);

axil4_to_axi4_wr #(
    .DATA_WIDTH(DATA_WIDTH),
    .ADDR_WIDTH(ADDR_WIDTH),
    .ID_WIDTH(ID_WIDTH),
    .DEFAULT_AWID(DEFAULT_ID)
) u_wr (/* connections */);

```

**endmodule**

## 12.7 3.3.7 Resource Utilization

*Table 3.12: AXIL4 to AXI4 Resources*

Module	Registers	LUTs
axil4_to_axi4_rd	0	~50
axil4_to_axi4_wr	0	~60
axil4_to_axi4 (combined)	0	~110

**Note:** Zero registers - purely combinational logic.

## 12.8 3.3.8 Performance

*Table 3.13: AXIL4 to AXI4 Performance*

Metric	Value
Latency	0 cycles
Throughput	100%
Max frequency	Wire speed

## 12.9 3.3.9 Test Coverage

**Test Suite:** 14 tests passing

Table 3.14: Test Coverage Summary

Test Category	Tests	Status
Single-beat read	3	Pass
Single-beat write	3	Pass
Mixed traffic	4	Pass
Default ID verification	2	Pass
Edge cases	2	Pass

## 12.10 3.3.10 Usage Example

```
// Upgrade simple register block to AXI4 fabric
axil4_to_axi4 #(
    .DATA_WIDTH(32),
    .ADDR_WIDTH(32),
    .ID_WIDTH(4),
    .DEFAULT_ID(4'h5) // Unique ID for this IP
) u_protocol_upgrade (
    // Connect AXIL4 register block
    .s_arvalid (reg_block_arvalid),
    .s_arready (reg_block_arready),
    .s_araddr (reg_block_araddr),
    // ... other AXIL4 signals

    // Connect to AXI4 crossbar
    .m_arvalid (xbar_arvalid),
    .m_arready (xbar_arready),
    .m_araddr (xbar_araddr),
    .m_arlen (xbar_arlen),
    // ... other AXI4 signals
);
```

Next: [AXI4 to APB](#)

## 13 3.4 AXI4 to APB Converter

The **axi4\_to\_apb\_convert** module provides full protocol translation from AXI4 to APB, enabling AXI4 masters to access APB peripherals.

## 13.1 3.4.1 Purpose

---

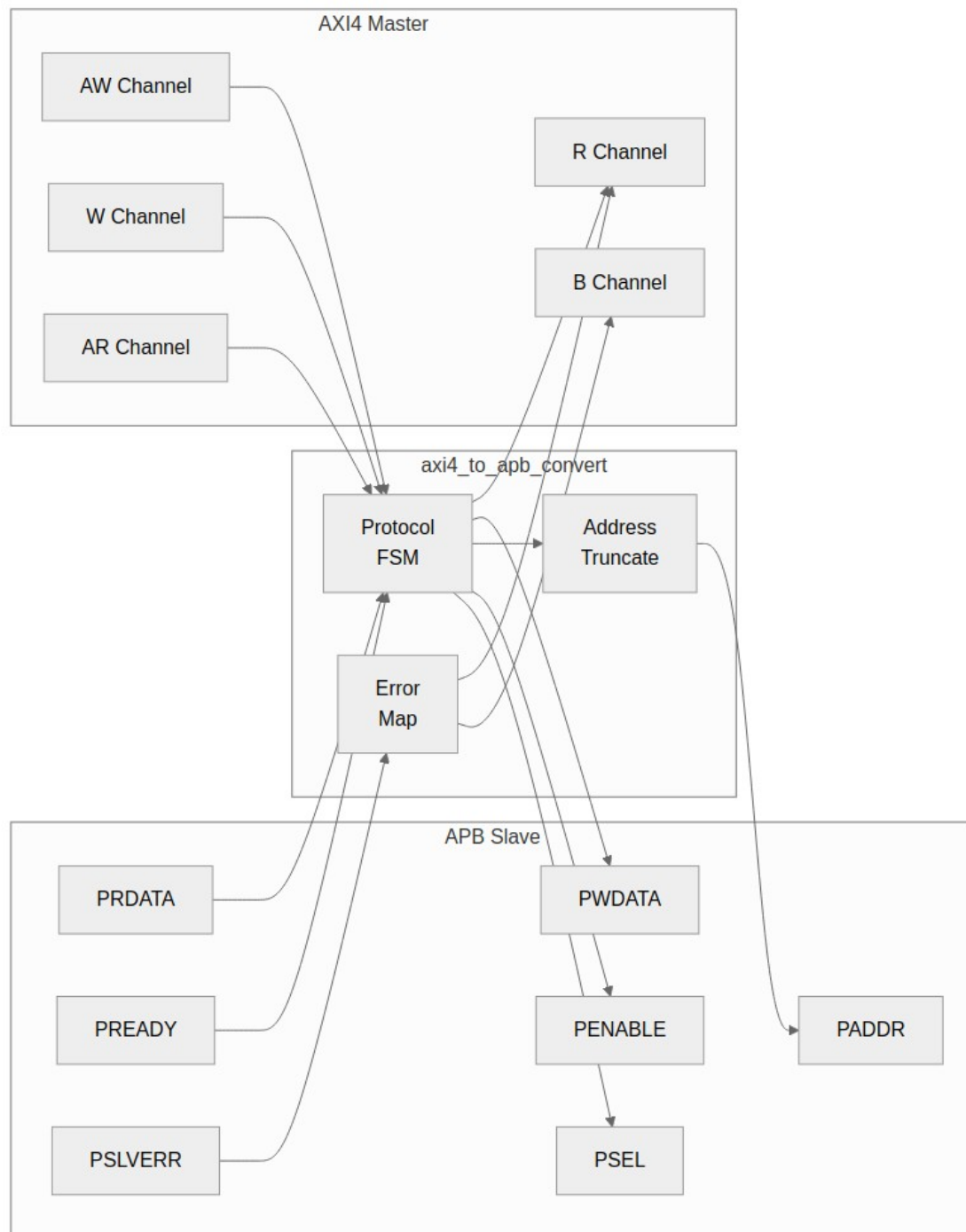
Bridge the significant protocol differences between AXI4 and APB:

*Table 3.15: AXI4 vs APB Comparison*

Aspect	AXI4	APB
Channels	5 (AW, W, B, AR, R)	1 (combined)
Phases	Pipelined	2-phase (setup, access)
Bursts	Up to 256 beats	Single transfer
Address width	Up to 64 bits	Typically 32 bits
Data width	8-1024 bits	8-32 bits

## 13.2 3.4.2 Block Diagram

13.2.1 Figure 3.6: AXI4 to APB Converter



*AXI4 to APB*

## 13.3 3.4.3 Interface Specification

### 13.3.1 Parameters

Table 3.16: AXI4 to APB Parameters

Parameter	Type	Default	Description
AXI_ADDR_WI DTH	int	64	AXI4 address width
AXI_DATA_WI DTH	int	32	AXI4 data width
AXI_ID_WIDTH	int	4	AXI4 ID width
APB_ADDR_WI DTH	int	32	APB address width
APB_DATA_WI DTH	int	32	APB data width

### 13.3.2 Ports

```
module axi4_to_apb_convert #(
    parameter int AXI_ADDR_WIDTH = 64,
    parameter int AXI_DATA_WIDTH = 32,
    parameter int AXI_ID_WIDTH   = 4,
    parameter int APB_ADDR_WIDTH = 32,
    parameter int APB_DATA_WIDTH = 32
) (
    input  logic clk,
    input  logic rst_n,

    // AXI4 slave interface
    // AW channel
    input  logic          s_awvalid,
    output logic          s_awready,
    input  logic [AXI_ADDR_WIDTH-1:0] s_awaddr,
    input  logic [7:0] s_awlen,
    input  logic [AXI_ID_WIDTH-1:0] s_awid,

    // W channel
    input  logic          s_wvalid,
    output logic          s_wready,
    input  logic [AXI_DATA_WIDTH-1:0] s_wdata,
    input  logic [AXI_DATA_WIDTH/8-1:0] s_wstrb,
    input  logic          s_wlast,
```

```

// B channel
output logic                                s_bvalid,
input  logic                                s_bready,
output logic [AXI_ID_WIDTH-1:0]            s_bid,
output logic [1:0]                          s_bresp,

// AR channel
input  logic                                s_arvalid,
output logic                                s_arready,
input  logic [AXI_ADDR_WIDTH-1:0]          s_araddr,
input  logic [7:0]                          s_arlen,
input  logic [AXI_ID_WIDTH-1:0]            s_arid,

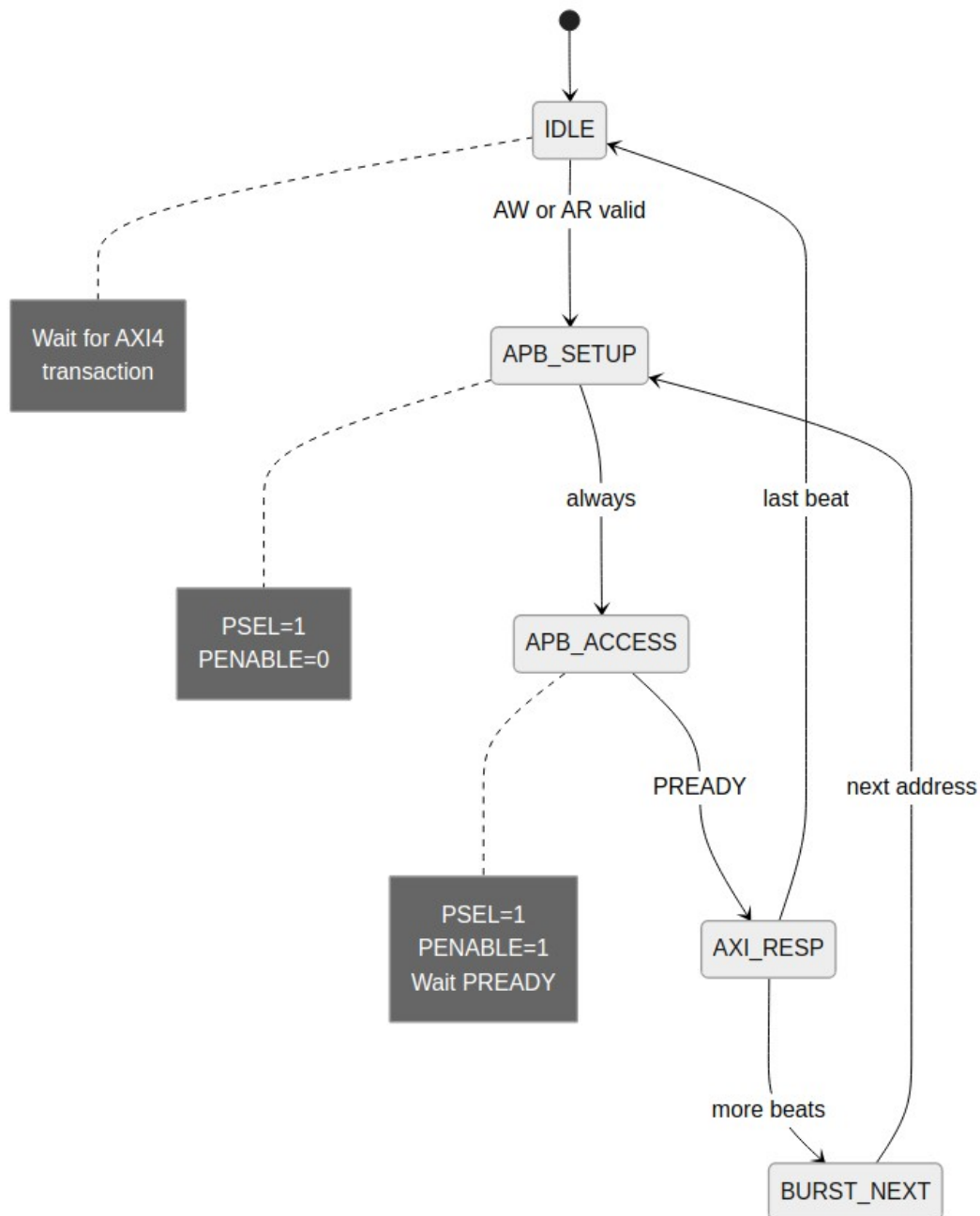
// R channel
output logic                                s_rvalid,
input  logic                                s_rready,
output logic [AXI_DATA_WIDTH-1:0]          s_rdata,
output logic [AXI_ID_WIDTH-1:0]            s_rid,
output logic [1:0]                          s_rresp,
output logic                                s_rlast,

