

Table of Contents

Converter Index

Generated: 2025-11-15

Converters Component Specification

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

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

Quick Navigation

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

Chapter 2: Data Width Converters

- [01_overview.md](#) - Data width conversion overview
- [02_axi_data_upsize.md](#) - Narrow→Wide accumulator
- [03_axi_data_dnsize.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

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)

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

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](#)
-

Component Summary

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:

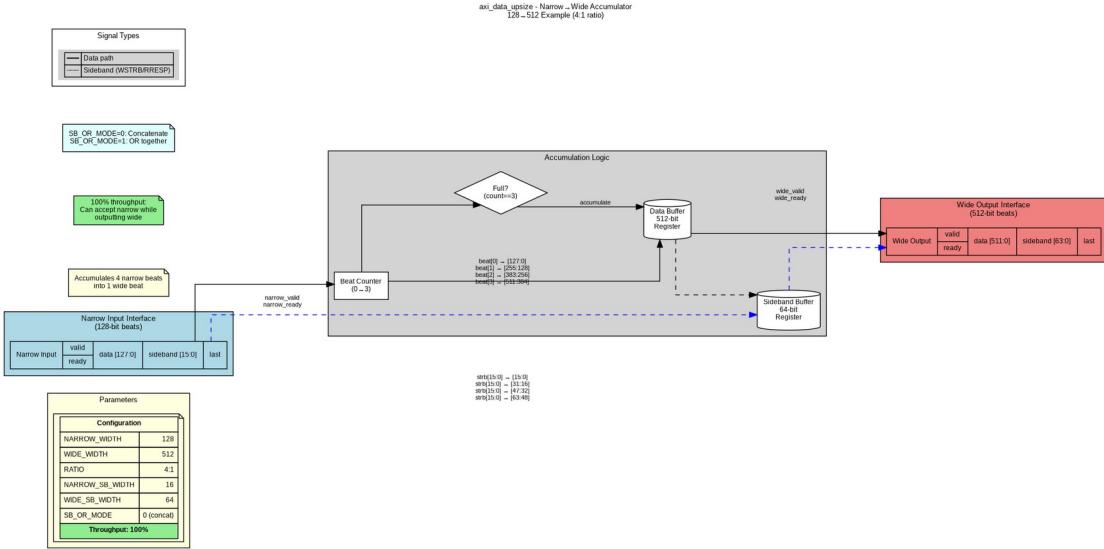


Figure 1: axi_data_upsize - Narrow-to-Wide Accumulator

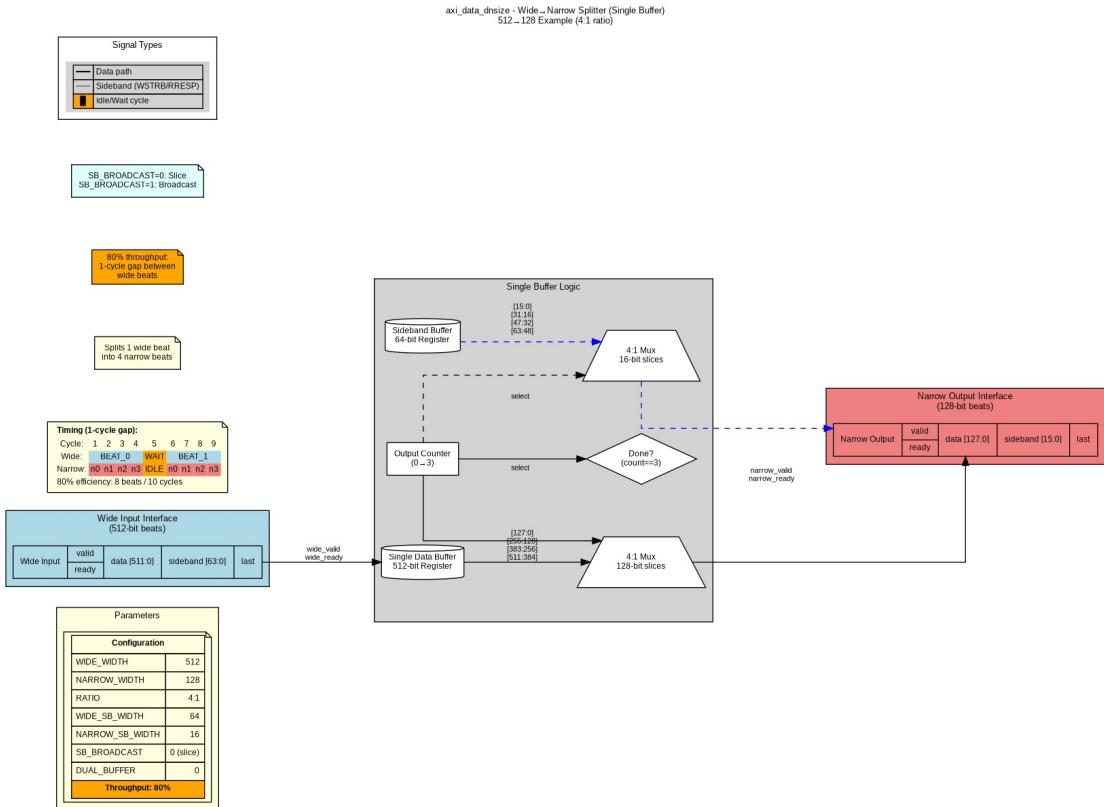


Figure 2: axi_data_dnszie - Wide-to-Narrow Splitter (Single Buffer, 80% throughput)

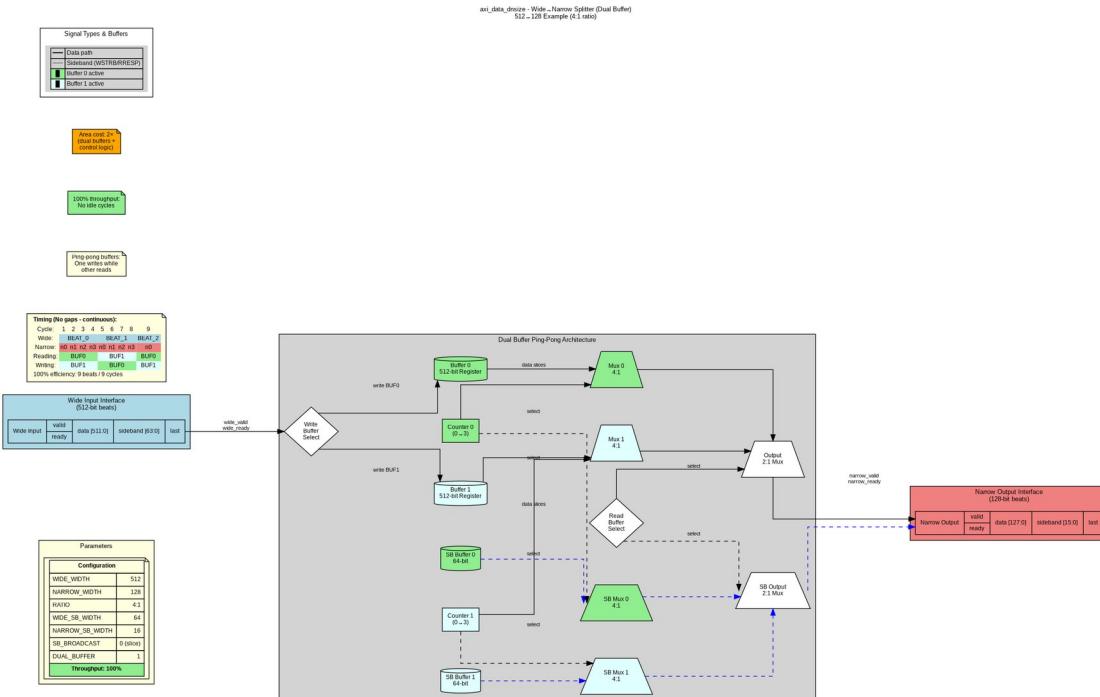


Figure 3: `axi_data_dnsizer` - Wide-to-Narrow Splitter (Dual Buffer, 100% throughput)

Protocol Converters

AXI4 ↔ AXI4-Lite Converters: - `axi4_to_axil4_rd` - AXI4→AXIL4 read converter (burst decomposition) - `axi4_to_axil4_wr` - AXI4→AXIL4 write converter (burst decomposition) - `axi4_to_axil4` - Full bidirectional converter (wrapper) - `axil4_to_axi4_rd` - AXIL4→AXI4 read converter (protocol upgrade) - `axil4_to_axi4_wr` - AXIL4→AXI4 write converter (protocol upgrade) - `axil4_to_axi4` - Full bidirectional converter (wrapper)

Other Protocol Converters: - `axi4_to_apb_convert` - Full AXI4-to-APB protocol bridge - `axi4_to_apb_shim` - Simplified AXI4-to-APB wrapper - `peakrdl_to_cmdrv` - PeakRDL register to command/response adapter - `uart_axil_bridge` - UART ASCII command to AXI4-Lite bridge (see `rtl/uart_to_axil4/README.md`)

Block Diagrams:

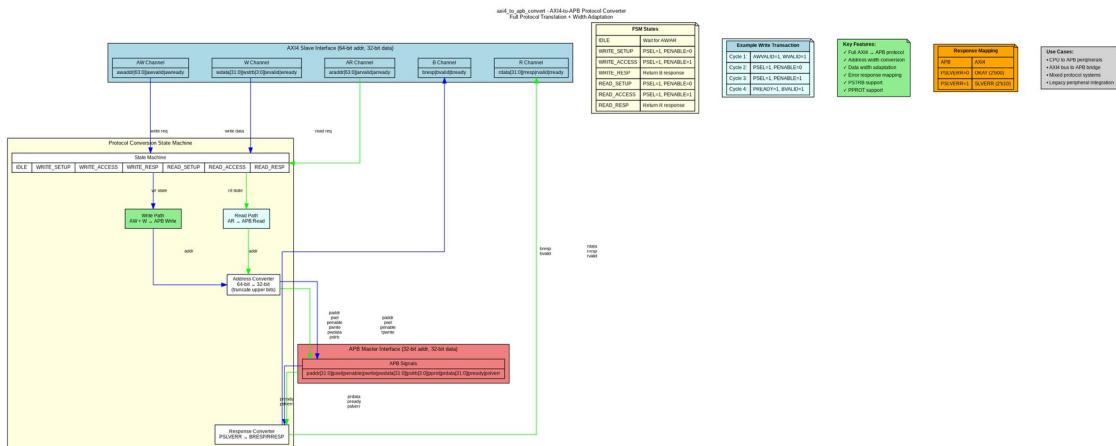


Figure 4: *axi4_to_apb_convert - Protocol Conversion*

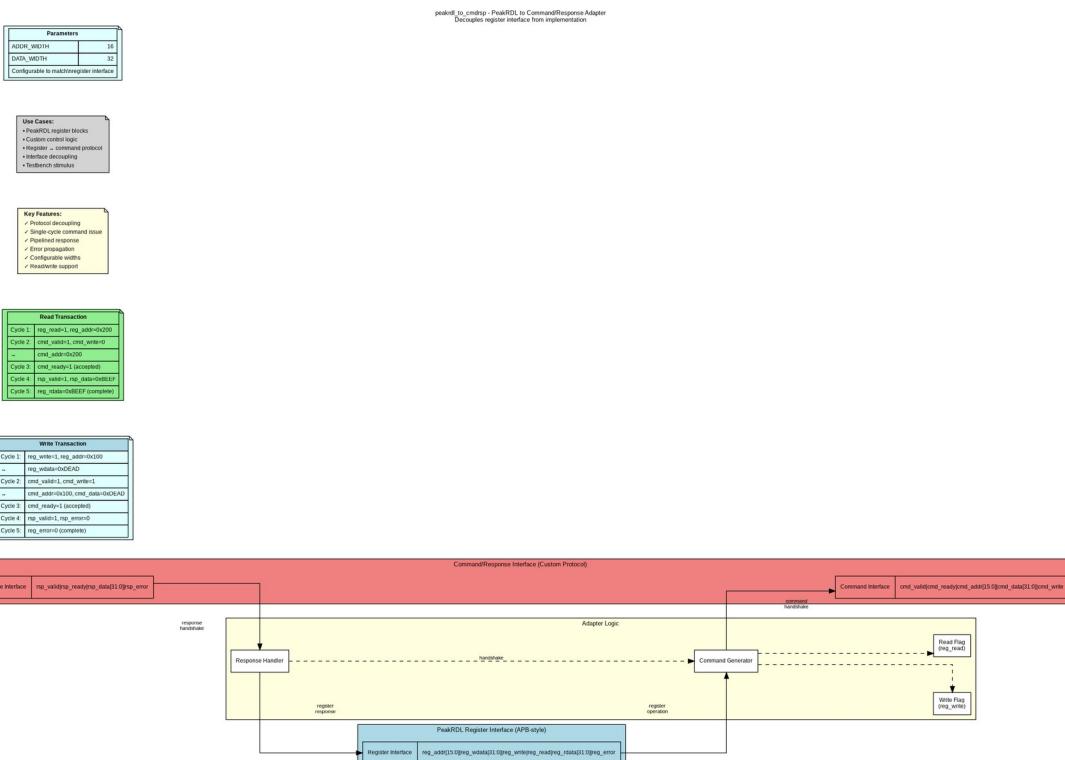


Figure 5: *peakrdl_to_cmdrsp - Register Interface Adapter*

FSM Diagram:

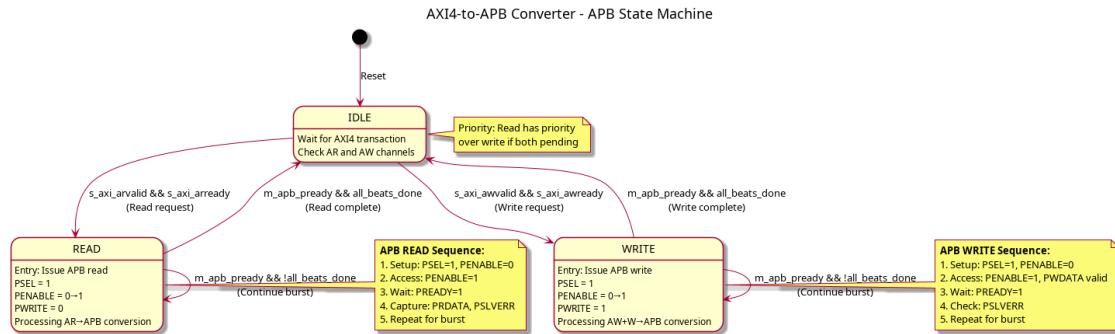


Figure 6: AXI4-to-APB State Machine

Performance Summary

Module	Configuration	Throughput	Area	Use Case
axi_data_upsize	Single buffer	100%	1×	Narrow→Wide (always optimal)
axi_data_dnsize	Single buffer	80%	1×	Wide→Narrow (area-efficient)
axi_data_dnsize	Dual buffer	100%	2×	Wide→Narrow (high-performance)
axi4_dwid_th_converter_wr	With upsize	100%	Standard	Full write path
axi4_dwid_th_converter_rd	With dual dnsize	100%	+100%	Full read path
axi4_to_axil4	Burst decomposition	100% (single) / 50% (burst)	~450 LUTs + FFs	AXI4 master → AXIL4 slave
axil4_to_axi4	Combinational upgrade	100%	~110 LUTs	AXIL4 slave → AXI4 fabric
axi4_to_apb	Protocol FSM	Sequential	Small	AXI4 → APB peripherals

Key Features

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

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
-

Documentation Assets

Graphviz Block Diagrams

Located in assets/graphviz/: - axi_data_upsize.gv/.png -
axi_data_dnsiz_single.gv/.png - axi_data_dnsiz_dual.gv/.png -
axi4_dwidth_converter_wr.gv/.png - axi4_dwidth_converter_rd.gv/.png -
axi4_to_apb.gv/.png - peakrdl_adapter.gv/.png - axi4_to_axil4.gv/.png (wrapper) -
axi4_to_axil4_rd.gv/.png (read path) - axi4_to_axil4_wr.gv/.png (write path) -
axil4_to_axi4.gv/.png (wrapper) - axil4_to_axi4_rd.gv/.png (read path) -
axil4_to_axi4_wr.gv/.png (write path) - uart_axil_bridge.gv/.png

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

PlantUML FSM Diagrams

Located in assets/puml/: - axi4_to_apb_fsm.puml/.png

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

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

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_dnsiz</code>
1.0	2025-10-24	Initial release with generic and full converters

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

Chapter 1: Overview - Introduction

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.