// APB master interface
output logic                                psel,
output logic                                penable,
output logic                                pwrite,
output logic [APB_ADDR_WIDTH-1:0]          paddr,
output logic [APB_DATA_WIDTH-1:0]          pdata,
output logic [APB_DATA_WIDTH/8-1:0]        pstrb,
input  logic                                pready,
input  logic [APB_DATA_WIDTH-1:0]          prdata,
input  logic                                pslverr
);

```

## 13.4 3.4.4 State Machine

13.4.1 Figure 3.7: AXI4 to APB FSM



*AXI4 to APB FSM*

### 13.4.2 States

```
typedef enum logic [2:0] {  
    IDLE          = 3'b000, // Wait for AXI4 transaction  
    APB_SETUP     = 3'b001, // APB setup phase  
    APB_ACCESS    = 3'b010, // APB access phase (wait PREADY)  
    AXI_RESP_B    = 3'b011, // Send AXI4 B response  
    AXI_RESP_R    = 3'b100, // Send AXI4 R response  
    BURST_NEXT    = 3'b101  // Next beat in burst  
} apb_state_t;
```

### 13.4.3 Transitions

Table 3.17: FSM Transitions

Current State	Condition	Next State
IDLE	s_awvalid	APB_SETUP (write)
IDLE	s_arvalid	APB_SETUP (read)
APB_SETUP	always	APB_ACCESS
APB_ACCESS	pready && is_write	AXI_RESP_B
APB_ACCESS	pready && is_read	AXI_RESP_R
AXI_RESP_B	s_bready && ! more_beats	IDLE
AXI_RESP_B	s_bready && more_beats	BURST_NEXT
AXI_RESP_R	s_rready && ! more_beats	IDLE
AXI_RESP_R	s_rready && more_beats	BURST_NEXT
BURST_NEXT	always	APB_SETUP

## 13.5 3.4.5 Burst Handling

### 13.5.1 Burst Decomposition

AXI4 bursts are decomposed into sequential APB transfers:

AXI4: AWADDR=0x1000, AWLEN=3 (4 beats)

APB sequence:

Transfer 0: PADDR=0x1000

Transfer 1: PADDR=0x1004



Transfer 2: PADDR=0x1008  
 Transfer 3: PADDR=0x100C

### 13.5.2 Address Calculation

```
// Calculate next address for INCR burst
logic [APB_ADDR_WIDTH-1:0] r_current_addr;
logic [2:0] r_awsz;

always_ff @(posedge clk) begin
    if (r_state == APB_ACCESS && pready) begin
        r_current_addr <= r_current_addr + (1 << r_awsz);
    end
end
```

## 13.6 3.4.6 Address Width Adaptation

### 13.6.1 64-bit to 32-bit Conversion

```
// Truncate upper address bits
assign paddr = s_awaddr[APB_ADDR_WIDTH-1:0];

// Optional: Check for out-of-range access
wire w_addr_oor = |s_awaddr[AXI_ADDR_WIDTH-1:APB_ADDR_WIDTH];
```

## 13.7 3.4.7 Error Response Mapping

Table 3.18: Error Mapping

APB Signal	AXI4 Response
PSLVERR = 0	OKAY (2'b00)
PSLVERR = 1	SLVERR (2'b10)

### 13.7.1 Error Aggregation

```
// Track worst error in burst
logic r_error_seen;

always_ff @(posedge clk) begin
    if (r_state == IDLE)
        r_error_seen <= 1'b0;
    else if (r_state == APB_ACCESS && pready && pslverr)
        r_error_seen <= 1'b1;
end

// Final response
assign s_bresp = r_error_seen ? 2'b10 : 2'b00;
```

## 13.8 3.4.8 Implementation

---

### 13.8.1 Core FSM

```
always_ff @(posedge clk or negedge rst_n) begin
    if (!rst_n) begin
        r_state <= IDLE;
        r_is_write <= 1'b0;
    end else begin
        case (r_state)
            IDLE: begin
                if (s_awvalid && s_wvalid) begin
                    r_state <= APB_SETUP;
                    r_is_write <= 1'b1;
                    r_current_addr <= s_awaddr[APB_ADDR_WIDTH-1:0];
                    r_beat_count <= '0;
                    r_awlen <= s_awlen;
                    r_awid <= s_awid;
                end else if (s_arvalid) begin
                    r_state <= APB_SETUP;
                    r_is_write <= 1'b0;
                    r_current_addr <= s_araddr[APB_ADDR_WIDTH-1:0];
                    r_beat_count <= '0;
                    r_arlen <= s_arlen;
                    r_arid <= s_arid;
                end
            end
        end

        APB_SETUP: begin
            r_state <= APB_ACCESS;
        end

        APB_ACCESS: begin
            if (pready) begin
                r_rdata_saved <= prdata;
                r_error_seen <= r_error_seen || pslverr;
                if (r_is_write)
                    r_state <= AXI_RESP_B;
                else
                    r_state <= AXI_RESP_R;
            end
        end

        AXI_RESP_B: begin
            if (s_bready) begin
                if (r_beat_count == r_awlen)
                    r_state <= IDLE;
                else

```

```

        r_state <= BURST_NEXT;
    end
end

AXI_RESP_R: begin
    if (s_rready) begin
        if (r_beat_count == r_arlen)
            r_state <= IDLE;
        else
            r_state <= BURST_NEXT;
        end
    end
end

BURST_NEXT: begin
    r_beat_count <= r_beat_count + 1;
    r_current_addr <= r_current_addr + (1 << r_size);
    r_state <= APB_SETUP;
end
endcase
end
end

```

### 13.9 3.4.9 Resource Utilization

State machine:	~50 LUTs, ~20 regs
Address logic:	~30 LUTs, ~40 regs
Data buffering:	~10 LUTs, ~70 regs
Control:	~60 LUTs, ~20 regs

Total: ~150 LUTs, ~150 regs

### 13.10 3.4.10 Performance

#### 13.10.1 Timing Analysis

*Table 3.19: APB Converter Timing*

Operation	Cycles
Single write	3-4 (setup + access + B)
Single read	3-4 (setup + access + R)
N-beat write burst	3N + 1
N-beat read burst	3N + 1

### 13.10.2 Throughput

**Best case:** 1 transfer per 3 cycles **With slow PREADY:** Additional cycles per transfer

## 13.11 3.4.11 Usage Example

---

```
axi4_to_apb_convert #(
    .AXI_ADDR_WIDTH(64),
    .AXI_DATA_WIDTH(32),
    .AXI_ID_WIDTH(4),
    .APB_ADDR_WIDTH(32),
    .APB_DATA_WIDTH(32)
) u_axi2apb (
    .clk      (aclk),
    .rst_n    (aresetn),

    // AXI4 slave (from CPU)
    .s_awvalid (cpu_awvalid),
    .s_awready (cpu_awready),
    // ... other AXI4 signals

    // APB master (to peripherals)
    .psel      (uart_psel),
    .penable   (uart_penable),
    .pwrite    (uart_pwrite),
    .paddr     (uart_paddr),
    .pwrdata   (uart_pwrdata),
    .pready    (uart_pready),
    .prdata    (uart_prdata),
    .pslverr   (uart_pslverr)
);
```

---

Next: [PeakRDL Adapter](#)

## 14 3.5 PeakRDL Adapter

The `peakrdl_to_cmdrsp` module adapts PeakRDL-generated register interfaces to a custom command/response protocol, enabling protocol decoupling and flexible register implementations.

### 14.1 3.5.1 Purpose

---

PeakRDL generates register blocks with an APB-style interface. This adapter:

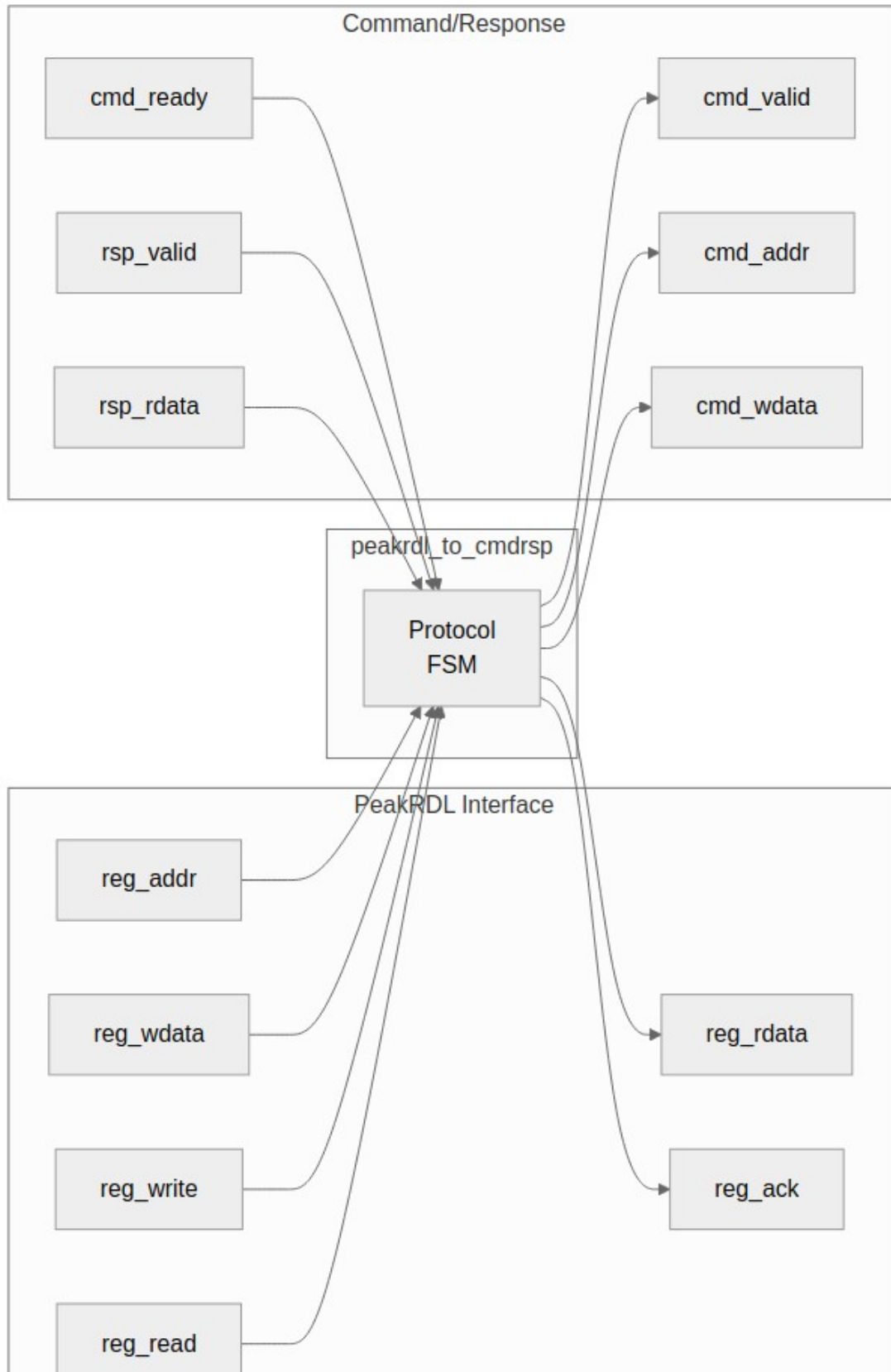
1. Decouples register interface from implementation

2. Provides clean handshake protocol
3. Enables pipelined register access
4. Supports custom control logic integration

## 14.2 3.5.2 Block Diagram

---

### 14.2.1 Figure 3.8: PeakRDL Adapter



## 14.3 3.5.3 Interface Specification

### 14.3.1 Parameters

Table 3.20: PeakRDL Adapter Parameters

Parameter	Type	Default	Description
ADDR_WIDTH	int	32	Address width
DATA_WIDTH	int	32	Data width

### 14.3.2 Ports

```

module peakrdl_to_cmdrsp #(
    parameter int ADDR_WIDTH = 32,
    parameter int DATA_WIDTH = 32
) (
    input logic clk,
    input logic rst_n,

    // Register interface (from PeakRDL)
    input logic [ADDR_WIDTH-1:0] reg_addr,
    input logic [DATA_WIDTH-1:0] reg_wdata,
    input logic reg_write,
    input logic reg_read,
    output logic [DATA_WIDTH-1:0] reg_rdata,
    output logic reg_error,
    output logic reg_ack,

    // Command interface (output)
    output logic cmd_valid,
    input logic cmd_ready,
    output logic [ADDR_WIDTH-1:0] cmd_addr,
    output logic [DATA_WIDTH-1:0] cmd_wdata,
    output logic cmd_write,

    // Response interface (input)
    input logic rsp_valid,
    output logic rsp_ready,
    input logic [DATA_WIDTH-1:0] rsp_rdata,
    input logic rsp_error
);

```



## 14.4 3.5.4 Operation

---

### 14.4.1 Write Transaction

Cycle 0: reg\_write asserted  
cmd\_valid = 1, cmd\_write = 1  
Cycle 1: cmd\_ready = 1 (downstream accepts)  
Wait for response  
Cycle N: rsp\_valid = 1  
reg\_ack = 1  
Cycle N+1: Transaction complete

### 14.4.2 Read Transaction

Cycle 0: reg\_read asserted  
cmd\_valid = 1, cmd\_write = 0  
Cycle 1: cmd\_ready = 1 (downstream accepts)  
Wait for response  
Cycle N: rsp\_valid = 1  
reg\_rdata = rsp\_rdata  
reg\_ack = 1  
Cycle N+1: Transaction complete

## 14.5 3.5.5 Implementation

---

```
// State machine
typedef enum logic [1:0] {
    IDLE    = 2'b00,
    CMD     = 2'b01,
    RSP     = 2'b10
} state_t;

state_t r_state;
logic r_is_write;

always_ff @(posedge clk or negedge rst_n) begin
    if (!rst_n) begin
        r_state <= IDLE;
    end else begin
        case (r_state)
            IDLE: begin
                if (reg_write || reg_read) begin
                    r_state <= CMD;
                    r_is_write <= reg_write;
                end
            end

            CMD: begin
```



### 14.7.22. Register Access Logging

PeakRDL Registers → Adapter → Logger → Registers  
↓  
Log Buffer

### 14.7.33. Pipeline Insertion

PeakRDL Registers → Adapter → Pipeline → Slow Registers  
(for timing)

## 14.8 3.5.8 Integration Example

---

```
// Instantiate PeakRDL-generated register block
my_regs u_regs (
    .clk      (clk),
    .rst_n    (rst_n),

    // APB-style interface from CPU
    .s_apb_psel    (apb_psel),
    .s_apb_penable (apb_penable),
    // ... other APB signals

    // Register interface to adapter
    .reg_addr    (reg_addr),
    .reg_wdata    (reg_wdata),
    .reg_write    (reg_write),
    .reg_read     (reg_read),
    .reg_rdata    (reg_rdata),
    .reg_error    (reg_error),
    .reg_ack      (reg_ack)
);

// Adapter to custom protocol
peakrdl_to_cmdrsp #(
    .ADDR_WIDTH(32),
    .DATA_WIDTH(32)
) u_adapter (
    .clk      (clk),
    .rst_n    (rst_n),

    // From PeakRDL registers
    .reg_addr    (reg_addr),
    .reg_wdata    (reg_wdata),
    .reg_write    (reg_write),
    .reg_read     (reg_read),
    .reg_rdata    (reg_rdata),
    .reg_error    (reg_error),
```

```
.reg_ack      (reg_ack),  
  
// To custom control logic  
.cmd_valid    (ctrl_cmd_valid),  
.cmd_ready    (ctrl_cmd_ready),  
.cmd_addr     (ctrl_cmd_addr),  
.cmd_wdata    (ctrl_cmd_wdata),  
.cmd_write    (ctrl_cmd_write),  
  
.rsp_valid    (ctrl_rsp_valid),  
.rsp_ready    (ctrl_rsp_ready),  
.rsp_rdata    (ctrl_rsp_rdata),  
.rsp_error    (ctrl_rsp_error)  
);
```

---

Next: [Chapter 4: FSM Design](#)

## 15 4.1 Width Converter FSMs

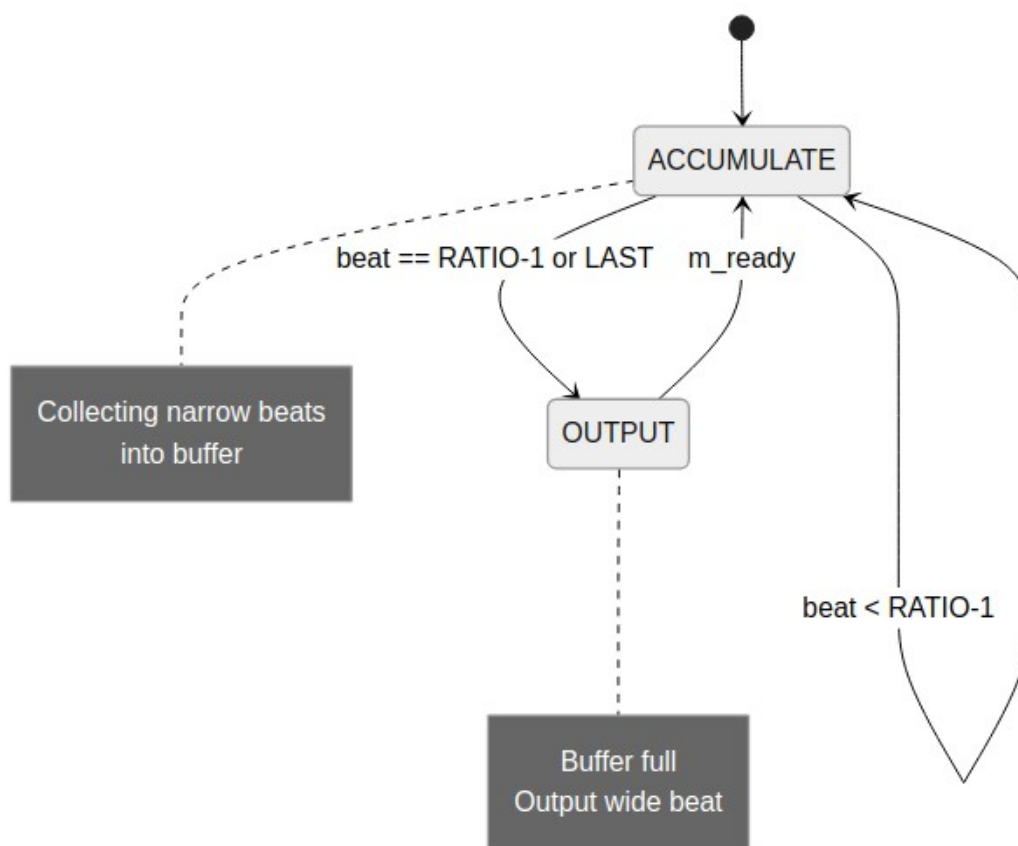
This section describes the state machines used in data width converter modules.

### 15.1 4.1.1 Upsize FSM

---

The **axi\_data\_upsize** module uses a simple accumulation state machine.

### 15.1.1 Figure 4.1: Upsize FSM



### *Upsize FSM*

#### 15.1.2 States

*Table 4.1: Upsize FSM States*

State	Description
ACCUMULATE	Collecting narrow beats into buffer
OUTPUT	Buffer full, outputting wide beat

#### 15.1.3 Transitions

##### ACCUMULATE:

- `s_valid && count < RATIO-1` → stay, increment count
- `s_valid && (count == RATIO-1 || s_last)` → OUTPUT

##### OUTPUT:

- m\_ready → ACCUMULATE, reset count
- !m\_ready → stay

#### 15.1.4 Implementation

```
typedef enum logic {
    ACCUMULATE = 1'b0,
    OUTPUT      = 1'b1
} upsize_state_t;

upsize_state_t r_state;