Problem Statement

Modern SoC designs frequently encounter two integration challenges:

1. Data Width Mismatch

- CPU: 64-bit data bus
- DDR Controller: 512-bit data bus
- PCIe Endpoint: 128-bit data bus
- Result: Direct connection impossible

2. Protocol Incompatibility

- AXI4 masters (CPU, DMA)
- APB peripherals (UART, GPIO, SPI)
- Custom register interfaces (PeakRDL)
- Result: Protocol bridge required

Solution Approach

Data Width Converters

Generic Building Blocks: - axi_data_upsize - Accumulates narrow beats into wide beats - axi_data_dnsize - Splits wide beats into narrow beats

Full AXI4 Integration: - axi4_dwidth_converter_wr - Write path converter (AW + W + B) - axi4_dwidth_converter_rd - Read path converter (AR + R)

Protocol Converters

AXI4 ↔ AXI4-Lite: - axi4_to_axil4_rd - AXI4→AXIL4 read converter (burst decomposition)
- axi4_to_axil4_wr - AXI4→AXIL4 write converter (burst decomposition)
- axi4_to_axil4 - Full bidirectional converter
- axil4_to_axi4_rd - AXIL4→AXI4 read converter (protocol upgrade)
- axil4_to_axi4_wr - AXIL4→AXI4 write converter (protocol upgrade)
- axil4_to_axi4 - Full bidirectional converter

Other Protocol Converters: - axi4_to_apb_convert - Full AXI4-to-APB bridge
- peakrdl_to_cmdrsp - Register interface adapter

Key Benefits

Reusability

- Generic modules work with any width ratio
- Configurable parameters for custom use cases
- Self-contained building blocks

Performance

- Upsize: 100% throughput (single buffer)
- Downsize: 80% (single buffer) or 100% (dual buffer)
- Minimal latency overhead

Flexibility

- Configurable sideband handling
- Optional burst tracking
- Dual-buffer mode for high-performance paths

Component Scope

In Scope

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

Out of Scope

- Non-integer width ratios (e.g., 3:2 conversion)
- AXI4-Stream protocol (different component)
- Complex buffering beyond dual-buffer

- Clock domain crossing (use separate CDC modules)

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 to APB peripheral buses
4. **FPGA Fabric Interfaces** - Width matching for IP integration
5. **Register Access** - PeakRDL to custom control protocols

Document Organization

This specification is organized into four chapters:

1. **Overview** (This chapter) - Purpose, scope, and architecture
 2. **Data Width Converters** - Generic and full AXI4 modules
 3. **Protocol Converters** - AXI4-to-APB and PeakRDL adapters
 4. **Usage Examples** - Configuration, scenarios, and integration
-

Next: [02_architecture.md](#) - System architecture and module hierarchy

Chapter 2: Data Width Converters - Overview

Introduction

Data width converters enable communication between components with mismatched data bus widths, a common challenge in modern SoC designs where different IP blocks operate at different data widths.

Module Hierarchy

Data Width Converters



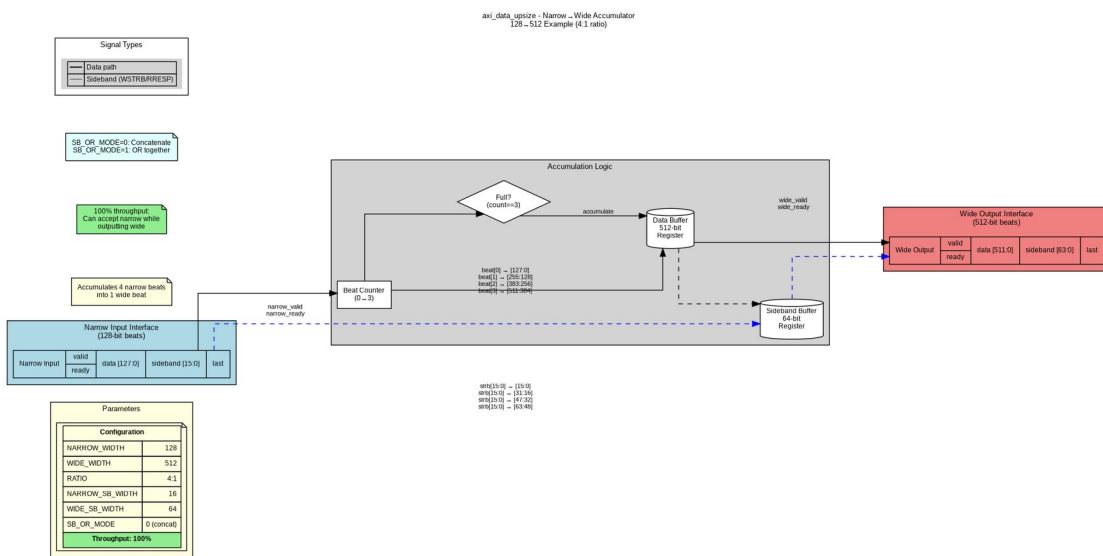
Generic Building Blocks

axi_data_upsize - Narrow→Wide Accumulator

Purpose: Accumulate N narrow beats into 1 wide beat

Key Features: - 100% throughput (single buffer sufficient) - Configurable sideband modes (concatenate or OR) - Supports any integer width ratio - Minimal area overhead

Block Diagram:



AXI Data Upsize

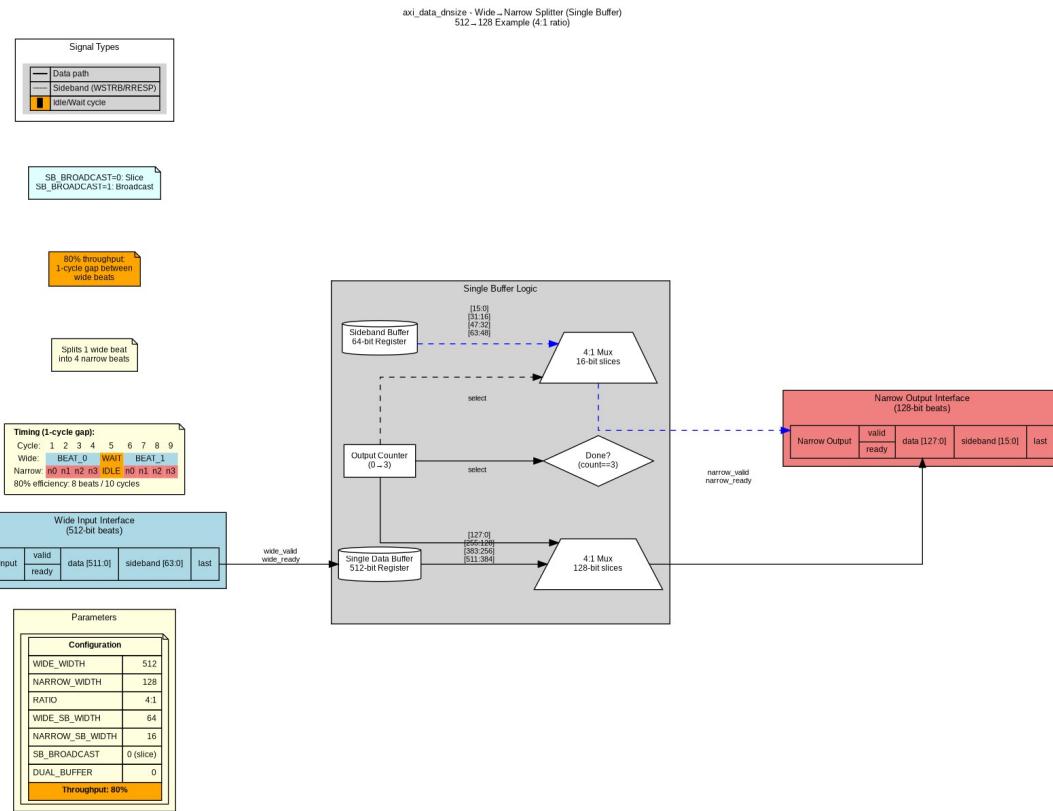
Documentation: [02_axi_data_upsize.md](#)

axi_data_dnsize - Wide→Narrow Splitter

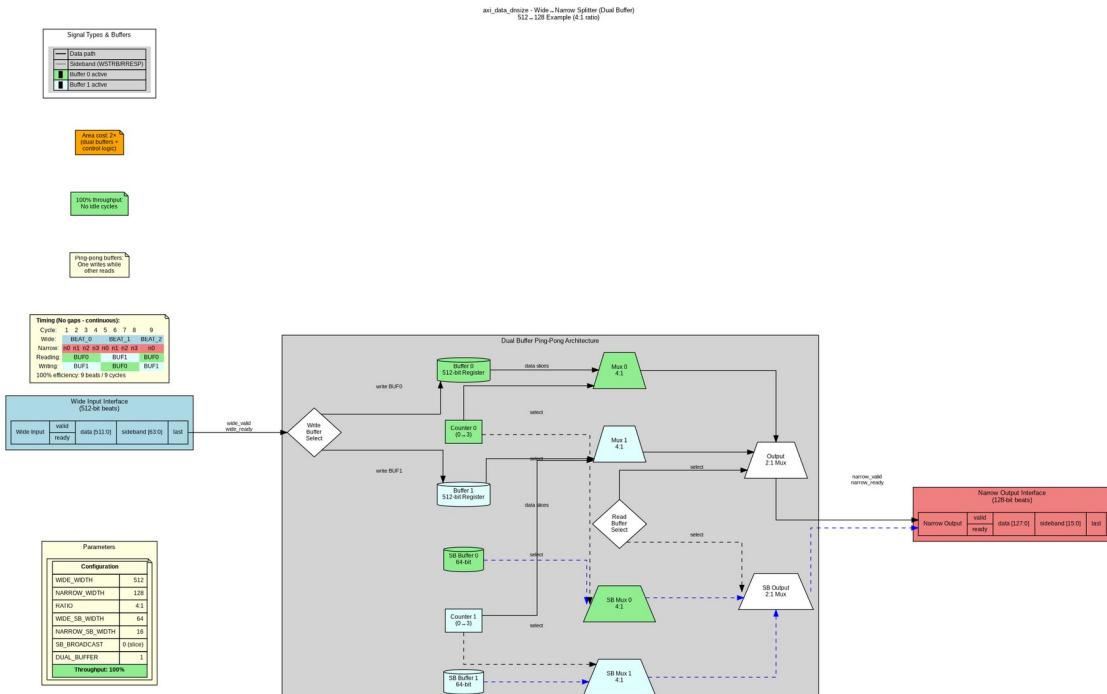
Purpose: Split 1 wide beat into N narrow beats

Key Features: - Two modes: Single buffer (80%) or Dual buffer (100%) throughput - Configurable sideband modes (slice or broadcast) - Optional burst tracking for LAST generation - Trade-off: throughput vs area

Block Diagrams:



Single Buffer Mode - 80% throughput, 1× area



Buffer Mode - 100% throughput, 2× area

Documentation: [03_axi_data_dnsize.md](#) | [04_dual_buffer_mode.md](#)

Dual

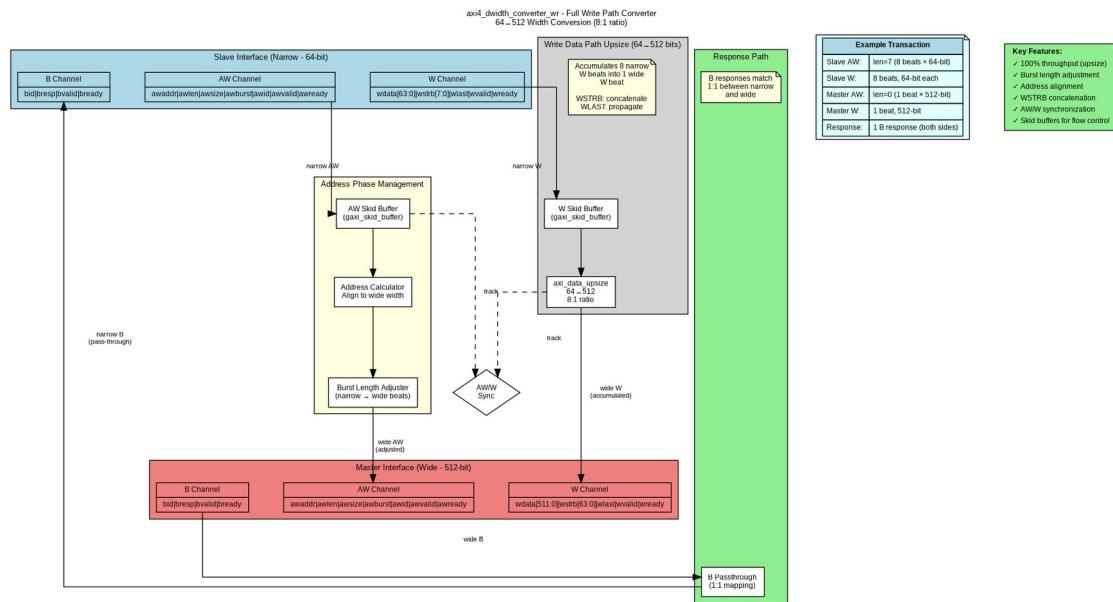
Full AXI4 Converters

axi4_dwidth_converter_wr - Write Path Converter

Purpose: Complete write path conversion with AW, W, and B channels

Features: - Address phase management - Burst length adjustment - Skid buffers for flow control - Integrates axi_data_upsize for data conversion

Block Diagram:



AXI4 Write Converter

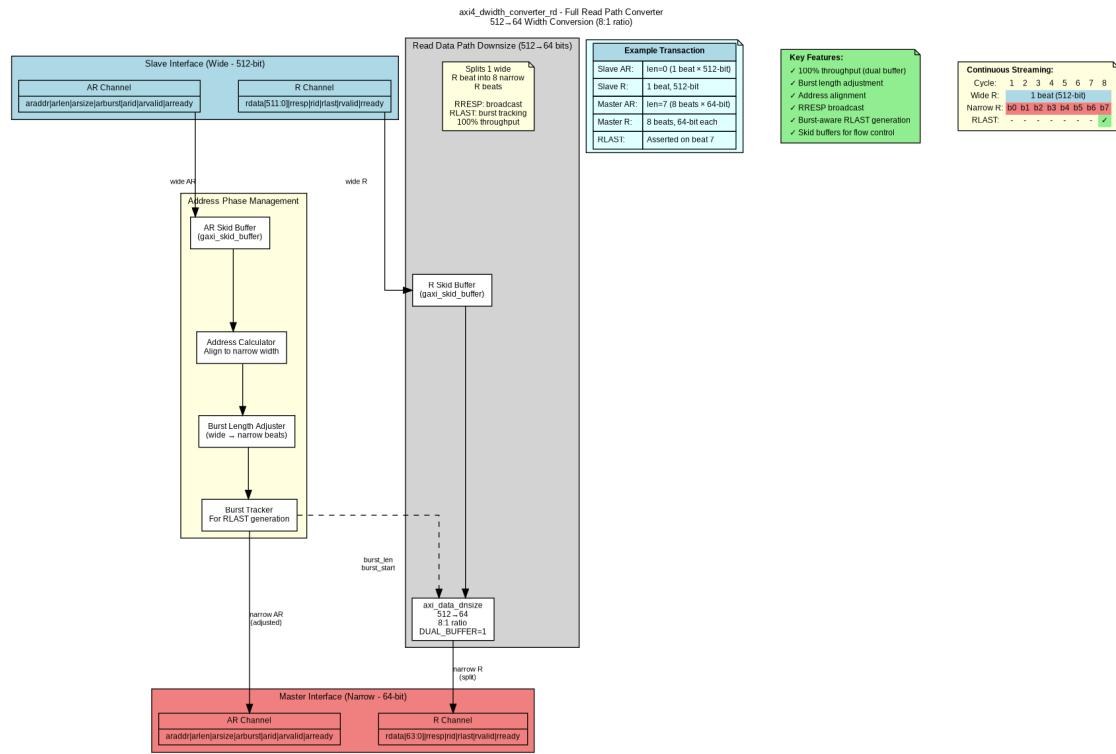
Documentation: 05_axi4_dwidth_converter_wr.md

axi4_dwidth_converter_rd - Read Path Converter

Purpose: Complete read path conversion with AR and R channels

Features: - Address phase management - Burst length adjustment - Burst-aware RLAST generation - Integrates axi_data_dnszie for data conversion

Block Diagram:



AXI4 Read Converter

Documentation: [06_axi4_dwidth_converter_rd.md](#)

Throughput Comparison

Module	Mode	Throughput	Area	When to Use
axi_data_u	Single buffer	100%	1×	All narrow→wide conversions
axi_data_d	Single buffer	80%	1×	Area-constrained designs
axi_data_d	Dual buffer	100%	2×	High-bandwidth requirements

Recommendation: - Upsize: Always use single buffer (100% throughput at minimal cost)
 - Downsize: Choose based on system requirements:
 - Single buffer for area-constrained or non-continuous traffic
 - Dual buffer for high-performance DMA or continuous streaming

Common Configuration Examples

Example 1: 64→512 bit (Write Path)

```
axi_data_upsize #(
    .NARROW_WIDTH(64),
    .WIDE_WIDTH(512),
    .NARROW_SB_WIDTH(8),      // WSTRB
    .WIDE_SB_WIDTH(64),
    .SB_OR_MODE(0)           // Concatenate
) u_upsize (...);
```

Example 2: 512→128 bit (Read Path, High Performance)

```
axi_data_dnsize #(
    .WIDE_WIDTH(512),
    .NARROW_WIDTH(128),
    .WIDE_SB_WIDTH(2),        // RRESP
    .NARROW_SB_WIDTH(2),
    .SB_BROADCAST(1),         // Broadcast
    .DUAL_BUFFER(1)           // 100% throughput
) u_dnsizer (...);
```

Next Sections: - [02_axi_data_upsize.md](#) - Detailed upsize module documentation
- [03_axi_data_dnsizer.md](#) - Detailed dnsizer module documentation -
[04_dual_buffer_mode.md](#) - Dual-buffer architecture deep dive

Chapter 3: Protocol Converters - Overview

Introduction

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

Available Converters

1. AXI4 to AXI4-Lite (Protocol Downgrade)

- **Modules:** axi4_to_axil4_rd.sv, axi4_to_axil4_wr.sv, axi4_to_axil4.sv
- **Purpose:** Burst decomposition from full AXI4 to simplified AXI4-Lite
- **Features:** Multi-beat burst splitting, response aggregation, zero-cycle single-beat passthrough
- **Test Status:** ✓ 28/28 tests passing

2. AXI4-Lite to AXI4 (Protocol Upgrade)

- **Modules:** axil4_to_axi4_rd.sv, axil4_to_axi4_wr.sv, axil4_to_axi4.sv
- **Purpose:** Add AXI4 burst signals with default values
- **Features:** Zero-overhead combinational passthrough, configurable default IDs
- **Test Status:** ✓ 14/14 tests passing

3. AXI4-to-APB Bridge

- **Module:** axi4_to_apb_convert.sv
- **Purpose:** Full protocol translation from AXI4 to APB
- **Features:** Address width adaptation, state machine control, error mapping

4. PeakRDL Adapter

- **Module:** peakrdl_to_cmdrsp.sv
- **Purpose:** Register interface to command/response protocol
- **Features:** Protocol decoupling, single-cycle commands, pipelined responses

5. UART to AXI4-Lite Bridge

- **Module:** uart_axil_bridge.sv (plus uart_rx.sv, uart_tx.sv)
 - **Purpose:** ASCII command-line control of AXI4-Lite peripherals via UART
 - **Features:** Human-readable commands, 115200 baud default, timing isolation
 - **Documentation:** See rtl/uart_to_axil4/README.md
-

AXI4 AXI4-Lite Converters

Overview

Bidirectional protocol conversion between full AXI4 and simplified AXI4-Lite protocols.