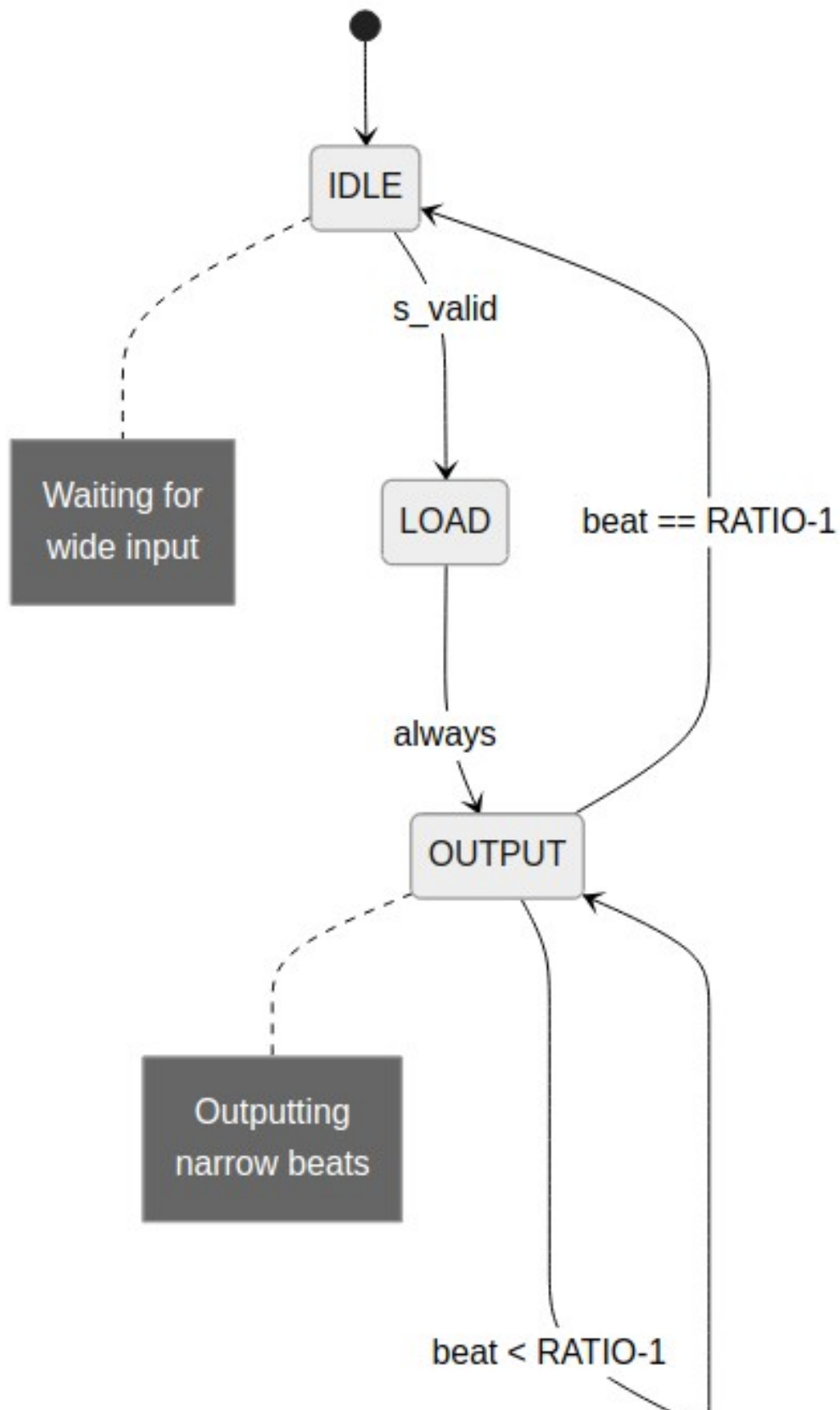
always_ff @(posedge clk or negedge rst_n) begin
    if (!rst_n) begin
        r_state <= ACCUMULATE;
    end else begin
        case (r_state)
            ACCUMULATE: begin
                if (s_valid && s_ready) begin
                    if (s_last || r_count == RATIO - 1)
                        r_state <= OUTPUT;
                end
            end
            OUTPUT: begin
                if (m_ready)
                    r_state <= ACCUMULATE;
            end
        endcase
    end
end
```

## 15.2 4.1.2 Downsize FSM (Single Buffer)

---

The **axi\_data\_dnsz** single-buffer mode uses a load/output state machine.

### 15.2.1 Figure 4.2: Downsize Single-Buffer FSM





## Downsize FSM

### 15.2.2 States

Table 4.2: Downsize FSM States

State	Description
IDLE	Waiting for wide input
LOAD	Loading wide beat
OUTPUT	Outputting narrow beats

### 15.2.3 Transitions

IDLE:

- s\_valid → LOAD

LOAD:

- always → OUTPUT (combinational)

OUTPUT:

- m\_ready && count < RATIO-1 → stay, increment count
- m\_ready && count == RATIO-1 → IDLE

## 15.3 4.1.3 Downsize FSM (Dual Buffer)

Dual-buffer mode uses two parallel state machines with an arbiter.

### 15.3.1 Buffer State Machine

```
typedef enum logic [1:0] {  
    BUF_EMPTY      = 2'b00,  
    BUF_LOADED     = 2'b01,  
    BUF_OUTPUTTING = 2'b10  
} buf_state_t;
```

```
buf_state_t r_buf_a_state, r_buf_b_state;
```

### 15.3.2 Arbiter Logic

```
// Select which buffer outputs  
always_comb begin  
    if (r_buf_a_state == BUF_OUTPUTTING)  
        output_sel = 1'b0; // Buffer A  
    else if (r_buf_b_state == BUF_OUTPUTTING)  
        output_sel = 1'b1; // Buffer B  
    else if (r_buf_a_state == BUF_LOADED)  
        output_sel = 1'b0; // A loaded first  
    else
```

```

        output_sel = 1'b1; // B loaded first
end

// Select which buffer loads
always_comb begin
    if (r_buf_a_state == BUF_EMPTY)
        load_sel = 1'b0;
    else
        load_sel = 1'b1;
end

```

## 15.4 4.1.4 Full Converter FSMs

---

### 15.4.1 Write Converter (axi4\_dwidth\_converter\_wr)

IDLE:

- AW valid → accept AW, store info
- W valid → buffer W data

AW\_ACCEPT:

- downstream AW ready → forward adjusted AW

W\_CONVERT:

- upsize accumulating narrow W beats
- on output → forward wide W beat

B\_FORWARD:

- B from downstream → forward to master

### 15.4.2 Read Converter (axi4\_dwidth\_converter\_rd)

IDLE:

- AR valid → accept AR, store info, forward adjusted AR

AR\_FORWARD:

- downstream AR ready → wait for R

R\_CONVERT:

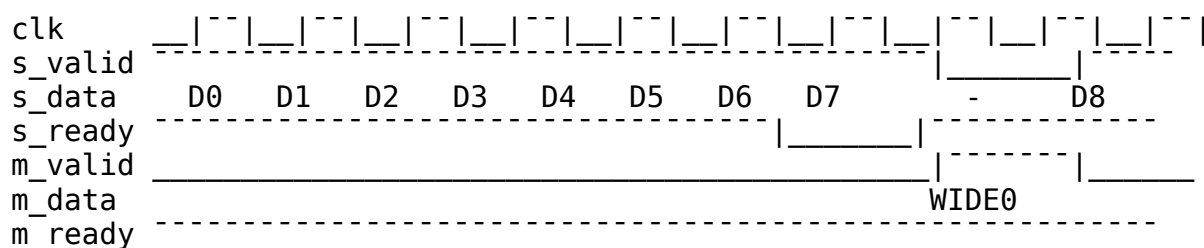
- downsize splitting wide R into narrow beats
- track burst count for RLAST

R\_FORWARD:

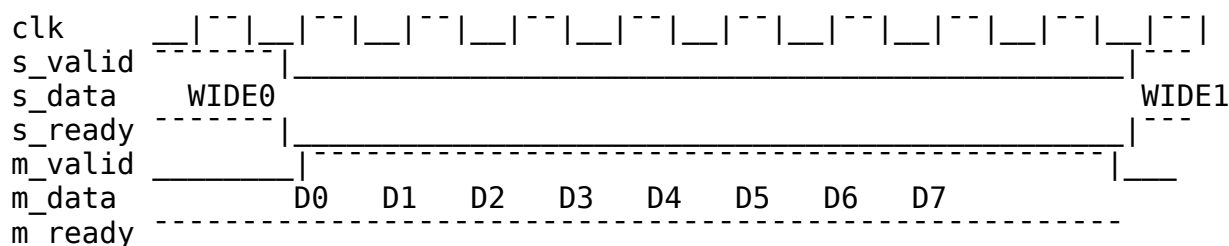
- narrow R beat ready → forward to master
- on RLAST → IDLE

## 15.5 4.1.5 Timing Diagrams

### 15.5.1 Upsize Timing (8:1 ratio)



### 15.5.2 Downsize Timing (8:1 ratio, single buffer)



Next: [Protocol Converter FSMs](#)

## 16 4.2 Protocol Converter FSMs

This section describes the state machines used in protocol converter modules.

### 16.1 4.2.1 AXI4 to AXI4-Lite Read FSM

#### 16.1.1 Figure 4.3: AXI4 to AXI4-Lite Read FSM

AXI4 to AXI4-Lite Read FSM

*AXI4 to AXI4-Lite Read FSM*

#### 16.1.2 States

Table 4.3: AXI4 to AXI4-Lite Read FSM States

State	Description
IDLE	Waiting for AR transaction
SINGLE	Single-beat passthrough
DECOMPOSE	Issuing burst as single beats
WAIT_R	Waiting for final R response

### 16.1.3 Transitions

IDLE:

- s\_arvalid && s\_arlen == 0 → SINGLE (passthrough)
- s\_arvalid && s\_arlen > 0 → DECOMPOSE

SINGLE:

- m\_arready → forward AR, wait for R
- m\_rvalid && m\_rready → IDLE

DECOMPOSE:

- m\_arready → issue single AR
- increment address, decrement remaining
- remaining == 0 → WAIT\_R

WAIT\_R:

- s\_rvalid && s\_rready && s\_rlast → IDLE

### 16.1.4 Implementation

```
typedef enum logic [1:0] {  
    IDLE      = 2'b00,  
    SINGLE    = 2'b01,  
    DECOMPOSE = 2'b10,  
    WAIT_R    = 2'b11  
} rd_state_t;  
  
rd_state_t r_state;  
logic [7:0] r_beats_remaining;  
logic [ADDR_WIDTH-1:0] r_current_addr;  
  
always_ff @(posedge clk or negedge rst_n) begin  
    if (!rst_n) begin  
        r_state <= IDLE;  
    end else begin  
        case (r_state)  
            IDLE: begin  
                if (s_arvalid && s_arready) begin  
                    r_current_addr <= s_araddr;  
                    r_beats_remaining <= s_arlen;  
                    r_state <= (s_arlen == 0) ? SINGLE : DECOMPOSE;  
                end  
            end  
  