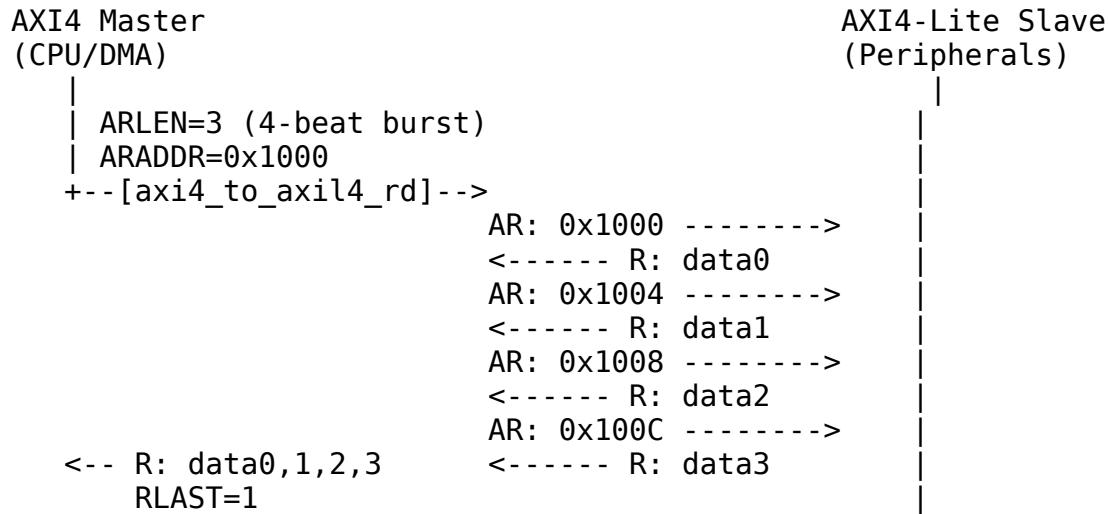
Key Features: - Burst decomposition (AXI4→AXIL4): Multi-beat → multiple single-beat
- Protocol upgrade (AXIL4→AXI4): Add default burst signals
- Independent read/write paths
- Data width must match (no width conversion)
- Production ready with comprehensive test coverage

AXI4 → AXI4-Lite (Downgrade)

Converts full AXI4 with burst support to AXI4-Lite single-beat transactions.

Challenges: - Decomposing multi-beat bursts into sequential single beats - Synchronizing AW and W channels (write path) - Aggregating responses across burst - Maintaining ID through burst decomposition

Block Diagram:



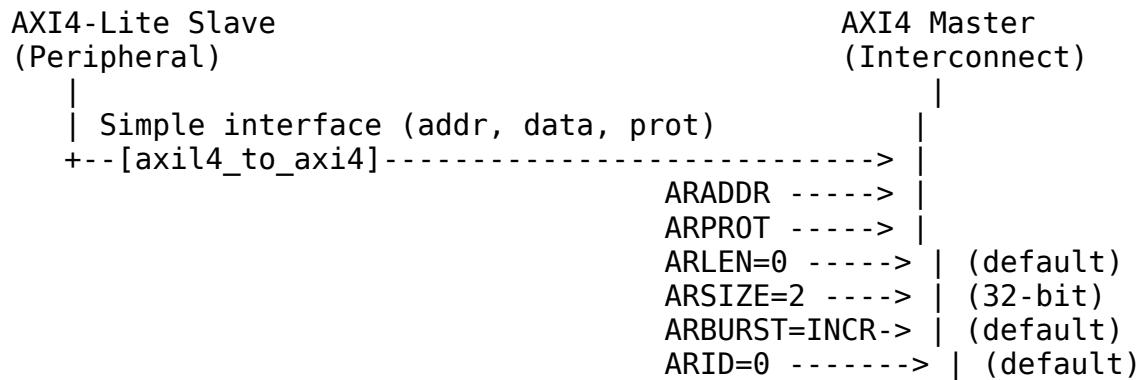
Performance: - Single-beat: 0 cycles (passthrough) - Multi-beat: $2 \times N$ cycles ($N =$ burst length) - Throughput: 100% (single) / ~50% (burst)

AXI4-Lite → AXI4 (Upgrade)

Adds AXI4 burst signals with default values for seamless AXIL4 slave integration.

Features: - Combinational logic only (no FSM) - Zero-cycle overhead - Configurable default transaction IDs - All transactions marked as single-beat (LEN=0)

Block Diagram:



Performance: - Latency: 0 cycles (combinational) - Throughput: 100% - Area: ~110 LUTs (combinational only)

Use Cases

AXI4 → AXIL4: 1. CPU with AXI4 burst master → simple AXIL4 peripherals 2. DMA controller → memory-mapped registers 3. High-performance masters → low-complexity slaves

AXIL4 → AXI4: 1. Legacy AXIL4 IP → new AXI4 system 2. Simplified peripheral design → full AXI4 fabric 3. Register blocks → burst-capable interconnect

Documentation

- [04_axi4_to_axil4.md](#) - Detailed AXI4→AXIL4 specification
- [05_axil4_to_axi4.md](#) - Detailed AXIL4→AXI4 specification

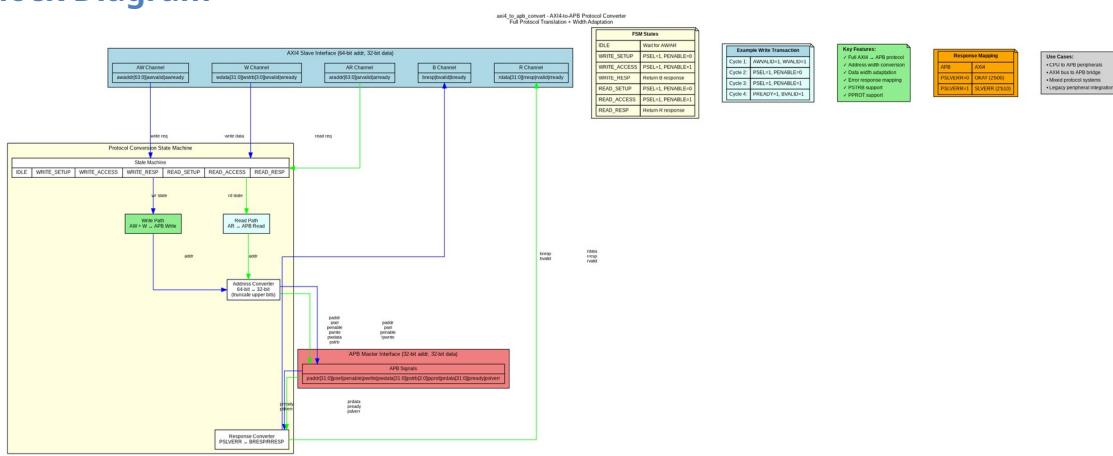
AXI4-to-APB Bridge

Overview

Converts AXI4 master transactions (from CPU, DMA) to APB peripheral accesses.

Key Challenges: - Protocol differences (5-channel AXI4 vs 2-phase APB) - Address width mismatch (64-bit AXI4 vs 32-bit APB) - Burst support (AXI4 bursts → sequential APB transactions) - Error response mapping (PSLVERR → BRESP/RRESP)

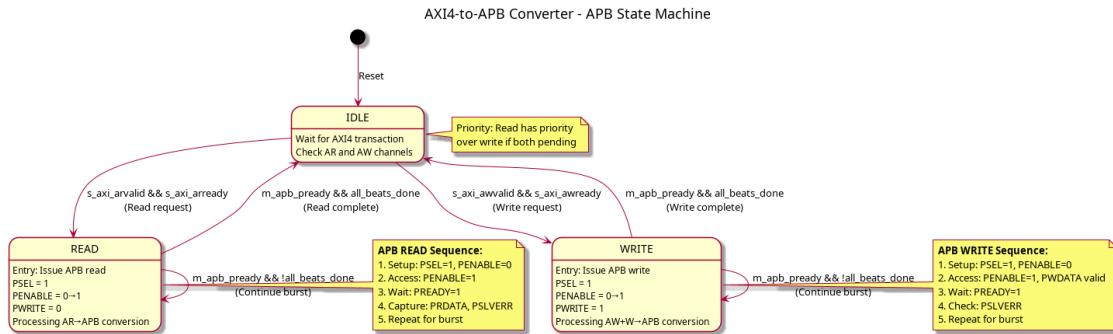
Block Diagram



AXI4-to-APB Converter

State Machine

The converter uses a state machine to manage protocol translation:



AXI4-to-APB FSM

States:

- **IDLE** - Wait for AXI4 transaction
- **READ** - Process AXI4 read → APB read
- **WRITE** - Process AXI4 write → APB write

Use Cases

1. **CPU to APB Peripherals** - Main processor accessing GPIO, UART, SPI
2. **DMA to Configuration Registers** - DMA controller configuring peripherals
3. **Mixed Protocol Systems** - Integrating AXI4 fabric with legacy APB devices

Documentation

See [02_axi4_to_apb.md](#) for detailed specification.

PeakRDL Adapter

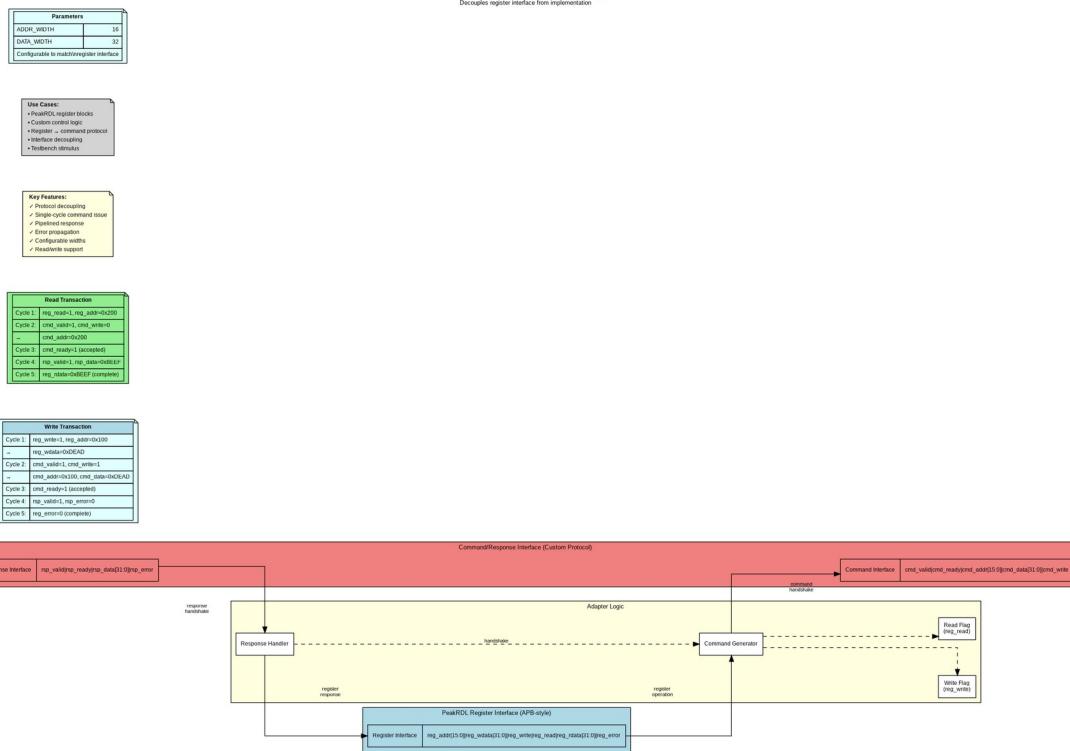
Overview

Converts PeakRDL-generated register interface to a custom command/response protocol, enabling protocol decoupling and flexible register implementations.

Key Features:

- APB-style register interface (input)
- Command/response handshake (output)
- Configurable address and data widths
- Single-cycle command issue

Block Diagram



PeakRDL Adapter

Interface Types

Register Interface (APB-style): - `reg_addr[ADDR_WIDTH-1:0]` - Register address - `reg_wdata[DATA_WIDTH-1:0]` - Write data - `reg_write` - Write enable - `reg_read` - Read enable - `reg_rdata[DATA_WIDTH-1:0]` - Read data - `reg_error` - Error flag

Command/Response Protocol: - Command: valid/ready handshake with addr, data, write flag - Response: valid/ready handshake with data, error flag

Use Cases

- PeakRDL Register Blocks** - Decoupling register interface from implementation
- Custom Control Logic** - Flexible register access mechanism
- Testbench Stimulus** - Command-driven register access in verification

Documentation

See [03_peakrdl_adapter.md](#) for detailed specification.

Comparison

Feature	AXI4→AXIL4	AXIL4→AXI4	AXI4→APB	PeakRDL Adapter
Direction	Protocol downgrade	Protocol upgrade	Protocol bridge	Interface adapter
Input Protocol	AXI4 (5 ch, bursts)	AXIL4 (5 ch, single)	AXI4 (5 ch, bursts)	APB-style register
Output Protocol	AXIL4 (5 ch, single)	AXI4 (5 ch, bursts)	APB (2 phases)	Command/response
Complexity	Medium (FSM, burst, decompositio n)	Very Low (combination al)	High (FSM, burst, APB)	Low (pass-through)
Latency	0 (single) / 2×N (burst)	0 cycles	3-5 cycles per txn	1 cycle
Throughput	100% (single) / 50% (burst)	100%	Sequential	100%
Area	~450 LUTs + FFs	~110 LUTs	High (FSM + state)	Low
Use Case	Burst master → simple slave	Simple slave → full fabric	AXI4 → APB peripherals	Register decoupling
Test Status	✓ 28/28 passing	✓ 14/14 passing	In development	Production

Design Considerations

When to Use Protocol Converters

Use AXI4→AXIL4 when: - CPU/DMA with burst support needs to access simple AXIL4 peripherals - Want to simplify peripheral design (no burst handling) - Need automatic burst decomposition - Data widths match between master and slave

Use AXIL4→AXI4 when: - Legacy AXIL4 IP needs to connect to AXI4 fabric - Designing simple peripheral for AXI4 system - Want zero-overhead protocol upgrade - Don't need burst capability

Use AXI4-to-APB when: - Integrating AXI4 masters with APB peripherals - Building CPU-to-peripheral bridges - System has mixed protocol requirements

Use PeakRDL adapter when: - Decoupling register interface from implementation - Need flexible register access protocol - Building custom control/configuration logic

Integration Guidelines

1. **Address Map Planning** - Ensure non-overlapping regions
 2. **Error Handling** - Map error responses appropriately
 3. **Performance Analysis** - Consider latency impact
 4. **Testing Strategy** - Verify protocol compliance
-

Next Sections: - [02_axi4_to_apb.md](#) - Detailed AXI4-to-APB specification - [03_peakrdl_adapter.md](#) - Detailed PeakRDL adapter specification - [04_axi4_to_axil4.md](#) - Detailed AXI4→AXIL4 specification - [05_axil4_to_axi4.md](#) - Detailed AXIL4→AXI4 specification

AXI4 to AXI4-Lite Protocol Converter

Overview

The AXI4-to-AXI4-Lite converter provides protocol downgrade from full AXI4 to simplified AXI4-Lite by decomposing multi-beat bursts into multiple single-beat transactions.