            SINGLE: begin  
                if (m_rvalid && s_rready)  
                    r_state <= IDLE;  
            end  
        end  
    end  
end
```

```

        DECOMPOSE: begin
            if (m_arvalid && m_arready) begin
                r_current_addr <= r_current_addr + (1 <<
r_arsize);
                if (r_beats_remaining == 0)
                    r_state <= WAIT_R;
                else
                    r_beats_remaining <= r_beats_remaining - 1;
            end
        end

        WAIT_R: begin
            if (s_rvalid && s_rready && s_rlast)
                r_state <= IDLE;
            end
        endcase
    end
end

```

## 16.2 4.2.2 AXI4 to AXI4-Lite Write FSM

### 16.2.1 States

Table 4.4: AXI4 to AXI4-Lite Write FSM States

State	Description
IDLE	Waiting for AW/W transactions
SYNC_AW_W	Synchronizing AW and W channels
DECOMPOSE	Issuing burst as single beats
WAIT_B	Waiting for all B responses
SEND_B	Sending aggregated B response

### 16.2.2 AW/W Synchronization

// Track which channels have been accepted

**logic** r\_aw\_pending, r\_w\_pending;

**always\_ff** @(posedge clk) **begin**

// Accept AW

**if** (s\_awvalid && s\_awready && !r\_aw\_pending)  
 r\_aw\_pending <= 1'b1;

// Accept W

```

if (s_wvalid && s_wready && !r_w_pending)
    r_w_pending <= 1'b1;

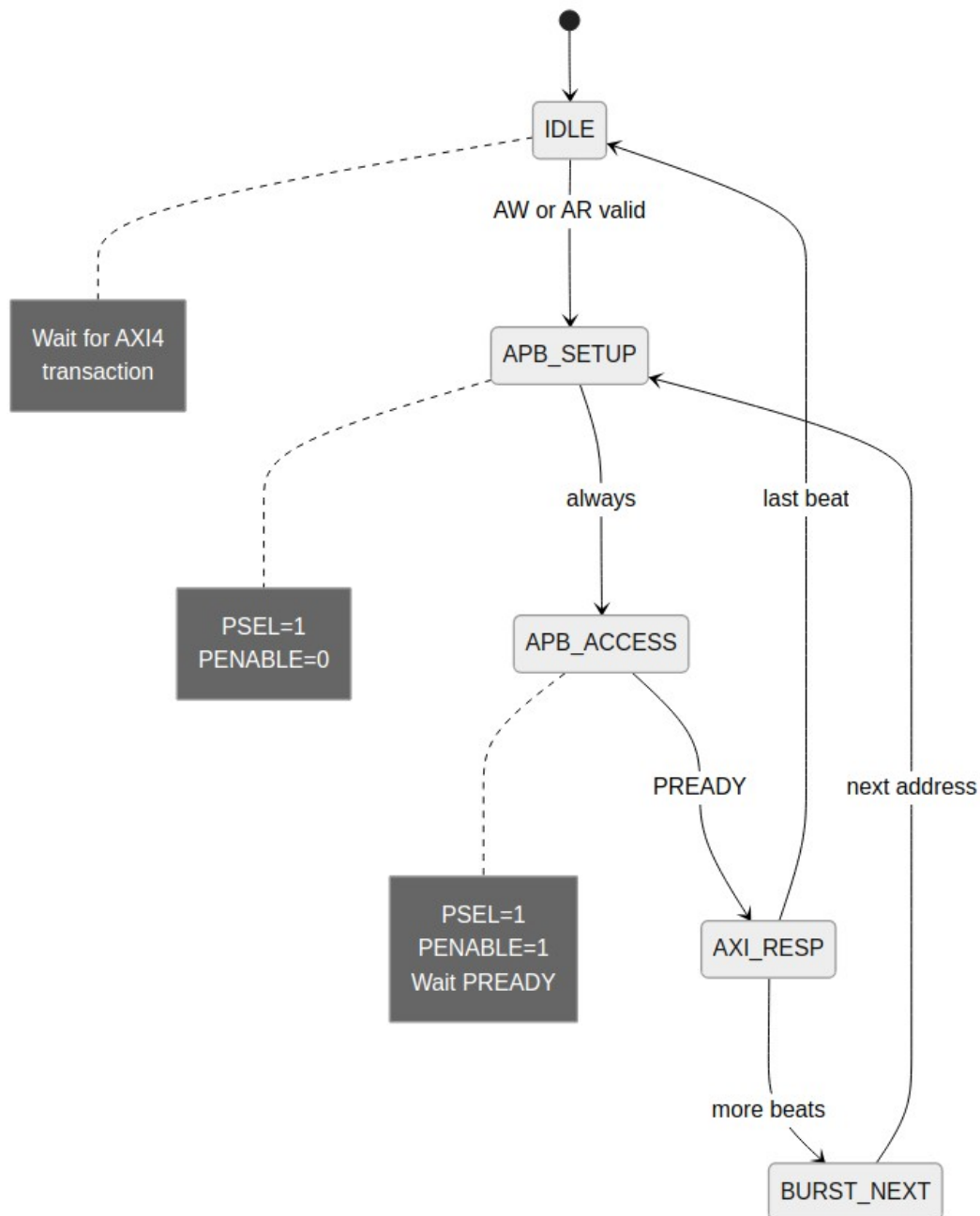
    // Issue AXIL4 when both ready
    if (w_issue_axil4) begin
        r_aw_pending <= 1'b0;
        r_w_pending <= 1'b0;
    end
end

assign w_issue_axil4 = (r_aw_pending || s_awvalid) &&
    (r_w_pending || s_wvalid) &&
    m_awready && m_wready;

```

## 16.3 4.2.3 AXI4 to APB FSM

16.3.1 Figure 4.4: AXI4 to APB FSM



*AXI4 to APB FSM*

### 16.3.2 States

Table 4.5: AXI4 to APB FSM States

State	Description
IDLE	Waiting for AXI4 transaction
APB_SETUP	APB setup phase (PSEL=1, PENABLE=0)
APB_ACCESS	APB access phase (PSEL=1, PENABLE=1)
AXI_RESP	Sending AXI4 response
BURST_NEXT	Preparing next beat in burst

### 16.3.3 APB Protocol Phases

Setup Phase:

- PSEL = 1
- PENABLE = 0
- PADDR, PWDATA, PWRITE stable
- Duration: 1 cycle

Access Phase:

- PSEL = 1
- PENABLE = 1
- Wait for PREADY
- Sample PRDATA (read) or complete write
- Duration: 1+ cycles (depends on PREADY)

### 16.3.4 Implementation

```
always_ff @(posedge clk or negedge rst_n) begin
    if (!rst_n) begin
        r_state <= IDLE;
        psel <= 1'b0;
        penable <= 1'b0;
    end else begin
        case (r_state)
            IDLE: begin
                psel <= 1'b0;
                penable <= 1'b0;
                if (s_awvalid || s_arvalid) begin
                    r_state <= APB_SETUP;
                    psel <= 1'b1;
                end
            end
        end
    end
end
```



```

APB_SETUP: begin
    r_state <= APB_ACCESS;
    penable <= 1'b1;
end

APB_ACCESS: begin
    if (pready) begin
        psel <= 1'b0;
        penable <= 1'b0;
        r_state <= AXI_RESP;
    end
end

AXI_RESP: begin
    if ((r_is_write && s_bready) ||
        (!r_is_write && s_rready)) begin
        if (r_beat_count == r_burst_len)
            r_state <= IDLE;
        else
            r_state <= BURST_NEXT;
        end
    end
end

BURST_NEXT: begin
    r_beat_count <= r_beat_count + 1;
    r_current_addr <= r_current_addr + (1 << r_size);
    r_state <= APB_SETUP;
    psel <= 1'b1;
end
endcase
end
end

```

## 16.4 4.2.4 Timing Analysis

---

### 16.4.1 AXI4 to AXIL4 Timing

#### Single-beat transaction:

Cycle 0: AR accepted  
 Cycle 1: AR forwarded to AXIL4  
 Cycle 2: R received from AXIL4  
 Cycle 3: R forwarded to AXI4  
 Total: ~2 cycles overhead (can be pipelined)

#### N-beat burst:

Cycle 0: AR accepted  
Cycles 1-2N: Decomposed transactions  
Total: 2N cycles

## 16.4.2 AXI4 to APB Timing

### Single transfer:

Cycle 0: AXI4 AR/AW accepted  
Cycle 1: APB setup phase  
Cycle 2+: APB access phase (wait PREADY)  
Cycle N: APB complete, AXI4 response  
Total: 3+ cycles (minimum)

---

Next: [Burst Decomposition](#)

## 17 4.3 Burst Decomposition

This section describes how converters decompose AXI4 bursts into smaller transactions.

### 17.1 4.3.1 Width Converter Burst Handling

---

#### 17.1.1 Burst Length Adjustment

When converting widths, burst length changes inversely with data width:

$$M\_AWLEN = (S\_AWLEN + 1) / \text{RATIO} - 1$$

Example (64-bit to 512-bit, RATIO=8):

$$S\_AWLEN = 7 \text{ (8 beats} \times 64 \text{ bits} = 512 \text{ bits)}$$

$$M\_AWLEN = (7 + 1) / 8 - 1 = 0 \text{ (1 beat} \times 512 \text{ bits)}$$

$$S\_AWLEN = 15 \text{ (16 beats} \times 64 \text{ bits} = 1024 \text{ bits)}$$

$$M\_AWLEN = (15 + 1) / 8 - 1 = 1 \text{ (2 beats} \times 512 \text{ bits)}$$

#### 17.1.2 Figure 4.5: Width Burst Conversion

##### Width Burst Conversion

*Width Burst Conversion*

#### 17.1.3 Non-Aligned Bursts

When burst length is not a multiple of ratio:

$$S\_AWLEN = 5 \text{ (6 beats)}, \text{ RATIO} = 8$$

$$M\_AWLEN = (5 + 1) / 8 - 1 = -1 \rightarrow 0 \text{ (1 beat)}$$

The 6 narrow beats pack into 1 wide beat.  
Last 2 positions have WSTRB = 0 (no write).

## 17.2 4.3.2 Protocol Converter Burst Handling

---

### 17.2.1 AXI4 to AXI4-Lite Decomposition

AXI4-Lite only supports single-beat transactions, so all bursts must be decomposed:

AXI4 Burst:

ARADDR = 0x1000  
ARLEN = 3 (4 beats)  
ARSIZE = 2 (4 bytes)

AXI4 Sequence:

Transaction 0: ARADDR = 0x1000  
Transaction 1: ARADDR = 0x1004  
Transaction 2: ARADDR = 0x1008  
Transaction 3: ARADDR = 0x100C

### 17.2.2 Address Increment Calculation

```
// Calculate address increment based on burst type and size
function automatic [ADDR_WIDTH-1:0] next_address(
    input [ADDR_WIDTH-1:0] current_addr,
    input [2:0] size,
    input [1:0] burst,
    input [7:0] len,
    input [7:0] beat
);
    logic [ADDR_WIDTH-1:0] increment;
    logic [ADDR_WIDTH-1:0] wrap_mask;

    increment = 1 << size;

    case (burst)
        2'b00: // FIXED
            return current_addr; // No increment

        2'b01: // INCR
            return current_addr + increment;

        2'b10: // WRAP
            wrap_mask = ((len + 1) << size) - 1;
            return (current_addr & ~wrap_mask) |
                ((current_addr + increment) & wrap_mask);
    endcase
endfunction
```

```

        default:
            return current_addr + increment;
    endcase
endfunction

```

## 17.3 4.3.3 Response Aggregation

---

### 17.3.1 Read Response Aggregation

For burst reads decomposed into multiple single reads:

```

// Track responses as they arrive
logic [7:0] r_response_count;
logic [1:0] r_worst_rresp;

always_ff @(posedge clk) begin
    if (start_new_burst) begin
        r_response_count <= '0;
        r_worst_rresp <= 2'b00; // OKAY
    end else if (m_rvalid && m_rready) begin
        r_response_count <= r_response_count + 1;
        // Keep worst response
        if (m_rresp > r_worst_rresp)
            r_worst_rresp <= m_rresp;
    end
end

// Generate RLAST on final beat
assign s_rlast = (r_response_count == r_original_arlen);

// Forward individual responses or aggregate
assign s_rresp = m_rresp; // Forward each response
// Or: assign s_rresp = r_worst_rresp; // Aggregate

```

### 17.3.2 Write Response Aggregation

For burst writes:

```

// Track worst response across burst
logic [1:0] r_worst_bresp;
logic r_all_beats_done;

always_ff @(posedge clk) begin
    if (start_new_burst)
        r_worst_bresp <= 2'b00;
    else if (m_bvalid && m_bready)

```

```

        r_worst_bresp <= (m_bresp > r_worst_bresp) ? m_bresp :
r_worst_bresp;
end

```

```

// Send single aggregated B response

```

```

assign s_bvalid = r_all_beats_done;
assign s_bresp = r_worst_bresp;

```

## 17.4 4.3.4 Burst Tracking Registers

---

### 17.4.1 Required State

```

// Burst tracking registers

```

```

logic [ADDR_WIDTH-1:0] r_base_addr;
logic [ADDR_WIDTH-1:0] r_current_addr;
logic [7:0]             r_original_len;
logic [7:0]             r_remaining_beats;
logic [2:0]             r_size;
logic [1:0]             r_burst;
logic [ID_WIDTH-1:0]    r_id;
logic                  r_is_write;

```

### 17.4.2 Initialization

```

always_ff @(posedge clk) begin
    if (accept_new_transaction) begin
        r_base_addr <= s_axaddr;
        r_current_addr <= s_axaddr;
        r_original_len <= s_axlen;
        r_remaining_beats <= s_axlen;
        r_size <= s_axsize;
        r_burst <= s_axburst;
        r_id <= s_axid;
        r_is_write <= is_write_transaction;
    end else if (beat_complete) begin
        r_current_addr <= next_address(...);
        r_remaining_beats <= r_remaining_beats - 1;
    end
end
end

```

## 17.5 4.3.5 Timing Impact

---

### 17.5.1 Decomposition Overhead

Table 4.6: Decomposition Overhead

Transaction Type	Overhead
Single-beat	0 cycles (passthrough)

Transaction Type	Overhead
2-beat burst	2 cycles (sequential)
N-beat burst	2N cycles (2 per beat)

## 17.5.2 Pipeline Considerations

Decomposition is inherently sequential: - Cannot issue next AR until previous R received (AXIL4) - Cannot issue next AW/W until previous B received (AXIL4)

**Optimization:** Use response pipelining when downstream supports it.

Next: [Chapter 5: Verification](#)

# 18 5.1 Test Strategy

This section describes the verification approach for converter modules.

## 18.1 5.1.1 Test Organization

### 18.1.1 Test Hierarchy

```

projects/components/converters/dv/tests/
├── width_converters/
│   ├── test_axi_data_upsize.py
│   ├── test_axi_data_dsize.py
│   ├── test_axi4_dwidth_converter_wr.py
│   └── test_axi4_dwidth_converter_rd.py
├── protocol_converters/
│   ├── test_axi4_to_axil4_rd.py
│   ├── test_axi4_to_axil4_wr.py
│   ├── test_axil4_to_axi4_rd.py
│   ├── test_axil4_to_axi4_wr.py
│   ├── test_axi4_to_apb.py
│   └── test_peakrdl_adapter.py
└── integration/
    ├── test_width_protocol_chain.py
    └── test_full_system.py

```

## 18.2 5.1.2 Test Levels

### 18.2.1 Level 1: Unit Tests

**Purpose:** Verify individual module functionality in isolation.

**Coverage:** - All parameters combinations - Edge cases (min/max values) - Error injection

**Example:**

```
@pytest.mark.parametrize("narrow_width,wide_width", [
    (32, 64),
    (32, 128),
    (64, 256),
    (64, 512),
    (128, 1024),
])
async def test_upsize_ratios(dut, narrow_width, wide_width):
    """Test various width ratios."""
    tb = UpsizeTB(dut, narrow_width, wide_width)
    await tb.reset()
    await tb.run_basic_transfer(count=100)
    assert tb.scoreboard.check_passed()
```

## 18.2.2 Level 2: Integration Tests

**Purpose:** Verify module combinations work together.

**Coverage:** - Width converter + protocol converter chains - Back-to-back converters - Mixed traffic patterns

## 18.2.3 Level 3: System Tests

**Purpose:** Verify in realistic system context.

**Coverage:** - CPU-to-DDR paths - Peripheral access paths - Full bandwidth stress

## 18.3 5.1.3 Test Categories

---

### 18.3.1 Functional Tests

*Table 5.1: Functional Test Categories*

Category	Description	Example
Basic	Single transactions	Single read, single write
Burst	Multi-beat transactions	INCR burst, WRAP burst
Mixed	Interleaved R/W	Read-modify-write sequences
Edge	Boundary conditions	Max burst length, min width

### 18.3.2 Stress Tests

Table 5.2: Stress Test Categories

Category	Description	Duration
Throughput	Maximum bandwidth	10,000 transactions
Backpressure	Ready signal variations	Random delays
Reset	Reset during operation	Mid-transaction reset

### 18.3.3 Error Tests

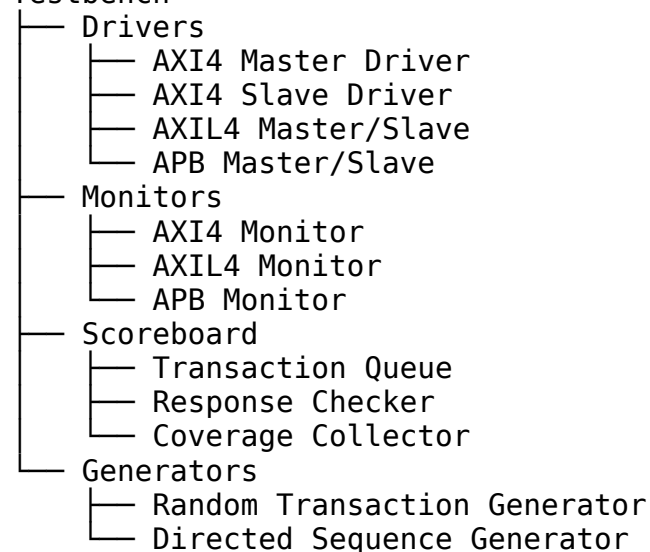
Table 5.3: Error Test Categories

Category	Description	Expected Behavior
SLVERR	Slave error injection	Error propagation
DECERR	Decode error	Error response
Timeout	Response timeout	Error handling

## 18.4 5.1.4 Testbench Architecture

### 18.4.1 Components

#### Testbench





## 18.4.2 Driver Implementation

```
class AXI4MasterDriver:
    def __init__(self, dut, clock, prefix="s_axi"):
        self.dut = dut
        self.clock = clock
        self.prefix = prefix

    async def write(self, addr, data, burst_len=0):
        """Issue AXI4 write transaction."""
        # AW phase
        self.dut.awvalid.value = 1
        self.dut.awaddr.value = addr
        self.dut.awlen.value = burst_len
        await self._wait_ready("awready")

        # W phase
        for i, d in enumerate(data):
            self.dut.wvalid.value = 1
            self.dut.wdata.value = d
            self.dut.wlast.value = (i == len(data) - 1)
            await self._wait_ready("wready")

        # B phase
        await self._wait_valid("bvalid")
        return self.dut.bresp.value
```

## 18.5 5.1.5 Coverage Model

---

### 18.5.1 Functional Coverage

```
@coverage
class ConverterCoverage:
    # Width ratio coverage
    ratio_cp = coverpoint(
        lambda: self.width_ratio,
        bins=[2, 4, 8, 16]
    )

    # Burst length coverage
    burst_len_cp = coverpoint(
        lambda: self.burst_len,
        bins=list(range(0, 256, 16)) + [255]
    )

    # Burst type coverage
    burst_type_cp = coverpoint(
```

```

        lambda: self.burst_type,
        bins=["FIXED", "INCR", "WRAP"]
    )

    # Cross coverage
    ratio_x_burst = cross(ratio_cp, burst_len_cp)

```

## 18.5.2 Code Coverage

**Target:** >95% line coverage, >90% branch coverage

*Table 5.4: Coverage Targets*

Module	Line	Branch	FSM
axi_data_upsize	98%	95%	100%
axi_data_dnsiz	97%	93%	100%
axi4_to_axil4_r	96%	91%	100%
axi4_to_axil4_wr	95%	90%	100%

## 18.6 5.1.6 Test Execution

### 18.6.1 Running Tests