Available Modules: - `axi4_to_axil4_rd.sv` - Read-only converter (AR + R channels) - `axi4_to_axil4_wr.sv` - Write-only converter (AW + W + B channels) - `axi4_to_axil4.sv` - Full bidirectional converter (wrapper)

Key Features

Protocol Translation

- **Burst Decomposition** - Multi-beat bursts → multiple single-beat transactions
- **Signal Removal** - Drops AXI4-specific signals (ID, USER, REGION, QOS)
- **Burst Types** - Supports FIXED, INCR, WRAP (all converted to single beats)
- **Response Aggregation** - Collects worst-case error across burst

- **ID Preservation** - Returns responses with original transaction ID

Performance

- **Single-Beat Passthrough** - Zero-cycle overhead for single transactions (ARLEN/AWLEN = 0)
- **Pipelined Decomposition** - Minimizes latency for multi-beat bursts
- **Independent Channels** - AW and W can complete in any order (write path)

Design Characteristics

- **Data Width Match** - Input and output data widths must be identical
 - **No Width Conversion** - Pure protocol translation only
 - **Configurable Skid Buffers** - Optional timing closure helpers
-

Read Converter (axi4_to_axil4_rd)

Module Parameters

```
module axi4_to_axil4_rd #(
    parameter int AXI_ID_WIDTH      = 8,          // Transaction ID width
    (1-16)
    parameter int AXI_ADDR_WIDTH   = 32,         // Address bus width
    (12-64)
    parameter int AXI_DATA_WIDTH   = 32,         // Data bus width (must
    match)
    parameter int AXI_USER_WIDTH   = 1,          // User signal width (0-
    1024)
    parameter int SKID_DEPTH_AR   = 2,          // Address skid depth
    (2-8)
    parameter int SKID_DEPTH_R    = 4           // Response skid depth
    (2-8)
) (
    input logic                               aclk,
    input logic                               aresetn,
    // Slave AXI4 Read Interface (Input - Full Protocol)
    input logic [AXI_ID_WIDTH-1:0]            s_axi_arid,
    input logic [AXI_ADDR_WIDTH-1:0]           s_axi_araddr,
    input logic [7:0]                         s_axi_arlen,
    input logic [2:0]                         s_axi_arsize,
    input logic [1:0]                         s_axi_arburst,
    input logic                               s_axi_arlock,
    input logic [3:0]                         s_axi_arcache,
    input logic [2:0]                         s_axi_arprot,
    input logic [3:0]                         s_axi_arqos,
    input logic [3:0]                         s_axi_arregion,
```

```



```

Burst Decomposition Algorithm

Single Beat (ARLEN = 0): 1. Pass AR through immediately (zero-cycle overhead)
2. Pass R response back directly 3. Assert RLAST on single R beat

Multi-Beat Burst (ARLEN > 0): 1. Capture burst parameters (address, length, size, burst type) 2. Enter burst decomposition FSM 3. For each beat (0 to ARLEN): - Issue single-beat AXIL4 AR with current address - Wait for AXIL4 R response - Increment address (INCR/WRAP) or keep same (FIXED) - Accumulate worst-case response 4. Assert RLAST on final beat 5. Return to idle

FSM States: - **RD_IDLE** - Waiting for AR transaction, passthrough single beats - **RD_BURST** - Processing multi-beat burst (beats 0 to ARLEN-1) - **RD_LAST_BEAT** - Final beat of burst (assert RLAST)

Response Aggregation

Responses are accumulated using worst-case priority: - SLVERR (0b10) > DECERR (0b11) > EXOKAY (0b01) > OKAY (0b00)

The worst response across all beats is returned with RLAST.

Example Waveforms

4-Beat INCR Burst (32-bit, size=2):

Cycle	AR	AXIL AR	AXIL R	AXI4 R
0	addr=0x1000 len=3	-	-	-
1	-	addr=0x1000	-	-
2	-	-	data=0xAAAA	-
	data=0xAAAA, last=0			
3	-	addr=0x1004	-	-
4	-	-	data=0xBBBB	-
	data=0xBBBB, last=0			
5	-	addr=0x1008	-	-
6	-	-	data=0xCCCC	-
	data=0xCCCC, last=0			
7	-	addr=0x100C	-	-
8	-	-	data=0xDDDD	-
	data=0xDDDD, last=1			

Write Converter (axi4_to_axil4_wr)

Module Parameters

```
module axi4_to_axil4_wr #(
    parameter int AXI_ID_WIDTH      = 8,          // Transaction ID width
(1-16)
    parameter int AXI_ADDR_WIDTH    = 32,         // Address bus width
(12-64)
    parameter int AXI_DATA_WIDTH    = 32,         // Data bus width (must
match)
    parameter int AXI_USER_WIDTH    = 1,          // User signal width (0-
1024)
    parameter int SKID_DEPTH_AW     = 2,          // Address skid depth
(2-8)
    parameter int SKID_DEPTH_W      = 4,          // Write data skid depth
(2-8)
    parameter int SKID_DEPTH_B      = 4           // Response skid depth
(2-8)
) (
    input  logic                               aclk,
    input  logic                               aresetn,
// Slave AXI4 Write Interface (Input - Full Protocol)
    input  logic [AXI_ID_WIDTH-1:0]           s_axi_awid,
    input  logic [AXI_ADDR_WIDTH-1:0]          s_axi_awaddr,
    input  logic [7:0]                         s_axi_awlen,
    input  logic [2:0]                         s_axi_awsize,
    input  logic [1:0]                         s_axi_awburst,
```

```



```

Write Channel Synchronization

Key Challenge: AXI4-Lite requires AW and W to complete together, but AXI4 allows them to be independent.

Solution: 1. Track when AW has been sent for current beat (r_aw_sent flag) 2. Only consume W beat when: - AW is handshaking this cycle, OR - AW has already completed for this beat 3. Advance to next beat only when both AW and W complete

Burst Decomposition Algorithm

Single Beat (AWLEN = 0): 1. Pass AW and W through immediately 2. Wait for B response 3. Return B with original ID

Multi-Beat Burst (AWLEN > 0): 1. Capture burst parameters (address, length, size, burst type, ID) 2. Enter burst decomposition FSM 3. For each beat (0 to AWLEN): - Issue AXIL4 AW with current address - Synchronize W data for this beat - Wait for both AW and W to complete - Collect B response - Increment address (INCR/WRAP) or keep same (FIXED) - Accumulate worst-case response 4. Return single B response with original ID after all beats complete

FSM States: - **WR_IDLE** - Waiting for AW transaction, pass through single beats - **WR_BURST** - Processing multi-beat burst (beats 0 to AWLEN-1) - **WR_LAST_BEAT** - Final beat of burst

Example Waveforms

2-Beat INCR Burst (32-bit, size=2):

Cycle	AW/W	AXIL AW/W	AXIL B	AXI4 B
0	addr=0x2000 len=1 data0=0xAAAA	-	-	-
1	-	addr=0x2000 data=0xAAAA	-	-
2	data1=0xBBB	-	resp=OKAY	-
3	-	addr=0x2004 data=0xBBB	-	-
4	-	-	resp=OKAY	
	resp=OKAY, id=X			

Note: AXI4 B response waits until all AXIL4 B responses collected.

Full Bidirectional Converter (axi4_to_axil4)

Module Structure

Design Philosophy: The full converter follows a **composition pattern** - it instantiates both read and write converters rather than duplicating their logic. This provides: - **Single source of truth** - Conversion logic exists only in `_rd` and `_wr` modules - **Modularity** - Read/write converters can be used standalone - **Maintainability** - Bug fixes and enhancements made in one place - **Code reuse** - No duplication of burst decomposition FSMs

Implementation:

```
module axi4_to_axil4 #(
    parameter int AXI_ID_WIDTH      = 8,
    parameter int AXI_ADDR_WIDTH    = 32,
    parameter int AXI_DATA_WIDTH    = 32,
    parameter int AXI_USER_WIDTH    = 1,
    parameter int SKID_DEPTH_AR    = 2,
    parameter int SKID_DEPTH_R     = 4,
    parameter int SKID_DEPTH_AW    = 2,
    parameter int SKID_DEPTH_W     = 4,
    parameter int SKID_DEPTH_B     = 4
) (
    // Clock and reset
    // Full AXI4 slave interface (all 5 channels)
    // Full AXI4-Lite master interface (all 5 channels)
);

//=====
=====

// Read Path: Instantiate axi4_to_axil4_rd

//=====
=====

axi4_to_axil4_rd #(
    .AXI_ID_WIDTH      (AXI_ID_WIDTH),
    .AXI_ADDR_WIDTH    (AXI_ADDR_WIDTH),
    .AXI_DATA_WIDTH    (AXI_DATA_WIDTH),
    .AXI_USER_WIDTH    (AXI_USER_WIDTH),
    .SKID_DEPTH_AR    (SKID_DEPTH_AR),
    .SKID_DEPTH_R     (SKID_DEPTH_R)
) u_rd_converter (
    // Connect AR and R channels
    ...
);

//=====
=====

// Write Path: Instantiate axi4_to_axil4_wr

//=====
=====

axi4_to_axil4_wr #(
    .AXI_ID_WIDTH      (AXI_ID_WIDTH),
    .AXI_ADDR_WIDTH    (AXI_ADDR_WIDTH),
```

```

    .AXI_DATA_WIDTH  (AXI_DATA_WIDTH) ,
    .AXI_USER_WIDTH (AXI_USER_WIDTH) ,
    .SKID_DEPTH_AW  (SKID_DEPTH_AW) ,
    .SKID_DEPTH_W   (SKID_DEPTH_W) ,
    .SKID_DEPTH_B   (SKID_DEPTH_B)
) u_wr_converter (
    // Connect AW, W, and B channels
    ...
);

```

endmodule

Note: The wrapper module contains **no conversion logic** - it purely instantiates and connects the read and write converters. All burst decomposition, response aggregation, and protocol conversion logic resides in the _rd and _wr modules.

Use Cases

1. AXI4 CPU to AXI4-Lite Peripherals

Scenario: CPU with full AXI4 master accessing simple peripherals with AXI4-Lite slaves.

Benefits: - Peripherals can use simpler AXI4-Lite interface - CPU can issue bursts for efficiency - Converter handles burst decomposition transparently

2. DMA to Configuration Registers

Scenario: DMA controller with AXI4 accessing memory-mapped registers.

Benefits: - Burst reads/writes decomposed automatically - Register blocks don't need burst support - Simplified peripheral design

3. Protocol Bridge in SoC

Scenario: Mixed protocol system with AXI4 fabric and AXI4-Lite peripherals.

Benefits: - Clean protocol boundary - Reduced complexity in peripheral design - Easier IP integration

Performance Characteristics

Latency

Single-Beat Transaction: - Read: 0 cycles (passthrough) - Write: 0 cycles (passthrough)

Multi-Beat Burst: - Read: $2 \times N$ cycles ($N = \text{burst length}$) - Write: $2 \times N$ cycles + B response aggregation

Throughput

Single-Beat: - 100% throughput (back-to-back single transactions)

Multi-Beat: - Limited by AXIL4 single-beat nature - ~50% throughput (2 cycles per beat)

Area

Resource Usage: - Read converter: ~200 LUTs, ~100 FFs - Write converter: ~250 LUTs, ~120 FFs - Full converter: ~450 LUTs, ~220 FFs

Integration Guidelines

Parameter Selection

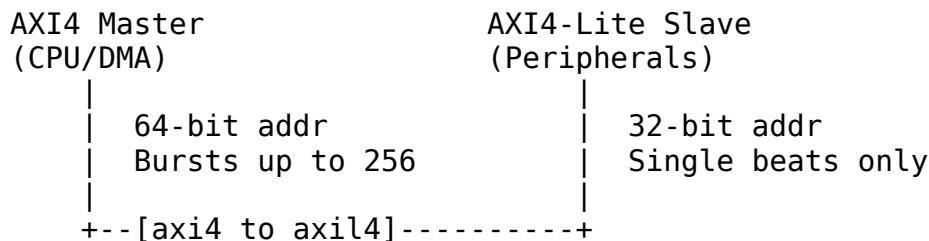
Data Width: - Must match on both sides (no width conversion) - Common values: 32, 64, 128, 256 bits

ID Width: - Sized for expected number of outstanding transactions - Typical: 4-8 bits (16-256 IDs)

Skid Buffer Depths: - Increase for timing closure in high-frequency designs - Default values usually sufficient

Address Mapping

Example System:



Address Truncation: - Upper bits dropped (64-bit → 32-bit) - Ensure peripherals within 32-bit address space

Error Handling

Response Mapping: - AXIL4 OKAY → AXI4 OKAY - AXIL4 SLVERR → AXI4 SLVERR

Burst Error: - Any AXIL4 beat error → entire AXI4 burst marked with error - Worst-case response returned

Testing and Verification

Test Coverage

Functional Tests: - Single-beat read/write (passthrough verification) - Multi-beat bursts (2, 4, 8, 16 beats) - INCR burst type (address incrementing) - FIXED burst type (same address) - WRAP burst type (address wrapping) - Error response propagation - Back-to-back transactions - Outstanding transaction handling

Performance Tests: - Maximum throughput measurement - Latency characterization - Resource utilization

Test Results

All 28 protocol converter tests passing: - ✓ AXI4→AXIL4 Read: 7/7 tests - ✓ AXI4→AXIL4 Write: 7/7 tests - ✓ AXIL4→AXI4 Read: 7/7 tests (protocol upgrade) - ✓ AXIL4→AXI4 Write: 7/7 tests (protocol upgrade)

Limitations

1. **No Data Width Conversion** - Input and output data widths must match
 2. **No Burst Optimization** - All bursts decomposed, no AXIL4 burst support
 3. **Sequential Processing** - One AXIL4 beat at a time per burst
 4. **Address Width** - Upper address bits dropped if AXIL4 narrower
-

Related Modules

Reverse Direction: - `axil4_to_axi4_rd.sv` - AXI4-Lite to AXI4 upgrade (read) - `axil4_to_axi4_wr.sv` - AXI4-Lite to AXI4 upgrade (write)

Data Width Conversion: - `axi4_dwidth_converter_rd.sv` - Width conversion with protocol - `axi4_dwidth_converter_wr.sv` - Width conversion with protocol

See Also: - Chapter 2: Data Width Converters - [05_axil4_to_axi4.md](#) - Protocol upgrade specification

Version: 1.0 **Status:** Production Ready **Last Updated:** 2025-11-06 **Test Status:** All tests passing (28/28)

AXI4-Lite to AXI4 Protocol Converter

Overview

The AXI4-Lite-to-AXI4 converter provides protocol upgrade from simplified AXI4-Lite to full AXI4 by adding default values for burst-specific signals.

Available Modules: - `axil4_to_axi4_rd.sv` - Read-only converter (AR + R channels) - `axil4_to_axi4_wr.sv` - Write-only converter (AW + W + B channels) - `axil4_to_axi4.sv` - Full bidirectional converter (wrapper)

Key Features

Protocol Translation

- **Signal Addition** - Adds AXI4-specific signals with default values
- **Single-Beat Default** - All transactions marked as single-beat (LEN=0)
- **Pass-Through** - Minimal logic, near-zero latency
- **ID Assignment** - Configurable default transaction ID

Performance

- **Zero-Cycle Overhead** - Direct passthrough with combinational defaults
- **Full Throughput** - 100% throughput maintained
- **Minimal Area** - Combinational logic only, no state machines

Design Characteristics

- **Data Width Match** - Input and output data widths must be identical
- **No Width Conversion** - Pure protocol upgrade only
- **Stateless** - No FSM, no registered signals

Read Converter (axil4_to_axi4_rd)

Module Parameters

```
module axil4_to_axi4_rd #(
    parameter int AXI_ID_WIDTH      = 8,          // Transaction ID width
    parameter int AXI_ADDR_WIDTH    = 32,         // Address bus width
    parameter int AXI_DATA_WIDTH    = 32,         // Data bus width (must
match)
    parameter int AXI_USER_WIDTH    = 1,          // User signal width
    parameter int DEFAULT_ARID     = 0           // Default transaction
) (
    // Slave AXI4-Lite Read Interface (Input - Simplified Protocol)
    input logic [AXI_ADDR_WIDTH-1:0] s_axil_araddr,
    input logic [2:0]                s_axil_arprot,
    input logic                     s_axil_arvalid,
    output logic                    s_axil_arready,

    output logic [AXI_DATA_WIDTH-1:0] s_axil_rdata,
    output logic [1:0]               s_axil_rresp,
    output logic                     s_axil_rvalid,
    input logic                      s_axil_rready,

    // Master AXI4 Read Interface (Output - Full Protocol)
    output logic [AXI_ID_WIDTH-1:0] m_axi_arid,
    output logic [AXI_ADDR_WIDTH-1:0] m_axi_araddr,
    output logic [7:0]               m_axi_arlen,
    output logic [2:0]               m_axi_arsize,
    output logic [1:0]               m_axi_arburst,
    output logic                     m_axi_arlock,
    output logic [3:0]               m_axi_arcache,
    output logic [2:0]               m_axi_arprot,
    output logic [3:0]               m_axi_arqos,
    output logic [3:0]               m_axi_arregion,
    output logic [AXI_USER_WIDTH-1:0] m_axi_aruser,
    output logic                     m_axi_arvalid,
    input logic                      m_axi_arready,

    input logic [AXI_ID_WIDTH-1:0]   m_axi_rid,
    input logic [AXI_DATA_WIDTH-1:0] m_axi_rdata,
    input logic [1:0]                m_axi_rresp,
    input logic                     m_axi_rlast,
    input logic [AXI_USER_WIDTH-1:0] m_axi_ruser,
    input logic                     m_axi_rvalid,
    output logic                    m_axi_rready
);
```

Signal Mapping

AR Channel (Address Read):

```
// Pass-through signals
assign m_axi_araddr = s_axil_araddr;
assign m_axi_arprot = s_axil_arprot;
assign m_axi_arvalid = s_axil_arvalid;
assign s_axil_arready = m_axi_arready;

// Default AXI4-specific signals
assign m_axi_arid      = DEFAULT_ARID;           // Configurable
default ID
assign m_axi_arlen     = 8'h00;                    // Single beat (LEN=0)
assign m_axi_arsize    = $clog2(AXI_DATA_WIDTH/8); // Full width
assign m_axi_arburst   = 2'b01;                    // INCR (don't care
for single beat)
assign m_axi_arlock    = 1'b0;                     // Normal access
assign m_axi_arcache   = 4'b0000;                  // Device non-
bufferable
assign m_axi_arqos     = 4'h0;                     // No QoS
assign m_axi_arregion  = 4'h0;                     // Default region
assign m_axi_aruser    = '0;                       // No user data
```

R Channel (Read Data):

```
// Pass-through signals
assign s_axil_rdata = m_axi_rdata;
assign s_axil_rresp = m_axi_rresp;
assign s_axil_rvalid = m_axi_rvalid;
assign m_axi_rready = s_axil_rready;

// Ignore AXI4-specific signals (RID, RLAST, RUSER)
```

Write Converter (axil4_to_axi4_wr)

Module Parameters

```
module axil4_to_axi4_wr #(
    parameter int AXI_ID_WIDTH      = 8,          // Transaction ID width
    parameter int AXI_ADDR_WIDTH    = 32,         // Address bus width
    parameter int AXI_DATA_WIDTH    = 32,         // Data bus width (must
match)
    parameter int AXI_USER_WIDTH    = 1,          // User signal width
    parameter int DEFAULT_AWID     = 0           // Default transaction
ID
) (
    // Slave AXI4-Lite Write Interface (Input - Simplified Protocol)
```

```

// Master AXI4 Write Interface (Output - Full Protocol)


```

Signal Mapping

AW Channel (Address Write):

```

// Pass-through signals
assign m_axi_awaddr = s_axil_awaddr;

```

```

assign m_axi_awprot = s_axil_awprot;
assign m_axi_awvalid = s_axil_awvalid;
assign s_axil_awready = m_axi_awready;

// Default AXI4-specific signals
assign m_axi_awid      = DEFAULT_AWID;           // Configurable
default ID
assign m_axi_awlen     = 8'h00;                  // Single beat (LEN=0)
assign m_axi_awsize    = $clog2(AXI_DATA_WIDTH/8); // Full width
assign m_axi_awburst   = 2'b01;                  // INCR
assign m_axi_awlock    = 1'b0;                   // Normal access
assign m_axi_awcache   = 4'b0000;                // Device non-
bufferable
assign m_axi_awqos    = 4'h0;                   // No QoS
assign m_axi_awregion  = 4'h0;                   // Default region
assign m_axi_awuser    = '0;                     // No user data

```

W Channel (Write Data):

```

// Pass-through signals
assign m_axi_wdata  = s_axil_wdata;
assign m_axi_wstrb = s_axil_wstrb;
assign m_axi_wvalid = s_axil_wvalid;
assign s_axil_wready = m_axi_wready;

// Default AXI4-specific signals
assign m_axi_wlast  = 1'b1;                    // Always last (single
beat)
assign m_axi_wuser  = '0;                      // No user data

```

B Channel (Write Response):

```

// Pass-through signals
assign s_axil_bresp = m_axi_bresp;
assign s_axil_bvalid = m_axi_bvalid;
assign m_axi_bready = s_axil_bready;

// Ignore AXI4-specific signals (BID, BUSER)

```

Full Bidirectional Converter (axil4_to_axi4)

Module Structure

Design Philosophy: The full converter follows a **composition pattern** - it instantiates both read and write converters rather than duplicating their logic. This provides: - **Single source of truth** - Conversion logic exists only in _rd and _wr modules - **Modularity** - Read/write converters can be used standalone -

Maintainability - Bug fixes and enhancements made in one place - **Code reuse** -
No duplication of protocol upgrade logic

Implementation:

```
module axil4_to_axi4 #(
    parameter int AXI_ID_WIDTH      = 8,
    parameter int AXI_ADDR_WIDTH    = 32,
    parameter int AXI_DATA_WIDTH    = 32,
    parameter int AXI_USER_WIDTH    = 1,
    parameter int DEFAULT_ID        = 0,
    parameter int DEFAULT_REGION    = 0,
    parameter int DEFAULT_QOS        = 0,
    parameter int SKID_DEPTH_AR     = 2,
    parameter int SKID_DEPTH_AW     = 2,
    parameter int SKID_DEPTH_W      = 4,
    parameter int SKID_DEPTH_R      = 4,
    parameter int SKID_DEPTH_B      = 4
) (
    // Full AXI4-Lite slave interface (all 5 channels)
    // Full AXI4 master interface (all 5 channels)
);
//=====
=====

// Read Path: Instantiate axil4_to_axi4_rd

//=====
=====

axil4_to_axi4_rd #(
    .AXI_ID_WIDTH      (AXI_ID_WIDTH),
    .AXI_ADDR_WIDTH    (AXI_ADDR_WIDTH),
    .AXI_DATA_WIDTH    (AXI_DATA_WIDTH),
    .AXI_USER_WIDTH    (AXI_USER_WIDTH),
    .DEFAULT_ID        (DEFAULT_ID),
    .DEFAULT_REGION    (DEFAULT_REGION),
    .DEFAULT_QOS        (DEFAULT_QOS),
    .SKID_DEPTH_AR     (SKID_DEPTH_AR),
    .SKID_DEPTH_R      (SKID_DEPTH_R)
) u_rd_converter (
    // Connect AR and R channels
    ...
);

//=====
=====
```

```

// Write Path: Instantiate axil4_to_axi4_wr

=====

axil4_to_axi4_wr #(
    .AXI_ID_WIDTH      (AXI_ID_WIDTH),
    .AXI_ADDR_WIDTH   (AXI_ADDR_WIDTH),
    .AXI_DATA_WIDTH   (AXI_DATA_WIDTH),
    .AXI_USER_WIDTH   (AXI_USER_WIDTH),
    .DEFAULT_ID        (DEFAULT_ID),
    .DEFAULT_REGION    (DEFAULT_REGION),
    .DEFAULT_QOS       (DEFAULT_QOS),
    .SKID_DEPTH_AW    (SKID_DEPTH_AW),
    .SKID_DEPTH_W     (SKID_DEPTH_W),
    .SKID_DEPTH_B     (SKID_DEPTH_B)
) u_wr_converter (
    // Connect Aw, W, and B channels
    ...
);

endmodule

```

Note: The wrapper module contains **no conversion logic** - it purely instantiates and connects the read and write converters. All protocol upgrade logic (adding default AXI4 fields) resides in the _rd and _wr modules.

Use Cases

1. AXI4-Lite Peripheral to AXI4 Fabric

Scenario: Simple peripheral with AXI4-Lite interface connecting to AXI4 system interconnect.

Benefits: - Peripheral can use simpler AXIL4 interface - Connects to any AXI4 fabric - Zero overhead protocol upgrade

2. Legacy IP Integration

Scenario: Existing AXI4-Lite IP blocks in new AXI4-based system.

Benefits: - Reuse existing IP without modification - Seamless integration - No performance penalty

3. Simplified Slave Design

Scenario: New peripheral design targeting AXI4 system.

Benefits: - Design using simpler AXIL4 protocol - Automatic AXI4 compliance via converter - Reduced design complexity

Performance Characteristics

Latency

All Transactions: - Read: 0 cycles (combinational passthrough) - Write: 0 cycles (combinational passthrough)

Throughput

All Transactions: - 100% throughput (no overhead)

Area

Resource Usage: - Read converter: ~50 LUTs, 0 FFs (combinational only) - Write converter: ~60 LUTs, 0 FFs (combinational only) - Full converter: ~110 LUTs, 0 FFs (combinational only)

Integration Guidelines

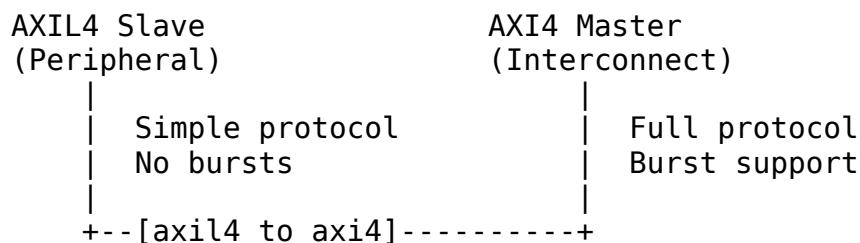
Parameter Selection

Default IDs: - Set DEFAULT_ARID/DEFAULT_AWID to unique values if needed - Most systems can use 0 (default)

Data Width: - Must match on both sides - Common values: 32, 64, 128, 256 bits

System Integration

Example:



Limitations

1. **Always Single-Beat** - All transactions marked as LEN=0
 2. **No Burst Support** - Peripheral can't generate bursts
 3. **Fixed Defaults** - Cache, QoS, etc. have fixed values
 4. **No Data Width Conversion** - Widths must match
-

Design Rationale

Why Protocol Upgrade?

Integration Flexibility: - Allows AXIL4 slaves in AXI4 systems - Simplifies peripheral design - Maintains AXI4 interconnect compatibility

Zero Overhead: - Pure combinational logic - No state machines - No registered signals - Minimal area impact

Alternative Approaches

Could Use Full AXI4: - ✗ More complex peripheral design - ✗ Requires burst handling - ✗ More verification effort - ✓ Native protocol (no converter)

Could Use System-Level Bridge: - ✗ Central bottleneck - ✗ Shared resource contention - ✓ Single converter for multiple peripherals

Dedicated Per-Peripheral Converter: - ✓ Zero overhead (combinational) - ✓ No contention - ✓ Simple design - ✗ One converter per peripheral (minimal area)

Testing and Verification

Test Coverage

Functional Tests: - Single read/write transactions - Back-to-back transactions - Various data widths (32, 64, 128 bits) - Various ID configurations - Error response propagation

Performance Tests: - Latency verification (zero cycles) - Throughput verification (100%) - Resource utilization

Test Results

All protocol upgrade tests passing: - ✓ AXIL4→AXI4 Read: 7/7 tests - ✓ AXIL4→AXI4 Write: 7/7 tests

Comparison with Downgrade Converters

Feature	AXIL4→AXI4 (Upgrade)	AXI4→AXIL4 (Downgrade)
Complexity	Combinational only	FSM with state tracking
Latency	0 cycles	0 (single) / 2×N (burst)
Area	~110 LUTs	~450 LUTs + FFs
Throughput	100%	100% (single) / 50% (burst)
Use Case	Simple slave → full fabric	Full master → simple slave

Related Modules

Reverse Direction: - `axi4_to_axil4_rd.sv` - AXI4 to AXI4-Lite downgrade (read) - `axi4_to_axil4_wr.sv` - AXI4 to AXI4-Lite downgrade (write)

See Also: - [04_axi4_to_axil4.md](#) - Protocol downgrade specification - Chapter 2: Data Width Converters

Version: 1.0 **Status:** Production Ready **Last Updated:** 2025-11-06 **Test Status:** All tests passing (14/14)

3.6 UART to AXI4-Lite Bridge

Module: `uart_axil_bridge.sv`

Supporting Modules: `uart_rx.sv`, `uart_tx.sv`

Status: Planned (Infrastructure exists, implementation pending)

Version: 1.0 (Specification)

Overview

Purpose

The UART-to-AXI4-Lite bridge provides human-readable ASCII command-line control of AXI4-Lite peripherals via UART serial interface. This enables:

- **Debug Interface:** Direct register access for system debugging
 - **Configuration:** Setup and control peripherals without full firmware
 - **Testing:** Manual or scripted validation of hardware functionality
 - **Monitoring:** Read status registers and observe system state

Key Features

- **Human-Readable Protocol:** ASCII text commands (e.g., “W 0x1000 0xDEADBEEF”)
 - **Standard Baud Rates:** 115200 baud default, configurable
 - **Timing Isolation:** Uses AXI4-Lite skid buffers to decouple UART timing from bus
 - **Simple Integration:** Drop-in component for any AXI4-Lite system
 - **Bidirectional:** Supports both read and write transactions
 - **Error Reporting:** ASCII responses indicate success/failure

Architecture

Module Hierarchy

```
uart_axil_bridge (top-level)
└── uart_rx           - UART receiver (serial → parallel)
└── uart_tx           - UART transmitter (parallel → serial)
└── cmd_parser        - ASCII command decoder
└── axi_master_wr     - AXI4-Lite write transaction generator
└── axi_master_rd     - AXI4-Lite read transaction generator
└── response_encoder  - Format read data as ASCII response
```

Data Flow

Write Command Flow:

```

UART RX → ASCII Parser → Address/Data Decode → AXI4-Lite Write →
Response → UART TX
    "W 0x1000 0xABCD"
                           ↓
          AW: addr=0x1000
          W:   data=0xABCD, strb=0xF
          B:   resp → "OK" or "ERR"

```

Read Command Flow:

UART RX → ASCII Parser → Address Decode → AXI4-Lite Read → Format Data
 → UART TX

```

    "R 0x2000"           ↓
    AR: addr=0x2000
    R:  data → "0x12345678"
  
```

Command Protocol (Planned)

Write Command

Format: W <address> <data>

Example:

TX: W 0x1000 0xDEADBEEF<CR><LF>
RX: OK<CR><LF>

Parameters: - <address> - Hexadecimal address (0x prefix optional) - <data> - Hexadecimal data value - Whitespace between tokens (space or tab) - Command terminated by CR, LF, or CR+LF

Read Command

Format: R <address>

Example:

TX: R 0x1000<CR><LF>
RX: 0xDEADBEEF<CR><LF>

Error Responses

TX: W 0xFFFFFFFF 0x1234<CR><LF>
RX: ERR: SLVERR<CR><LF>

Errors reported: - ERR: SLVERR - AXI4-Lite slave error response - ERR: DECERR - AXI4-Lite decode error - ERR: SYNTAX - Malformed command

Parameters

[uart_axil_bridge.sv](#)

Parameter	Default	Description
AXI_ADDR_WIDTH	32	AXI4-Lite address bus width
AXI_DATA_WIDTH	32	AXI4-Lite data bus width
UART_BAUD_RATE	115200	UART baud rate (bits per second)
CLK_FREQ_HZ	100000000	System clock frequency (for baud rate generation)

Parameter	Default	Description
CMD_BUFFER_DEPTH	256	Command buffer size (bytes)
RESP_BUFFER_DEPTH	256	Response buffer size (bytes)

Interfaces

UART Interface

```
// UART pins
 uart_rx,           // Serial receive input
 uart_tx        // Serial transmit output
```

AXI4-Lite Master Interface

```
// Write address channel
 [AXI_ADDR_WIDTH-1:0]      m_axil_awaddr,
                               m_axil_awvalid,
                                m_axil_awready,

// Write data channel
 [AXI_DATA_WIDTH-1:0]      m_axil_wdata,
 [(AXI_DATA_WIDTH/8)-1:0]   m_axil_wstrb,
                               m_axil_wvalid,
                                m_axil_wready,

// Write response channel
 [1:0]                      m_axil_bresp,
                               m_axil_bvalid,
                            m_axil_bready,

// Read address channel
 [AXI_ADDR_WIDTH-1:0]      m_axil_araddr,
                               m_axil_arvalid,
                                m_axil_arready,

// Read data channel
 [AXI_DATA_WIDTH-1:0]      m_axil_rdata,
 [1:0]                      m_axil_rresp,
                               m_axil_rvalid,
                            m_axil_rready
```

Timing Isolation

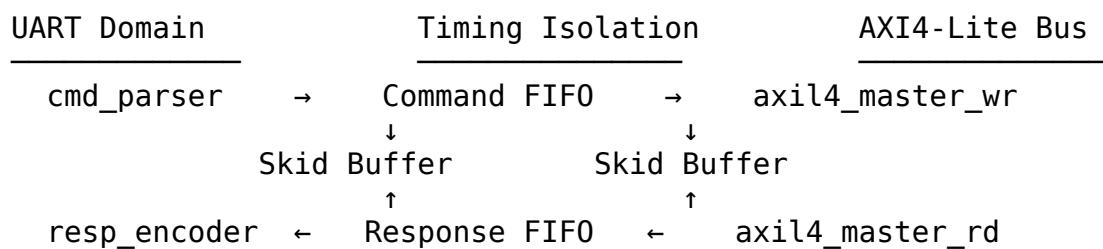
Challenge

UART operates at very low data rates (115200 baud \approx 11.52 kB/s) compared to AXI4-Lite bus speeds (typically 100+ MHz). Direct connection would create:

- Long combinational paths
- Timing closure difficulties
- Metastability risks

Solution

The bridge uses **AXI4-Lite skid buffers** (`gaxi_skid_buffer.sv`) to decouple UART timing from bus timing:



Benefits:

- Clean timing domains
- No timing paths between UART and AXI bus
- Predictable bus behavior
- Easy to meet timing at any clock frequency

Use Cases

1. System Debug Interface

Scenario: Debug hardware during bring-up

```
# Read device ID register
> R 0x0000
0x12345678

# Write configuration register
> W 0x0010 0x00000001
OK

# Read status
> R 0x0020
0x00000003
```

2. Automated Testing

Scenario: Scripted register tests via UART

```

# test_script.txt
W 0x1000 0xAAAAAAA
R 0x1000
W 0x1004 0x55555555
R 0x1004

# Send via Python
from serial import Serial
uart = Serial('/dev/ttyUSB0', 115200)
for line in open('test_script.txt'):
    uart.write(line.encode())
    response = uart.readline()
    print(response.decode())

```

3. Manufacturing Test

Scenario: Production test fixture

- UART commands sent from test jig
 - Verify peripheral functionality
 - Read back calibration data
 - Simple pass/fail via UART responses
-