```

# Run all converter tests
cd projects/components/converters/dv/tests
pytest -v

```

```

# Run specific module tests
pytest test_axi_data_upsize.py -v

```

```

# Run with coverage
pytest --cov=projects/components/converters/rtl -v

```

```

# Run with waveform dump
pytest test_axi_data_upsize.py -v --waves

```

### 18.6.2 CI/CD Integration

```

converter_tests:
  stage: test
  script:

```

```
- cd projects/components/converters/dv/tests
- pytest -v --junitxml=results.xml
- pytest --cov=rtl --cov-report=html
artifacts:
  reports:
    junit: results.xml
  paths:
    - htmlcov/
```

---

Next: [Debug Guide](#)

## 19 5.2 Debug Guide

This section provides debugging guidance for converter module issues.

### 19.1 5.2.1 Common Issues

---

#### 19.1.1 Width Converter Issues

##### 19.1.1.1 *Issue: Data Corruption*

**Symptoms:** - Output data doesn't match expected - Random bits flipped or missing

**Debug Steps:** 1. Check width ratio calculation:  $RATIO = WIDE\_WIDTH / NARROW\_WIDTH$  2. Verify beat counter is correct:  $\lceil \log_2(RATIO) \rceil$  bits 3. Check data packing/unpacking slice indices 4. Verify sideband mode matches use case

**Waveform Checkpoints:**

r_count	- Should cycle 0 to RATIO-1
r_data	- Check each slice is populated
s_data/m_data	- Compare input/output patterns

##### 19.1.1.2 *Issue: LAST Signal Incorrect*

**Symptoms:** - Transaction ends early or late - Master receives wrong beat count

**Debug Steps:** 1. Check USE\_LAST parameter 2. Verify burst tracker logic if enabled 3. Check s\_last input timing

**Solution:**

```
// Verify LAST generation
assign m_last = (r_count == RATIO - 1) || r_input_last_seen;
```

### 19.1.1.3 *Issue: Throughput Lower Than Expected*

**Symptoms:** - Gaps between output beats - Single-buffer achieving <80%

**Debug Steps:** 1. Check downstream ready signal behavior 2. Verify DUAL\_BUFFER parameter for downsize 3. Look for backpressure stalls

## 19.1.2 Protocol Converter Issues

### 19.1.2.1 *Issue: Burst Decomposition Incorrect*

**Symptoms:** - Wrong number of single transactions - Address increment wrong

**Debug Steps:** 1. Check burst type (FIXED, INCR, WRAP) 2. Verify size calculation 3. Check address increment logic

**Waveform Checkpoints:**

r\_current\_addr - Should increment by (1 << size)  
r\_beat\_count - Should count to s\_arlen/s\_awlen  
m\_arvalid - Should assert for each decomposed beat

### 19.1.2.2 *Issue: Response Aggregation Wrong*

**Symptoms:** - Wrong BRESP/RRESP value - Error not propagated correctly

**Debug Steps:** 1. Check worst-case response tracking 2. Verify response counter 3. Check RLAST generation

**Solution:**

```
// Proper error aggregation
always_ff @(posedge clk) begin
    if (new_burst)
        r_worst_resp <= 2'b00;
    else if (m_rvalid && m_rready)
        r_worst_resp <= (m_rresp > r_worst_resp) ? m_rresp :
r_worst_resp;
end
```

## 19.2 5.2.2 Debug Signals

---

### 19.2.1 Recommended Internal Signals

For width converters:

```
// Add debug outputs
output logic [$clog2(RATIO)-1:0] dbg_beat_count,
output logic                    dbg_buffer_valid,
output logic [1:0]              dbg_state
```

For protocol converters:

```
// Add debug outputs
output logic [7:0] dbg_remaining_beats,
output logic [2:0] dbg_state,
output logic [1:0] dbg_worst_resp,
output logic      dbg_in_burst
```

### 19.2.2 ILA Configuration

```
# Create ILA for converter debug
create_debug_core u_ila ila
```

```
# Add probes
```

```
set_property probe_count 10 [get_debug_cores u_ila]
connect_debug_port u_ila/clk [get_nets aclk]
```

```
# Key signals
```

```
connect_debug_port u_ila/probe0 [get_nets r_state]
connect_debug_port u_ila/probe1 [get_nets r_beat_count]
connect_debug_port u_ila/probe2 [get_nets s_arvalid]
connect_debug_port u_ila/probe3 [get_nets m_arvalid]
```

## 19.3 5.2.3 Simulation Debug

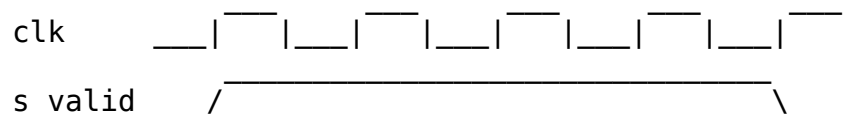
---

### 19.3.1 Waveform Analysis

Key Signal Groups:

1. **Input Channel:**
  - s\_valid, s\_ready, s\_data, s\_last
2. **Output Channel:**
  - m\_valid, m\_ready, m\_data, m\_last
3. **Control:**
  - r\_state, r\_count, r\_burst\_remaining
4. **Sideband:**
  - s\_wstrb/m\_wstrb (write path)
  - s\_rresp/m\_rresp (read path)

### 19.3.2 Timing Diagram Template



```

        [D0 ][D1 ][D2 ][D3 ][D4 ][D5 ][D6 ][D7 ]
s_data
s_ready _____

m_valid __|_____|__
                                     [WIDE_DATA ]
m_data

```

Check:

1. s\_ready stays high during accumulation
2. m\_valid asserts after RATIO beats
3. Data packing is correct

## 19.4 5.2.4 Common Mistakes

---

### 19.4.1 Mistake 1: Wrong Width Ratio

```

// WRONG: Manual ratio
localparam RATIO = 8; // May not match actual widths

// CORRECT: Calculated ratio
localparam RATIO = WIDE_WIDTH / NARROW_WIDTH;

```

### 19.4.2 Mistake 2: Missing Sideband Handling

```

// WRONG: Forgetting sideband
assign m_data = r_data;
// Missing: assign m_wstrb = ...

```

```

// CORRECT: Handle both
assign m_data = r_data;
assign m_wstrb = r_sideband;

```

### 19.4.3 Mistake 3: Incorrect LAST Timing

```

// WRONG: LAST on wrong beat
assign m_last = r_count == 0; // First beat!

// CORRECT: LAST on final beat
assign m_last = (r_count == RATIO - 1) || r_early_last;

```

### 19.4.4 Mistake 4: Burst Length Calculation Error

```

// WRONG: Off-by-one
assign m_awlen = s_awlen / RATIO; // Wrong!

```

```
// CORRECT: Account for LEN encoding
assign m_awlen = ((s_awlen + 1) >> RATIO_LOG2) - 1;
```

## 19.5 5.2.5 Verification Checklist

---

Before signoff, verify:

- ☐ All parameter combinations tested
- ☐ Single-beat transactions work
- ☐ Multi-beat bursts work (INCR, WRAP, FIXED)
- ☐ Backpressure handling correct
- ☐ Error propagation correct
- ☐ LAST signal timing correct
- ☐ Sideband signals handled correctly
- ☐ Reset behavior verified
- ☐ Coverage targets met

## 19.6 5.2.6 Performance Validation

---

### 19.6.1 Throughput Measurement

```
async def measure_throughput(tb, transaction_count=1000):
    start_time = get_sim_time()

    for _ in range(transaction_count):
        await tb.send_transaction()

    end_time = get_sim_time()
    elapsed_cycles = (end_time - start_time) / clock_period

    throughput = transaction_count / elapsed_cycles
    print(f"Throughput: {throughput:.2f} transactions/cycle")