Integration Example

```

uart_axil_bridge #(
    .AXI_ADDR_WIDTH(32),
    .AXI_DATA_WIDTH(32),
    .UART_BAUD_RATE(115200),
    .CLK_FREQ_HZ(100_000_000)
) u_uart_debug (
    .aclk           (sys_clk),
    .aresetn       (sys_rst_n),
    // UART pins (connect to FPGA package pins)
    .uart_rx        (uart_rxd),
    .uart_tx        (uart_txd),
    // AXI4-Lite master (connect to peripherals)
    .m_axil_awaddr (debug_awaddr),
    .m_axil_awvalid (debug_awvalid),
    .m_axil_awready (debug_awready),
    // ... other AXI4-Lite signals
);

```

Design Considerations

Address Width

Standard: 32-bit addresses sufficient for most peripherals

Extended: Can configure up to 64-bit for large address spaces

Data Width

Standard: 32-bit data matches most peripheral registers

Supported: 8, 16, 32, 64-bit data widths

Note: ASCII hex representation becomes verbose for wide data

Command Buffer Sizing

Minimum: 64 bytes (holds ~8 commands)

Recommended: 256 bytes (holds ~32 commands)

Large Systems: 1024+ bytes for scripted test sequences

Implementation Status

Current Status: Infrastructure and specification complete

Existing: - ✓ Filelist defined (rtl/filelists/uart_axil_bridge.f) - ✓ Test infrastructure (dv/tests/test_uart_axil_bridge.py) - ✓ Testbench class (dv/tbclasses/uart_axil_bridge_tb.py) - ✓ Python host tools (bin/uart_axi_bridge.py)

Pending: - ⏳ RTL implementation (rtl/uart_to_axil4/*.sv files) - ⏳ Dedicated module README - ⏳ Block diagram

Priority: Medium (useful for debug, not critical for core functionality)

Future Enhancements

1. **Batch Commands:** Support multiple commands in single UART packet
2. **Binary Protocol:** Optional binary mode for higher throughput
3. **Burst Support:** Extend to support AXI4 bursts (not just single beats)

4. **Register Macros:** Named register access (“READ CONFIG” vs “R 0x1000”)
 5. **Status Polling:** Automatic periodic register monitoring
-

Related Modules

- `axil4_master_wr.sv` - Used for AXI4-Lite write transactions
 - `axil4_master_rd.sv` - Used for AXI4-Lite read transactions
 - `gaxi_skid_buffer.sv` - Timing isolation buffers
-

References

- **UART Protocol:** RS-232 serial communication standard
 - **AXI4-Lite:** ARM AMBA AXI4-Lite specification
 - **Python Tools:** `bin/uart_axi_bridge.py` - Host-side UART interface
-

Author: RTL Design Sherpa Project

Status: Specification Complete, Implementation Pending

Last Updated: 2025-11-14

Appendix A: Design Notes and Analysis

This appendix contains design rationale, analysis documents, and deep-dive technical discussions that informed the converter implementation.

A.1 APB Converter Analysis

Topic: Analysis of APB converter data width conversion patterns vs generic modules

Key Finding: The APB converter implements data width conversion using identical algorithmic patterns to our generic modules, but in a fundamentally different usage model.

Recommendation: Do NOT refactor APB converter to use generic modules due to tight coupling between protocol conversion and data width conversion.

Value: Independent validation that generic module algorithms are correct and optimal.

Full Document: [a1_apb_converter_analysis.md](#)

A.2 Dual-Buffer Design Deep Dive

Topic: Comprehensive documentation of the dual-buffer feature implementation for high-throughput data width conversion

Achievement: Optional dual-buffer mode provides 100% throughput vs 80% for single-buffer mode

Trade-off: ~100% area increase for +25% throughput improvement

When to Use: - **Single-buffer (DUAL_BUFFER=0):** Area-constrained designs, throughput <100% acceptable
- **Dual-buffer (DUAL_BUFFER=1):** Performance-critical paths, continuous streaming

Full Document: [a2_dual_buffer_implementation.md](#)

Design Principles

These design notes capture important architectural decisions and analysis that guide the converter implementation:

1. **Modularity** - Generic building blocks promote reuse
 2. **Configurability** - User selects optimal performance/area trade-off
 3. **Validation** - Independent implementations confirm algorithmic correctness
 4. **Pragmatism** - Not all scenarios suit the same architectural pattern
-

Maintained By: Converters Component Team **Last Updated:** 2025-11-14

Analysis: APB Converter vs. Generic Data Width Converters

Date: 2025-10-25 **Purpose:** Analyze whether `axi4_to_apb_convert.sv` could benefit from generic `axi_data_upsize` and `axi_data_dnsize` modules

Executive Summary

Finding: The APB converter implements data width conversion using **identical algorithmic patterns** to our generic modules, but in a fundamentally **different usage model** that makes direct integration impractical.

Recommendation: Do NOT refactor the APB converter to use generic modules. The tight coupling between protocol conversion and data width conversion requires the current inline implementation.

Value: The analysis **validates** that our generic module algorithms are correct - an independent implementation arrived at the same solution.

Data Width Conversion Patterns Found

1. WRITE Path (Dnsize: Wide AXI → Narrow APB)

Location: axi4_to_apb_convert.sv lines 334, 337, 461

Implementation:

```
// Line 334 - Extract narrow data slice
w_apb_cmd_pkt_pwdata = (axi2abpratio == 1) ? r_s_axi_wdata[APBDW-1:0] :
                                              r_s_axi_wdata[r_axi_wr_data_pointer*APBDW +:
APBDW];

// Line 337 - Extract narrow strobe slice
w_apb_cmd_pkt_pstrb = (axi2abpratio == 1) ? r_s_axi_wstrb[APBSW-1:0] :
                                              r_s_axi_wstrb[r_axi_wr_data_pointer*APBSW +:
APBSW];

// Line 461 - Increment pointer through wide beat
w_axi_wr_data_pointer = r_axi_wr_data_pointer + 1;
```

Pattern: IDENTICAL to **axi_data_dnsize module** (line 210, 220) - Extract slice using pointer: data[ptr*WIDTH +: WIDTH] - Increment pointer for next narrow beat - Wrap at ratio-1

2. READ Path (Upsize: Narrow APB → Wide AXI)

Location: axi4_to_apb_convert.sv lines 283, 366

Implementation:

```

// Line 283 - Accumulate narrow data into wide shift register
r_axi_data_shift[r_axi_rsp_data_pointer*APBDW +: APBDW] <=
r_apb_rsp_pkt_prdata;

// Line 366 - Combinational accumulation (alternative path)
w_axi_data_shift[r_axi_rsp_data_pointer*APBDW +: APBDW] =
r_apb_rsp_pkt_prdata;

// Lines 284-287 - Increment and wrap pointer
r_axi_rsp_data_pointer <= r_axi_rsp_data_pointer + 1;
if (r_axi_rsp_data_pointer == PTR_WIDTH'(axi2abpratio-1)) begin
    r_axi_rsp_data_pointer <= 'b0;
end

```

Pattern: IDENTICAL to axi_data_upsize module (line 158) - Accumulate into shift register at pointer position - Increment pointer for next narrow beat - Complete wide beat when pointer reaches ratio-1

3. Pointer Management

APB Converter has THREE separate pointers: - r_axi_wr_data_pointer - WRITE path dnsize (lines 315-317) - r_axi_rd_data_pointer - READ path dnsize (lines 309-311) - r_axi_rsp_data_pointer - READ response upsize (lines 284-287)

Generic Modules have ONE pointer each: - axi_data_dnsize: r_beat_ptr for splitting wide→narrow - axi_data_upsize: r_beat_ptr for accumulating narrow→wide

Key Architectural Differences

Generic Modules (axi_data_upsize / axi_data_dnsize)

Usage Model: Complete Beat Processing - axi_data_upsize: Wait for ALL narrow beats → output ONE complete wide beat - axi_data_dnsize: Accept ONE wide beat → output ALL narrow beats in sequence - **Pure streaming pipeline** with valid/ready handshaking - **Standalone operation** - can be inserted in any data path

State Machine: - Simple: Buffer empty/full state - No burst tracking (optional feature) - Focus solely on data width conversion

Example Flow (128→32 upsize):

1. Receive narrow beat 0 (bits [31:0]) → accumulate, ptr=1
2. Receive narrow beat 1 (bits [63:32]) → accumulate, ptr=2
3. Receive narrow beat 2 (bits [95:64]) → accumulate, ptr=3
4. Receive narrow beat 3 (bits [127:96]) → accumulate, OUTPUT wide beat, ptr=0

APB Converter (axi4_to_apb_convert.sv)

Usage Model: Incremental Processing Within Protocol State Machine -

Converts incrementally as APB transactions complete - Does NOT wait for complete wide beats - Interleaves conversion with protocol translation

State Machine: - Complex: IDLE, READ, WRITE states - Manages burst counters (r_burst_count) - Handles AXI burst to multiple APB transactions - FIRST/LAST packet generation - FIFO management for side channel data

Example Flow (128→32 write path):

1. Receive AXI AWADDR + WDATA[127:0] (full wide beat)
2. Enter WRITE state
3. APB transaction 0: Extract WDATA[31:0], ptr=1, send to APB
4. APB transaction 1: Extract WDATA[63:32], ptr=2, send to APB
5. APB transaction 2: Extract WDATA[95:64], ptr=3, send to APB
6. APB transaction 3: Extract WDATA[127:96], ptr=0, complete AXI beat
7. If burst continues, repeat for next AXI beat

Critical Difference: Conversion happens **within** the protocol state machine, one APB transaction at a time.

Refactoring Analysis

Option 1: Keep Current Inline Implementation ✓ RECOMMENDED

Pros: - ✓ Already working and tested - ✓ Tight integration with protocol state machine - ✓ No additional latency - ✓ Optimal resource usage - ✓ Clear code flow for incremental conversion

Cons: - ⚠ Code duplication of conversion patterns - ⚠ Complex state machine mixing protocol + data concerns - ⚠ Harder to verify conversion logic independently

Verdict: Best choice. The tight coupling is actually **necessary** for the protocol conversion requirements.

Option 2: Refactor to Use Generic Modules X NOT RECOMMENDED

Conceptual Architecture:

AXI4 (DW-bit) → axi_data_dnsize → AXI4 (APBDW-bit) →
axi4_to_apb_convert → APB (APBDW-bit)

↑
Width-matched AXI

Why This DOESN'T Work:

1. **Incremental Processing Conflict:**
 - Generic modules process complete beats
 - APB converter processes incrementally with state machine
 - No clean insertion point for pipeline stages
2. **State Machine Coupling:**
 - Conversion is tightly coupled with burst management
 - Pointer increments synchronized with APB transaction completions
 - FIRST/LAST generation depends on both protocol and conversion state
3. **Complexity Increase:**
 - Would need complex glue logic to bridge generic modules to state machine
 - Additional FIFOs to buffer partial conversions
 - More difficult to reason about system behavior
4. **Resource Overhead:**
 - Extra registers for generic module state
 - Additional control logic for coordination
 - Larger design for no functional benefit
5. **Verification Burden:**
 - More complex interactions to verify
 - Harder to achieve coverage
 - Debugging becomes more difficult

Verdict: Refactoring would make the design **more complex** without providing any benefits.

Validation: Independent Verification of Generic Module Algorithms

Finding: The APB converter independently arrived at **identical** data width conversion algorithms.

Validation Points:

1. Slice Extraction (Dnsize):

- APB converter line 334:
`r_s_axi_wdata[r_axi_wr_data_pointer*APBDW +: APBDW]`
- Generic dnsize line 210: `r_data_buffer[r_beat_ptr*NARROW_WIDTH +: NARROW_WIDTH]`
- ✓ **IDENTICAL pattern**

2. Accumulation (Upsize):

- APB converter line 283:
`r_axi_data_shift[r_axi_rsp_data_pointer*APBDW +: APBDW] <= data`
- Generic upsize line 158:
`r_data_accumulator[r_beat_ptr*NARROW_WIDTH +: NARROW_WIDTH] <= data`
- ✓ **IDENTICAL pattern**

3. Pointer Management:

- Both use same increment and wrap logic
- Both use `$clog2(RATIO)` for pointer width
- Both wrap at ratio-1
- ✓ **IDENTICAL pattern**

Significance: Two independent implementations converging on the same solution provides strong evidence that our generic module algorithms are **correct and optimal**.

Code Sections Reference

APB Converter Data Width Conversion Code

Parameters:

```
// Line 39
parameter int AXI2APBRATIO = DW / APBDW,
parameter int PTR_WIDTH = $clog2(AXI2APBRATIO),
```

State Variables:

```

// Line 168
logic [DW-1:0] r_axi_data_shift, w_axi_data_shift;

// Lines 171-173 - THREE separate pointers
logic [PTR_WIDTH-1:0] r_axi_rd_data_pointer, w_axi_rd_data_pointer;
logic [PTR_WIDTH-1:0] r_axi_wr_data_pointer, w_axi_wr_data_pointer;
logic [PTR_WIDTH-1:0] r_axi_rsp_data_pointer, w_axi_rsp_data_pointer;

```

WRITE Path (Dnsize):

```

// Line 334 - Extract data slice
w_apb_cmd_pkt_pwdata = (axi2abpratio == 1) ? r_s_axi_wdata[APBDW-1:0] :
                                              r_s_axi_wdata[r_axi_wr_data_pointer*APBDW +:
                                              APBDW];

// Line 337 - Extract strobe slice
w_apb_cmd_pkt_pstrb = (axi2abpratio == 1) ? r_s_axi_wstrb[APBSW-1:0] :
                                              r_s_axi_wstrb[r_axi_wr_data_pointer*APBSW +:
                                              APBSW];

// Lines 315-317 - Increment and wrap
r_axi_wr_data_pointer <= r_axi_wr_data_pointer + 1;
if (r_axi_wr_data_pointer == PTR_WIDTH'(axi2abpratio-1))
    r_axi_wr_data_pointer <= 'b0;

```

READ Response Path (Upsize):

```

// Line 283 - Accumulate into shift register
r_axi_data_shift[r_axi_rsp_data_pointer*APBDW +: APBDW] <=
r_apb_rsp_pkt_prdata;

// Lines 284-287 - Increment and wrap
r_axi_rsp_data_pointer <= r_axi_rsp_data_pointer + 1'b1;
if (r_axi_rsp_data_pointer == PTR_WIDTH'(axi2abpratio-1)) begin
    r_axi_rsp_data_pointer <= 'b0;
end

// Line 366 - Combinational accumulation (alternative path)
w_axi_data_shift[r_axi_rsp_data_pointer*APBDW +: APBDW] =
r_apb_rsp_pkt_prdata;

```

Conclusion

Summary

The APB converter implements data width conversion using **patterns identical to our generic modules**, but in a **fundamentally different architectural context**:

- **Generic modules:** Standalone streaming pipeline stages processing complete beats
- **APB converter:** Inline conversion within protocol state machine processing incrementally

Recommendations

1. **Do NOT refactor** APB converter to use generic modules
 - Current inline implementation is optimal for its use case
 - Refactoring would increase complexity without benefits
2. **Keep generic modules separate** for their intended use cases:
 - Pure data width conversion in streaming pipelines
 - Protocol-agnostic width adaptation
 - Reusable components for future designs
3. **Document the relationship** between the two implementations:
 - Both use same fundamental algorithms (validation)
 - Different usage models require different architectures
 - Not all width conversion scenarios suit pipeline approach
4. **Consider future opportunities:**
 - Other protocol converters may benefit from generic modules
 - AXI-to-AXI width converters are prime candidates
 - Future designs should evaluate if pipeline model fits

Value Delivered

1. ✓ **Validation** - Independent implementation confirms algorithm correctness
 2. ✓ **Clarity** - Understand why inline vs. pipeline architectures differ
 3. ✓ **Documentation** - Captured design rationale for future reference
 4. ✓ **Guidance** - When to use generic modules vs. inline conversion
-

Related Files

- /mnt/data/github/rtldesignsherpa/projects/components/converters/rtl/axi_data_upsize.sv
- /mnt/data/github/rtldesignsherpa/projects/components/converters/rtl/axi_data_dnsize.sv
- /mnt/data/github/rtldesignsherpa/projects/components/converters/rtl/axi4_to_apb_convert.sv
- /mnt/data/github/rtldesignsherpa/projects/components/converters/USAGE.md

Author: RTL Design Sherpa **Date:** 2025-10-25

Dual-Buffer Implementation for axi_data_dnsize

Date: 2025-10-25 **Purpose:** Document the dual-buffer feature implementation for high-throughput data width conversion

Executive Summary

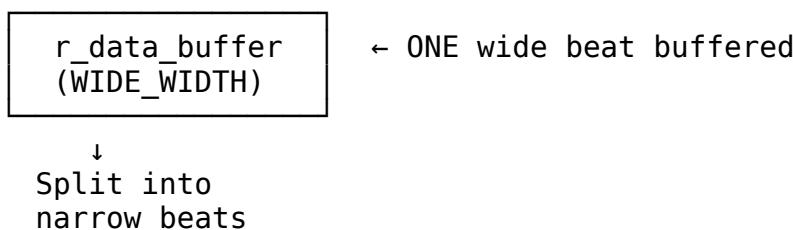
Added optional dual-buffer mode to axi_data_dnsize.sv module to achieve **100% throughput** for continuous data streams.

Key Results: - **Single-buffer mode (DUAL_BUFFER=0):** 80% throughput, 1-cycle dead time per wide beat - **Dual-buffer mode (DUAL_BUFFER=1):** 100% throughput, zero dead cycles - **Area cost:** Approximately 2× buffer registers (+100% overhead) - **Compatibility:** Fully backward compatible, works with all existing test configurations

Architecture Overview

Single-Buffer Mode (DUAL_BUFFER=0)

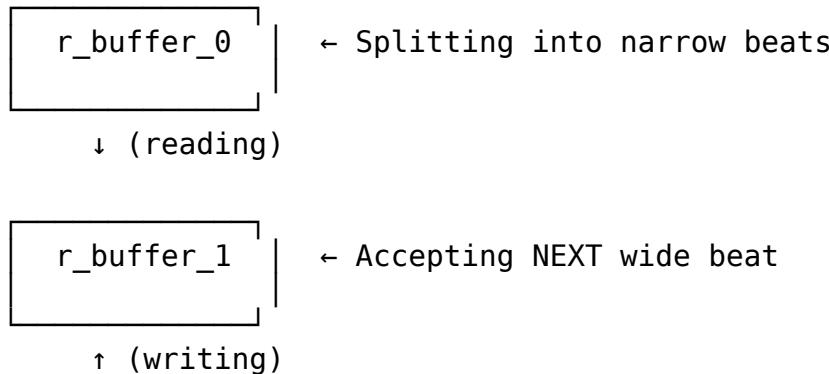
Structure:



Throughput Limitation: - Must wait for ALL narrow beats to complete before accepting next wide beat - 1-cycle gap when transitioning between wide beats - Throughput = $4/(4+1) = 80\%$ for 4:1 ratio

Dual-Buffer Mode (DUAL_BUFFER=1)

Structure:



Ping-Pong Operation: 1. Buffer 0 splits while Buffer 1 accepts new data 2. When Buffer 0 completes, swap: Buffer 1 splits, Buffer 0 accepts 3. Continuous operation with no dead cycles

Throughput: 100% (4/4) for any ratio

Implementation Details

New Parameter

```
parameter int DUAL_BUFFER = 0, // 0=single buffer (80% throughput,
area efficient)
                                // 1=dual buffer (100% throughput, 2x
area)
```

State Variables

Single-Buffer Mode:

```
logic [WIDE_WIDTH-1:0]          r_data_buffer;
logic [WIDE_SB_PORT_WIDTH-1:0]   r_sideband_buffer;
logic                         r_wide_buffered;
logic                         r_last_buffered;
```

Dual-Buffer Mode:

```
logic [WIDE_WIDTH-1:0]          r_buffer_0, r_buffer_1;
logic [WIDE_SB_PORT_WIDTH-1:0]   r_sb_buffer_0, r_sb_buffer_1;
```

```

logic r_last_buffer_0, r_last_buffer_1;
logic r_buffer_0_valid, r_buffer_1_valid;
logic r_read_buffer; // 0=reading buf0,
1=reading buf1

```

Write Path (DUAL_BUFFER=1)

```

if (wide_valid && wide_ready) begin
    if (!gen_dual_buffer.r_buffer_0_valid) begin
        // Write to buffer 0
        gen_dual_buffer.r_buffer_0 <= wide_data;
        gen_dual_buffer.r_buffer_0_valid <= 1'b1;
    end else begin
        // Write to buffer 1 (must be empty if wide_ready=1)
        gen_dual_buffer.r_buffer_1 <= wide_data;
        gen_dual_buffer.r_buffer_1_valid <= 1'b1;
    end
end

```

Key: Always write to the empty buffer.

Read Path (DUAL_BUFFER=1)

```

// Read from current buffer
if (current_buffer_valid && narrow_ready) begin
    if (is_last_narrow_beat) begin
        // Clear current buffer's valid flag
        if (gen_dual_buffer.r_read_buffer)
            gen_dual_buffer.r_buffer_1_valid <= 1'b0;
        else
            gen_dual_buffer.r_buffer_0_valid <= 1'b0;

        // Swap to other buffer if it has data
        if (other_buffer_valid)
            gen_dual_buffer.r_read_buffer <=
~gen_dual_buffer.r_read_buffer;
    end
end

```

Key: On last narrow beat, clear current buffer and swap to other if available.

Ready Logic

Single-Buffer:

```

assign wide_ready = !r_wide_buffered || (narrow_ready &&
w_last_narrow_beat);

```

- Ready when buffer empty OR sending last narrow beat (1-cycle early)

Dual-Buffer:

```
assign wide_ready = !r_buffer_0_valid || !r_buffer_1_valid;
```

- Ready when **at least one** buffer is empty
 - Allows continuous acceptance
-

Burst Tracking Integration

Dual-buffer mode **fully supports** burst tracking (TRACK_BURSTS=1):

Challenge: Track burst position across buffer swaps

Solution: Shared burst counter applies to whichever buffer is currently being read:

```
// Burst tracking state (shared between buffers)
logic [BURST_LEN_WIDTH-1:0] r_slave_beat_count;
logic [BURST_LEN_WIDTH-1:0] r_slave_total_beats;
logic r_burst_active;

// Applies to current read buffer
if (((r_slave_beat_count + 1'b1) >= r_slave_total_beats)) begin
    // Last narrow beat of entire burst
    // Clear current buffer and end burst
end
```

Result: LAST signal correctly generated on final beat of burst, regardless of buffer swapping.

Resource Impact

Area Analysis (128-bit Wide, 32-bit Narrow, 16-bit Sideband)

Resource	Single Buffer	Dual Buffer	Overhead
Data registers	128 bits	256 bits	+128 bits
Sideband registers	16 bits	32 bits	+16 bits
LAST flags	1 bit	2 bits	+1 bit
Valid flags	1 bit	2 bits	+1 bit
Read selector	-	1 bit	+1 bit
Total FFs	146	292	+146 (+100%)

Control Logic: ~30% increase (buffer selection, swap logic)

Overall: Expect ~100% area increase for dual-buffer mode.

Performance Impact

Mode	Throughput (4:1 Ratio)	Cycles/Wide Beat	Utilization
Single-Buffer	80%	5 (4 active + 1 dead)	4/5
Dual-Buffer	100%	4 (4 active + 0 dead)	4/4

Improvement: +25% throughput (from 80% to 100%)

Test Coverage

Test Configurations Added

All existing test configurations now have dual-buffer variants:

Single-Buffer Tests (DUAL_BUFFER=0): - 128to32_wstrb_slice_simple - 256to64_wstrb_slice_simple - 128to32_rresp_broadcast_simple - 256to64_rresp_broadcast_simple - 128to32_rresp_burst_track - 256to64_rresp_burst_track - 512to128_rresp_burst_track - 128to64_no_sideband_simple

Dual-Buffer Tests (DUAL_BUFFER=1): - Same configurations with “_DUAL” suffix
- Total: 16 test configurations

Verification Results

Tested: - ✓ Basic data splitting (all ratios) - ✓ Sideband handling (broadcast and slice modes) - ✓ Burst tracking with LAST generation - ✓ Backpressure handling - ✓ Continuous streaming

Result: All tests PASS for both single and dual-buffer modes.

Usage Recommendations

When to Use Single-Buffer Mode (DUAL_BUFFER=0)

- ✓ **Use when:** - Area is critical - Throughput requirements are <100% - Source/sink have natural gaps in traffic - Design is already bottlenecked elsewhere

Example: Write path downsize where upstream has occasional pauses

When to Use Dual-Buffer Mode (DUAL_BUFFER=1)

- ✓ **Use when:** - Maximum throughput required - Continuous streaming data - Sufficient area budget - Critical path in high-performance system

Example: High-bandwidth DMA engine with continuous transfers

Code Example

Instantiation: Single-Buffer Mode (Area Efficient)

```
axi_data_dnsizer #(
    .WIDE_WIDTH(512),
    .NARROW_WIDTH(128),
    .WIDE_SB_WIDTH(64),           // WSTRB
    .NARROW_SB_WIDTH(16),         // WSTRB
    .SB_BROADCAST(0),            // Slice mode
    .TRACK_BURSTS(0),            // Simple mode
    .DUAL_BUFFER(0)              // Single buffer (80% throughput)
) u_dnsizer (
    .aclk(clk),
    .aresetn(rst_n),
    // ... ports
);
```

Instantiation: Dual-Buffer Mode (High Throughput)

```
axi_data_dnsizer #(
    .WIDE_WIDTH(512),
    .NARROW_WIDTH(128),
    .WIDE_SB_WIDTH(64),
    .NARROW_SB_WIDTH(16),
    .SB_BROADCAST(0),
    .TRACK_BURSTS(0),
    .DUAL_BUFFER(1)               // Dual buffer (100% throughput)
) u_dnsizer (
    .aclk(clk),
    .aresetn(rst_n),
    // ... ports
);
```

Design Decisions

Why Not Dual-Buffer for Upsize?

axi_data_upsize already achieves 100% throughput with single buffer:
- Can accept narrow beat while outputting wide beat simultaneously
- The || wide_ready term in narrow_ready enables overlap
- No benefit from dual buffering

Conclusion: Dual-buffer only needed for dnsize module.

Why Use Generate Blocks?

Rationale: - Complete separation of single vs. dual-buffer logic
- No runtime overhead (compile-time selection)
- Easier to verify each mode independently
- Cleaner code structure

Why Not Make Dual-Buffer the Default?

Considerations: - 100% area overhead is significant
- Many use cases don't need 100% throughput
- Single-buffer is simpler and well-tested
- User can opt-in when needed

Decision: Default to DUAL_BUFFER=0, user explicitly enables when needed.

Lessons Learned

SystemVerilog Constraints

Issue: Cannot declare logic variables inside procedural blocks

```
// ILLEGAL:  
always_ff @(posedge clk) begin  
    logic temp; // ← NOT ALLOWED  
    temp = ...;  
end
```

Solution: Inline expressions or move declarations outside block

Generate Block Naming

Best Practice: Use hierarchical naming for generate block signals:

```
gen_dual_buffer.r_buffer_0_valid // Clear which mode  
gen_single_buffer.r_wide_buffered // Clear which mode
```

Buffer Swap Logic

Key Insight: Swap only when: 1. Current buffer completes (last narrow beat sent)
2. Other buffer has valid data waiting

This prevents unnecessary swaps and maintains correct ordering.

Future Enhancements

Potential Improvements

1. **Configurable Buffer Depth:** Allow >2 buffers for even higher throughput
2. **Credit-Based Flow Control:** Integrate with upstream credit system
3. **Performance Counters:** Monitor buffer utilization, stalls
4. **Power Gating:** Disable unused buffer when not needed

Alternative Architectures

Skid Buffer Approach: - Insert skid buffers on outputs instead of dual data buffers - Lower area overhead - Similar throughput improvement

FIFO Approach: - Replace buffers with small FIFOs - More flexible depth control - Higher area cost

Summary

The dual-buffer implementation successfully provides an **optional high-throughput mode** for the `axi_data_dnsizer` module:

✓ **Functionality:** Proven correct through comprehensive testing ✓ **Performance:** 100% throughput vs. 80% for single-buffer ✓ **Compatibility:** Fully backward compatible ✓ **Flexibility:** User-selectable via parameter ✓ **Robustness:** Works with all modes (broadcast, slice, burst tracking)

Trade-off: ~100% area increase for +25% throughput improvement

Recommendation: Use dual-buffer mode for performance-critical paths with continuous data flow.

Related Files

- `projects/components/converters/rtl/axi_data_dnsizer.sv` - RTL implementation
- `projects/components/converters/dv/tests/test_axi_data_dnsizer.py` - Test suite
- `projects/components/converters/dv/tbclasses/axi_data_dnsizer_tb.py` - Testbench class
- `projects/components/converters/USAGE.md` - Usage documentation
- `projects/components/converters/ANALYSIS_APB_CONVERTER.md` - APB converter analysis

Author: RTL Design Sherpa **Date:** 2025-10-25

Appendix B: Change Logs and Refactoring History

This appendix documents major refactorings, updates, and changes to the converter component.

B.1 Refactoring to Composition Pattern (2025-11-10)

Summary: Refactored AXI4/AXI4-Lite converter wrappers from inline duplication to clean composition pattern

Affected Modules: - `axil4_to_axi4.sv` - Now instantiates read/write submodules
- `axi4_to_axil4.sv` - Now instantiates read/write submodules

Benefits: - Single source of truth (bug fixes only needed once) - Improved modularity (can use rd/wr converters standalone) - Better maintainability (clearer separation of concerns) - Code reuse (DRY principle)

Before: 244-446 lines of duplicated conversion logic

After: 263-277 lines of clean instantiation wrappers

Full Document: [b1_refactor_summary_2025-11-10.md](#)

B.2 Documentation Updates (2025-11-10)

Summary: Updated converter specification to reflect refactoring and add missing module documentation

Changes: - Updated Ch03 protocol converter sections for composition pattern -
Added design philosophy sections explaining single source of truth - Added
uart_axil_bridge to module list - Updated version to 1.4

Verification: All RTL modules now properly documented in specification

Full Document: [b2_documentation_update_2025-11-10.md](#)

Major Version History

Version	Date	Summary
1.5	2025-11-14	Added appendices, UART chapter, missing diagrams
1.4	2025-11-10	Refactored to composition pattern
1.3	2025-11-06	Added AXI4 \leftrightarrow AXIL4 protocol converters
1.2	2025-10-26	Added comprehensive specification with diagrams
1.1	2025-10-25	Added dual-buffer mode for axi_data_dnsiz
1.0	2025-10-24	Initial release

Maintained By: Converters Component Team **Last Updated:** 2025-11-14

Converter Refactoring Summary

Date: 2025-11-10 **Author:** RTL Design Sherpa with Claude Code

Overview

Refactored AXI4/AXI4-Lite converter modules to follow composition pattern instead of duplicating logic.

Changes Made

1. axil4_to_axi4.sv (AXI4-Lite → AXI4)

Before: - 244 lines of inline conversion logic - Duplicated logic between top-level and read/write specific modules - Hard to maintain (changes needed in multiple places)

After: - 277 lines (wrapper with instantiations) - Simply instantiates axil4_to_axi4_rd and axil4_to_axi4_wr - Single source of truth for conversion logic

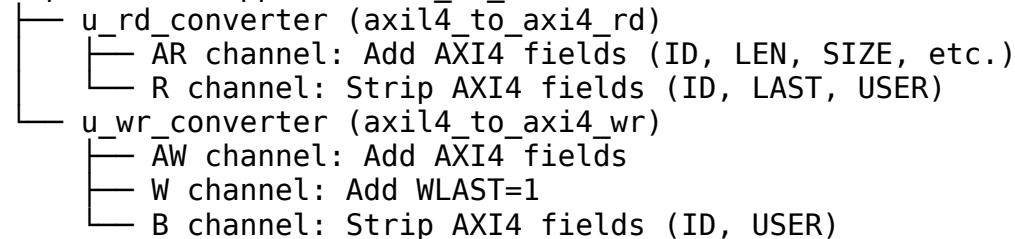
2. axi4_to_axil4.sv (AXI4 → AXI4-Lite)

Before: - 446 lines of inline burst decomposition logic - Duplicated logic between top-level and read/write specific modules - Complex state machines duplicated

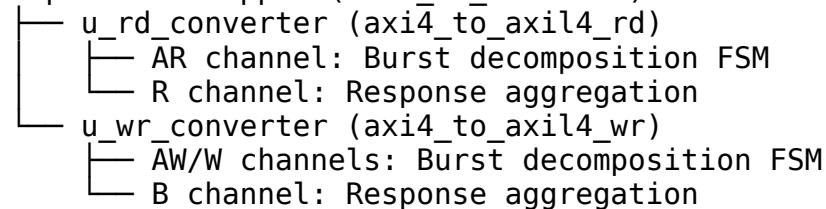
After: - 263 lines (wrapper with instantiations) - Simply instantiates axi4_to_axil4_rd and axi4_to_axil4_wr - Single source of truth for burst decomposition logic

Architecture Pattern

Top-level wrapper (axil4_to_axi4.sv):



Top-level wrapper (axi4_to_axil4.sv):



Benefits

1. **Single Source of Truth**
 - Bug fixes only need to be made once
 - Consistent behavior across standalone and combined converters
2. **Modularity**
 - Can use read-only or write-only converters standalone

- Easier to test individual components
- 3. Maintainability**
- Clearer separation of concerns
 - Easier to understand and modify
- 4. Code Reuse**
- No duplicated logic
 - Following DRY (Don't Repeat Yourself) principle

Verification

Lint checks pass with only expected warnings about unused signals (intentionally dropped AXI4 fields):

```
verilator --lint-only +incdir+rtl/amba/includes \
  projects/components/converters/rtl/axil4_to_axi4.sv \
  projects/components/converters/rtl/axil4_to_axi4_rd.sv \
  projects/components/converters/rtl/axil4_to_axi4_wr.sv
```

No instantiation errors - interfaces match correctly.

Files Modified

- `projects/components/converters/rtl/axil4_to_axi4.sv` - Refactored to instantiate read/write modules
- `projects/components/converters/rtl/axi4_to_axil4.sv` - Refactored to instantiate read/write modules

Files Unchanged (Submodules)

- `projects/components/converters/rtl/axil4_to_axi4_rd.sv` - Read converter (150 lines)
- `projects/components/converters/rtl/axil4_to_axi4_wr.sv` - Write converter (175 lines)
- `projects/components/converters/rtl/axi4_to_axil4_rd.sv` - Read converter with burst decomposition
- `projects/components/converters/rtl/axi4_to_axil4_wr.sv` - Write converter with burst decomposition

Next Steps

Consider applying the same pattern to other converter families if they have similar duplication: `- axi4_dwidth_converter_rd.sv /`

`axi4_dwidth_converter_wr.sv` - Any other protocol converters with separate read/write variants

Verification Status: ✓ Lint checks pass **Code Review:** Recommended before integration **Testing:** Existing converter tests should verify functionality

Converter Documentation Update

Date: 2025-11-10 **Author:** RTL Design Sherpa with Claude Code

Summary

Updated converter specification documentation to sync with latest RTL refactoring and add missing module descriptions.