    return throughput
```

### 19.6.2 Expected Throughput

*Table 5.5: Expected Throughput*

Module	Mode	Expected
axi_data_upsize	Single	1.0 trans/cycle
axi_data_dnsize	Single	0.8 trans/cycle

Module	Mode	Expected
axi_data_dnsz	Dual	1.0 trans/cycle
axi4_to_axil4	Single-beat	1.0 trans/cycle
axi4_to_axil4	Burst	0.5 trans/cycle

End of Micro-Architecture Specification

## 20 Converters Component Specification

**Version:** 1.5 **Status:** Production Ready **Last Updated:** 2025-11-15

### 20.1 Document Overview

This specification provides comprehensive documentation for the Converters component, which includes data width conversion and protocol conversion modules for seamless system integration.

**Document Structure:** - **Chapter 1:** Overview and Architecture - **Chapter 2:** Data Width Converters - **Chapter 3:** Protocol Converters - **Chapter 4:** Usage Examples and Integration

### 20.2 Quick Navigation

#### 20.2.1 Chapter 1: Overview

- [01\\_introduction.md](#) - Component overview and key features
- [02\\_architecture.md](#) - System architecture and module hierarchy
- [03\\_design\\_philosophy.md](#) - Design decisions and rationale

#### 20.2.2 Chapter 2: Data Width Converters

- [01\\_overview.md](#) - Data width conversion overview
- [02\\_axi\\_data\\_upsize.md](#) - Narrow→Wide accumulator
- [03\\_axi\\_data\\_dnsz.md](#) - Wide→Narrow splitter
- [04\\_dual\\_buffer\\_mode.md](#) - High-performance dual-buffer mode
- [05\\_axi4\\_dwidth\\_converter\\_wr.md](#) - Full write path converter
- [06\\_axi4\\_dwidth\\_converter\\_rd.md](#) - Full read path converter



### 20.2.3 Chapter 3: Protocol Converters

- [01\\_overview.md](#) - Protocol conversion overview
- [02\\_axi4\\_to\\_apb.md](#) - AXI4-to-APB bridge
- [03\\_peakrdl\\_adapter.md](#) - PeakRDL adapter
- [04\\_axi4\\_to\\_axil4.md](#) - AXI4→AXIL4 protocol downgrade
- [05\\_axil4\\_to\\_axi4.md](#) - AXIL4→AXI4 protocol upgrade
- [06\\_uart\\_axil\\_bridge.md](#) - UART-to-AXI4-Lite bridge (planned)

### 20.2.4 Chapter 4: Usage Examples

- [01\\_configuration\\_guide.md](#) - Parameter configuration guide
- [02\\_common\\_scenarios.md](#) - Common use case examples
- [03\\_integration\\_patterns.md](#) - System integration patterns

### 20.2.5 Appendices

- [Appendix A: Design Notes](#) - Design analysis and rationale
    - [A.1 APB Converter Analysis](#)
    - [A.2 Dual-Buffer Implementation](#)
  - [Appendix B: Change Logs](#) - Refactoring history and updates
    - [B.1 Refactoring Summary](#)
    - [B.2 Documentation Update](#)
- 

## 20.3 Component Summary

---

### 20.3.1 Data Width Converters

**Generic Building Blocks:** - **axi\_data\_upsize** - Accumulates N narrow beats into 1 wide beat (100% throughput) - **axi\_data\_dnsize** - Splits 1 wide beat into N narrow beats (80% or 100% throughput)

**Full AXI4 Converters:** - **axi4\_dwidth\_converter\_wr** - Complete write path (AW + W + B channels) - **axi4\_dwidth\_converter\_rd** - Complete read path (AR + R channels)

**Block Diagrams:**

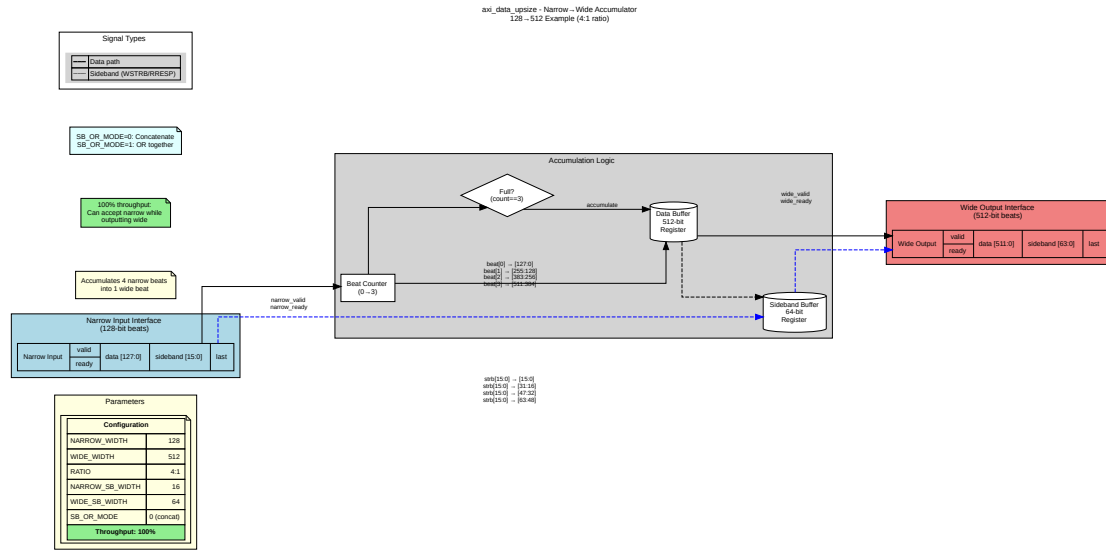


Figure 1:

### axi\_data\_upsize - Narrow-to-Wide Accumulator

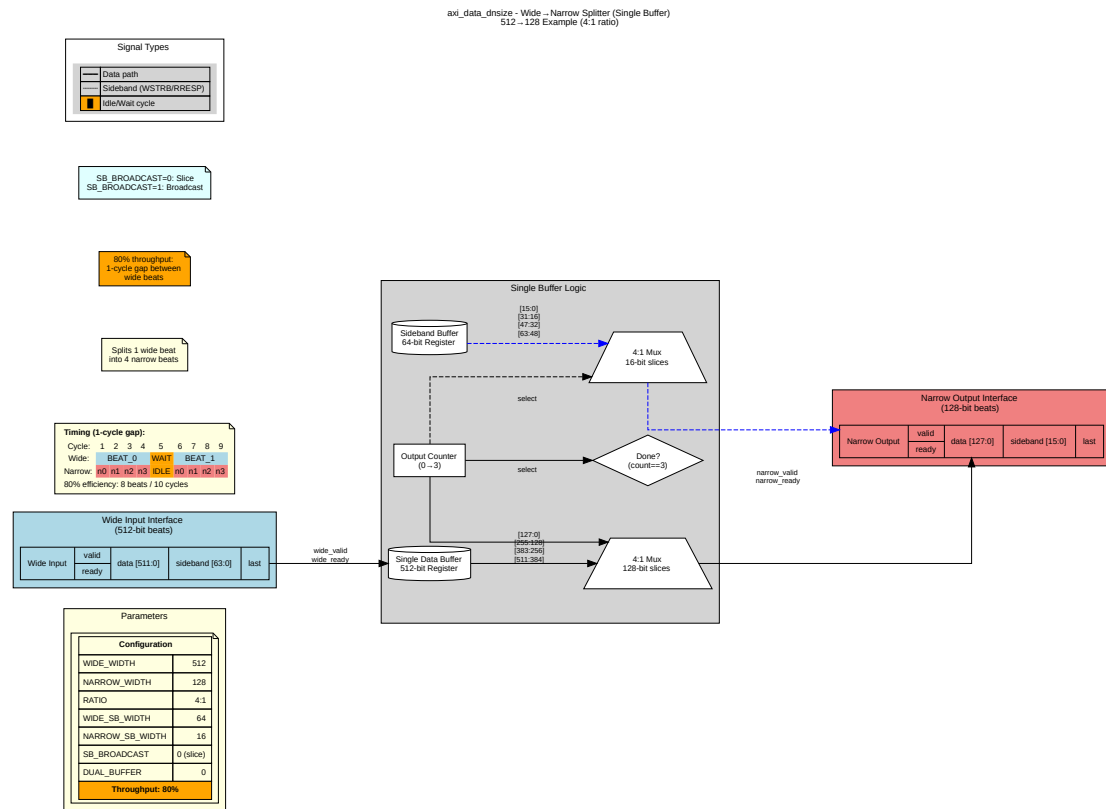


Figure 2:

### axi\_data\_dsize - Wide-to-Narrow Splitter (Single Buffer, 80% throughput)





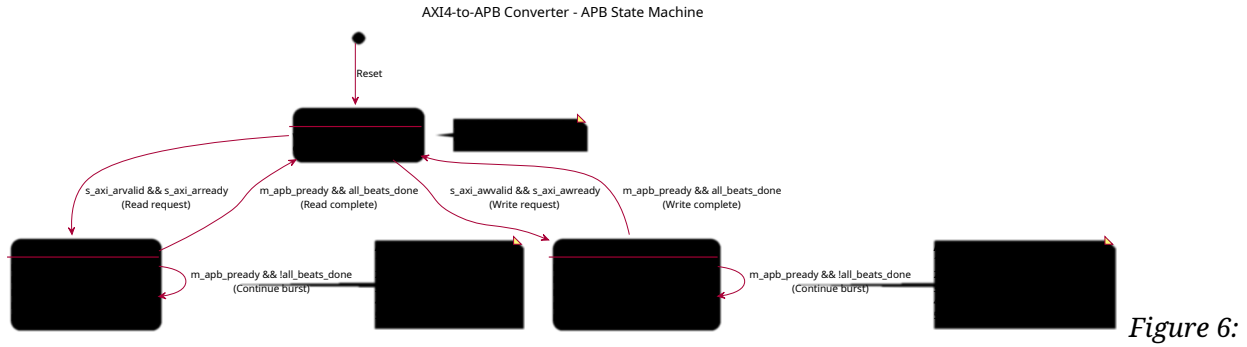


Figure 6:

AXI4-to-APB State Machine

## 20.4 Performance Summary

Module	Configuration	Throughput	Area	Use Case
<b>axi_data_upsize</b>	Single buffer	100%	1×	Narrow→Wide (always optimal)
<b>axi_data_dnsz</b>	Single buffer	80%	1×	Wide→Narrow (area-efficient)
<b>axi_data_dnsz</b>	Dual buffer	100%	2×	Wide→Narrow (high-performance)
<b>axi4_dwidth_converter_wr</b>	With upsize	100%	Standard	Full write path
<b>axi4_dwidth_converter_rd</b>	With dual dnsz	100%	+100%	Full read path
<b>axi4_to_axil4</b>	Burst decomposition	100% (single) / 50% (burst)	~450 LUTs + FFs	AXI4 master → AXIL4 slave
<b>axil4_to_axi4</b>	Combinational upgrade	100%	~110 LUTs	AXIL4 slave → AXI4 fabric
<b>axi4_to_apb</b>	Protocol FSM	Sequential	Small	AXI4 → APB peripherals

## 20.5 Key Features

---

### 20.5.1 Data Width Conversion

- Flexible width ratios (any integer multiple: 2:1, 4:1, 8:1, 16:1, etc.)
- Configurable sideband handling (concatenate or broadcast)
- Optional burst tracking for LAST signal generation
- Dual-buffer mode for 100% throughput (downsize only)
- Generic building blocks for custom converters

### 20.5.2 Protocol Conversion

- **AXI4 ↔ AXI4-Lite:** Bidirectional protocol conversion
    - Burst decomposition (AXI4→AXIL4) with FSM-based splitting
    - Zero-overhead upgrade (AXIL4→AXI4) with combinational defaults
    - Production ready: 42/42 tests passing
  - **AXI4-to-APB:** Full protocol translation
    - Address width adaptation (64-bit→32-bit)
    - Data width adaptation (configurable)
    - Error response mapping (PSLVERR→BRESP/RRESP)
  - **PeakRDL adapter:** Register interface decoupling
- 

## 20.6 Documentation Assets

---

### 20.6.1 Graphviz Block Diagrams

Located in `assets/graphviz/`: - `axi_data_upsize.gv.svg` - `axi_data_dsize_single.gv.svg` - `axi_data_dsize_dual.gv.svg` - `axi4_dwidth_converter_wr.gv.svg` - `axi4_dwidth_converter_rd.gv.svg` - `axi4_to_apb.gv.svg` - `peakrdl_adapter.gv.svg` - `axi4_to_axil4.gv.svg` (wrapper) - `axi4_to_axil4_rd.gv.svg` (read path) - `axi4_to_axil4_wr.gv.svg` (write path) - `axil4_to_axi4.gv.svg` (wrapper) - `axil4_to_axi4_rd.gv.svg` (read path) - `axil4_to_axi4_wr.gv.svg` (write path) - `uart_axil_bridge.gv.svg`

See [assets/graphviz/README.md](#) for regeneration instructions.

### 20.6.2 PlantUML FSM Diagrams

Located in `assets/puml/`: - `axi4_to_apb_fsm.puml.svg`

See [assets/puml/README.md](#) for regeneration instructions.

---

## 20.7 Related Documentation

---

**Component-Level:** - [../README.md](#) - Component quick start guide -  
[../DUAL\\_BUFFER\\_IMPLEMENTATION.md](#) - Dual-buffer feature deep dive -  
[../ANALYSIS\\_APB\\_CONVERTER.md](#) - APB converter analysis

**Repository-Level:** - RTL source: `projects/components/converters/rtl/` - Test suite:  
`projects/components/converters/dv/tests/` - Usage examples: Chapter 4 of this  
specification

---

## 20.8 Version History

---

Version	Date	Description
1.5	2025-11-15	Added appendices, UART chapter, 6 new diagrams for AXI4 ↔ AXIL4 converters
1.4	2025-11-10	Refactored AXI4 ↔ AXIL4 wrappers to composition pattern (instantiation vs duplication)
1.3	2025-11-06	Added AXI4 ↔ AXIL4 protocol converters (6 modules, 42 tests passing)
1.2	2025-10-26	Added comprehensive specification with block diagrams
1.1	2025-10-25	Added dual-buffer mode for <code>axi_data_dsize</code>
1.0	2025-10-24	Initial release with generic and full converters

## 20.9 Document Conventions

---

**Notation:** - `parameter_name` - RTL parameters and signals - **Bold** - Important concepts and module names - *Italic* - Figure captions and references

**Diagrams:** - Light blue - Input interfaces - Light coral - Output interfaces - Light yellow - Control logic - Light gray - Data path elements - Arrows - Data/control flow

---

**Author:** RTL Design Sherpa Project **Maintained By:** Converters Component Team **Last Review:** 2025-11-15