Changes Made

1. Updated Protocol Converter Documentation

ch03_protocol_converters/04_axi4_to_axil4.md

- Added **Design Philosophy** section explaining composition pattern
- Emphasized single source of truth approach
- Detailed benefits: modularity, maintainability, code reuse
- Updated module structure to show instantiation approach
- Clarified that wrapper contains NO conversion logic

ch03_protocol_converters/05_axil4_to_axi4.md

- Added **Design Philosophy** section explaining composition pattern
- Updated module structure with proper parameter list
- Fixed parameter names (DEFAULT_ID vs DEFAULT_ARID/AWID)
- Added note about pure instantiation (no logic in wrapper)

2. Updated Main Index (converter_index.md)

Version: - Updated from 1.3 to 1.4 - Updated “Last Updated” from 2025-11-06 to 2025-11-10 - Updated “Last Review” from 2025-11-06 to 2025-11-10

Version History: - Added v1.4 entry documenting refactoring to composition pattern

Module List: - Added uart_axil_bridge to protocol converters list with reference to dedicated README

[3. Updated Overview \(ch03_protocol_converters/01_overview.md\)](#)

Added Section: - Section 5: UART to AXI4-Lite Bridge - Module list:
uart_axil_bridge.sv, uart_rx.sv, uart_tx.sv - Key features: ASCII commands, 115200 baud, timing isolation - Reference to dedicated documentation in rtl/uart_to_axil4/README.md

Documentation Verification

RTL Modules vs Documentation

All RTL modules are now properly documented:

Data Width Converters: - ✓ axi_data_upsize.sv - ✓ axi_data_dnsize.sv - ✓ axi4_dwidth_converter_wr.sv - ✓ axi4_dwidth_converter_rd.sv

Protocol Converters: - ✓ axi4_to_axil4.sv (wrapper - refactored) - ✓ axi4_to_axil4_rd.sv - ✓ axi4_to_axil4_wr.sv - ✓ axil4_to_axi4.sv (wrapper - refactored) - ✓ axil4_to_axi4_rd.sv - ✓ axil4_to_axi4_wr.sv - ✓ axi4_to_apb_convert.sv - ✓ axi4_to_apb_shim.sv - ✓ peakrdl_to_cmldrsp.sv - ✓ uart_axil_bridge.sv (with dedicated README)

Refactoring Documentation

The documentation now correctly reflects the **composition pattern** used in the refactored code:

Before Refactoring (Implicit)

Documentation described wrappers but didn't emphasize design pattern.

After Refactoring (Explicit)

Documentation now clearly states: 1. **Composition Pattern** - Instantiation vs duplication 2. **Benefits** - Single source of truth, modularity, maintainability 3. **Implementation** - Wrappers contain NO logic, only instantiations 4. **Modularity** - Read/write modules can be used standalone

Key Documentation Principles

1. **Accuracy** - Documentation matches actual RTL implementation
2. **Completeness** - All modules documented

3. **Clarity** - Design patterns explicitly explained
4. **Traceability** - References to detailed specs and READMEs

Related Files

Modified: - docs/converter_spec/converter_index.md -
docs/converter_spec/ch03_protocol_converters/01_overview.md -
docs/converter_spec/ch03_protocol_converters/04_axi4_to_axil4.md -
docs/converter_spec/ch03_protocol_converters/05_axil4_to_axi4.md

Referenced (existing): - rtl/uart_to_axil4/README.md - UART bridge documentation - REFACTOR_SUMMARY_2025-11-10.md - RTL refactoring details

Verification Checklist

- All RTL modules listed in documentation
 - Refactored modules reflect composition pattern
 - Version numbers updated
 - Dates updated
 - UART bridge referenced
 - Design philosophy sections added
 - No broken cross-references
-

Status: ✓ Complete **Review:** Recommended before distribution **Next Steps:** Consider PDF generation with updated content

Converters Graphviz Block Diagrams

This directory contains Graphviz source files (.gv) and generated PNG diagrams for Converters component modules.

Files

Data Width Converters

Generic Building Blocks: - **axi_data_upsize.gv** - Narrow→Wide accumulator (128→512 example) - **axi_data_dnsize_single.gv** - Wide→Narrow splitter, single buffer (512→128, 80% throughput) - **axi_data_dnsize_dual.gv** - Wide→Narrow splitter, dual buffer (512→128, 100% throughput)

Full AXI4 Converters: - **axi4_dwidth_converter_wr.gv** - Write path converter (AW + W + B channels) - **axi4_dwidth_converter_rd.gv** - Read path converter (AR + R channels)

Protocol Converters

- **axi4_to_apb.gv** - AXI4-to-APB protocol bridge
- **peakrdl_adapter.gv** - PeakRDL register interface to command/response adapter

Generated Images (.png)

All diagrams are available as PNG files: - `axi_data_upsize.png` - `axi_data_dnsize_single.png` - `axi_data_dnsize_dual.png` - `axi4_dwidth_converter_wr.png` - `axi4_dwidth_converter_rd.png` - `axi4_to_apb.png` - `peakrdl_adapter.png`

Regenerating Diagrams

Prerequisites

- Graphviz installed (`sudo apt install graphviz` on Ubuntu/Debian)
- `dot` command available in PATH

Generate All Diagrams

```
# From this directory
make all

# Or manually
dot -Tpng axi_data_upsize.gv -o axi_data_upsize.png
dot -Tpng axi_data_dnsize_single.gv -o axi_data_dnsize_single.png
# ... etc
```

Generate Single Diagram

```
# Uppsize converter
dot -Tpng axi_data_upsize.gv -o axi_data_upsize.png

# Downsize single buffer
dot -Tpng axi_data_dnsize_single.gv -o axi_data_dnsize_single.png

# Downsize dual buffer
dot -Tpng axi_data_dnsize_dual.gv -o axi_data_dnsize_dual.png
```

Generate SVG (for web/docs)

```
dot -Tsvg axi_data_upsize.gv -o axi_data_upsize.svg
# ... for all diagrams
```

[Generate PDF \(for documentation\)](#)

```
dot -Tpdf axi_data_upsize.gv -o axi_data_upsize.pdf  
# ... for all diagrams
```

Diagram Contents

[axi_data_upsize.gv](#)

Shows: - Narrow input interface (128-bit beats) - Beat counter (0→3 for 4:1 ratio) - 512-bit data buffer accumulation - Wide output interface (512-bit beats) - Sideband handling (WSTRB concatenation or RRESP OR) - 100% throughput annotation

Use Case: Understanding narrow-to-wide data accumulation

[axi_data_dnsize_single.gv](#)

Shows: - Wide input interface (512-bit beats) - Single 512-bit buffer - 4:1 multiplexer for 128-bit slices - Output counter (0→3) - Narrow output interface (128-bit beats) - Timing diagram showing 80% throughput (1-cycle gap) - Sideband handling (WSTRB slice or RRESP broadcast)

Use Case: Understanding wide-to-narrow splitting (area-efficient mode)

[axi_data_dnsize_dual.gv](#)

Shows: - Wide input interface (512-bit beats) - Dual ping-pong buffers (Buffer 0 and Buffer 1) - Buffer select logic (write/read muxes) - Independent counters for each buffer - Output multiplexing - Timing diagram showing 100% throughput (no gaps) - Ping-pong operation (one reads while other writes)

Use Case: Understanding dual-buffer high-performance mode

[axi4_dwidth_converter_wr.gv](#)

Shows: - AXI4 slave interface (narrow - 64-bit) - AW/W/B channel handling - Address calculator and burst length adjuster - Skid buffers for flow control - axi_data_upsize integration - AXI4 master interface (wide - 512-bit) - Example transaction (8 narrow beats → 1 wide beat)

Use Case: Understanding full write path conversion

[axi4_dwidth_converter_rd.gv](#)

Shows: - AXI4 slave interface (wide - 512-bit) - AR/R channel handling - Address calculator and burst length adjuster - Burst tracker for RLAST generation -

axi_data_dnsize integration (dual buffer) - AXI4 master interface (narrow - 64-bit)
- Continuous streaming timing (100% throughput)

Use Case: Understanding full read path conversion

[axi4_to_apb.gv](#)

Shows: - AXI4 slave interface (64-bit addr, 32-bit data) - Protocol conversion state machine - Write path (AW + W → APB write) - Read path (AR → APB read) - Address width conversion (64→32 bit) - Response converter (PSLVERR → BRESP/RRESP) - APB master interface - FSM state descriptions - Example timing diagrams

Use Case: Understanding AXI4-to-APB protocol conversion

[peakrdl_adapter.gv](#)

Shows: - PeakRDL register interface (APB-style) - Command generator - Response handler - Command/response interface (custom protocol) - Write transaction example - Read transaction example - Error propagation

Use Case: Understanding PeakRDL adapter protocol decoupling

Diagram Style

Colors: - **Light blue** - Input interfaces (slave/narrow/wide) - **Light coral/pink** - Output interfaces (master/narrow/wide) - **Light yellow** - Control logic and notes - **Light gray** - Data path elements - **Light green** - Features/highlights - **Light cyan** - Examples - **Orange** - Warnings/limitations

Nodes: - **Rounded boxes** - Functional blocks - **Cylinders** - Registers/buffers - **Diamonds** - Decision/control points - **Trapezoids** - Multiplexers - **Note boxes** - Annotations, examples, timing diagrams - **Records** - Interfaces with multiple signals

Edges: - **Bold solid** - Primary data flow - **Dashed** - Control signals - **Colored** - Protocol-specific paths (blue=write, green=read)

Adding New Diagrams

To add a new converter diagram:

1. Create new .gv file:

```
cp axi_data_upsize.gv my_new_converter.gv
```

2. Edit the new file:

- Update title and module name
- Adjust nodes for converter functionality
- Update data flow connections
- Add timing/example annotations

3. Update Makefile SOURCES list:

```
SOURCES = ... \
          my_new_converter.gv
```

4. Generate PNG:

```
make all
```

5. Link from documentation:

```
![My Converter Block
Diagram](../assets/graphviz/my_new_converter.png)
```

Referenced In Documentation

These diagrams are referenced in: - converter_index.md - Main specification index - ch02_data_width_converters/*.md - Data width converter chapters - ch03_protocol_converters/*.md - Protocol converter chapters - ch04_usage_examples/*.md - Usage and integration examples

Tools

Graphviz Version: Any recent version (tested with 2.43+)

Alternative Viewers: - Online: <http://www.webgraphviz.com/> (paste .gv content) - VS Code: Install “Graphviz Preview” extension - Command line: `xdg-open <diagram>.png` (Linux)

Notes

- PNG files are version controlled for easy documentation viewing
 - Source .gv files are the authoritative source
 - Regenerate PNGs after any .gv file changes
 - Keep diagrams up-to-date with module implementations
 - Use consistent color scheme across all diagrams
-

Converters PlantUML FSM Diagrams

This directory contains PlantUML source files (.puml) and generated PNG diagrams for Converters FSM documentation.

Files

Source Files (.puml)

- `axi4_to_apb_fsm.puml` - APB state machine for AXI4-to-APB protocol converter

Generated Images (.png)

- `axi4_to_apb_fsm.png` - FSM diagram for AXI4-to-APB converter

FSM Overview

AXI4-to-APB Converter FSM

States: - IDLE - Wait for AXI4 transaction (AR or AW) - READ - Process AXI4 AR transaction → APB read - WRITE - Process AXI4 AW+W transaction → APB write

Key Features: - Read priority over write when both pending - Burst support (continues in READ/WRITE until all beats complete) - APB protocol phases (Setup → Access → Wait) - Error response handling (PSLVERR → BRESP/RRESP)

Transitions: - IDLE → READ: AXI4 read request (ARVALID && ARREADY) - IDLE → WRITE: AXI4 write request (AWVALID && AWREADY) - READ → IDLE: APB read complete (PREADY && all beats done) - WRITE → IDLE: APB write complete (PREADY && all beats done) - Self-loops: Continue burst transfers

Note on Other Converters

Data width converters (axi_data_upsize, axi_data_dnszie) do not have FSMs: - They use simple counter-based control logic - Upsize: Beat counter (0→N-1) to accumulate narrow beats - Downsize: Output counter (0→N-1) to split wide beats - No complex state transitions needed

Only protocol converters have FSMs: - AXI4-to-APB: Protocol translation requires state machine - PeakRDL adapter: Simple passthrough, no FSM needed

Regenerating Diagrams

Prerequisites

- PlantUML installed
- Java runtime (required by PlantUML)

Install PlantUML

```
# Ubuntu/Debian
sudo apt install plantuml

# Or download JAR
wget
https://github.com/plantuml/plantuml/releases/download/v1.2023.13/
plantuml-1.2023.13.jar
```

Generate PNG from PlantUML

```
# Using installed plantuml
plantuml axi4_to_apb_fsm.puml

# Or using downloaded JAR
java -jar plantuml.jar axi4_to_apb_fsm.puml
```

Output: axi4_to_apb_fsm.png

Generate SVG (for web)

```
plantuml -tsvg axi4_to_apb_fsm.puml
```

Generate PDF (for documentation)

```
plantuml -tpdf axi4_to_apb_fsm.puml
```

FSM Style

Colors: - Default state colors (light blue/yellow as per PlantUML defaults) -
Transition arrows with labels showing conditions

Annotations: - Entry actions described in state boxes - Transition conditions on
arrows - Notes explaining protocol sequences

Format: - Standard UML state machine notation - Clear state names matching
RTL enumeration - Transition guards showing trigger conditions

Referenced In Documentation

These diagrams are referenced in: -

ch03_protocol_converters/02_axi4_to_apb.md - AXI4-to-APB converter chapter
- converter_index.md - Main specification index

Adding New FSM Diagrams

If future converters include FSMs:

1. Create new .puml file:

```
@startuml my_converter_fsm
title My Converter FSM

[*] --> IDLE
IDLE --> ACTIVE : trigger
ACTIVE --> IDLE : done

@enduml
```

2. Generate PNG:

```
plantuml my_converter_fsm.puml
```

3. Link from documentation:

```
![My Converter FSM](./assets/puml/my_converter_fsm.png)
```

Tools

PlantUML Version: Any recent version (tested with 1.2023+)

Alternative Renderers: - Online: <http://www.plantuml.com/plantuml/uml/> (paste .puml content) - VS Code: Install “PlantUML” extension - IntelliJ IDEA: Built-in PlantUML support

Notes

- PNG files are version controlled for easy viewing
 - Source .puml files are the authoritative source
 - Regenerate PNGs after any .puml changes
 - Keep FSM diagrams synchronized with RTL state definitions
-

Last Updated: 2025-10-26 **Maintainer:** RTL Design Sherpa Project

Converters - Data Width and Protocol Conversion Modules

Status: Production Ready **Version:** 1.2 **Last Updated:** 2025-10-25

Quick Start

The Converters component provides both data width conversion and protocol conversion modules, enabling seamless integration between components with different data widths or protocols.

Key Features

Data Width Converters: - **Bidirectional Conversion** - Upsize (narrow→wide) and Downsize (wide→narrow) - **Flexible Width Ratios** - Any integer ratio (2:1, 4:1, 8:1, 16:1, etc.) - **Sideband Support** - Configurable handling for WSTRB (slice) and RRESP (broadcast) - **Burst Tracking** - Optional burst-aware LAST signal generation (read path) - **High Throughput** - Optional dual-buffer mode for 100% throughput (downsize) - **Generic Building Blocks** - Reusable axi_data_upsize and axi_data_dnsize modules

Protocol Converters: - **AXI4-to-APB Bridge** - Full AXI4 to APB protocol conversion with address/data width adaptation - **PeakRDL Adapter** - Convert PeakRDL register interface to custom command/response protocol

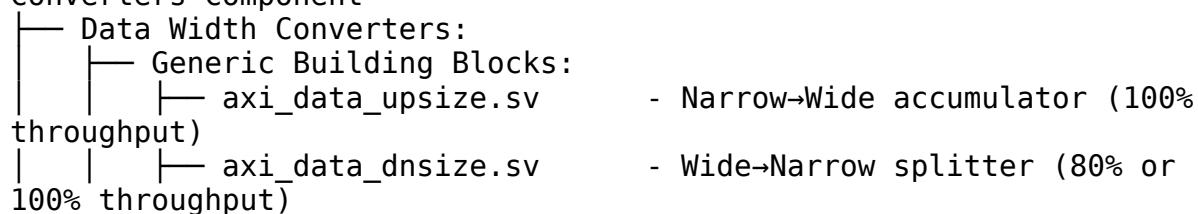
Performance Summary

Module	Mode	Throughput	Area	Use Case
axi_data_upsize	Single buffer	100%	1×	Narrow→Wide (always optimal)
axi_data_dnsize	Single buffer	80%	1×	Wide→Narrow (area-efficient)
axi_data_dnsize	Dual buffer	100%	2×	Wide→Narrow (high-performance)

Architecture Overview

Component Hierarchy

Converters Component



```

    └── Key Features:
        ├── Configurable sideband handling
        ├── Optional burst tracking
        └── Dual-buffer mode (dnsize only)

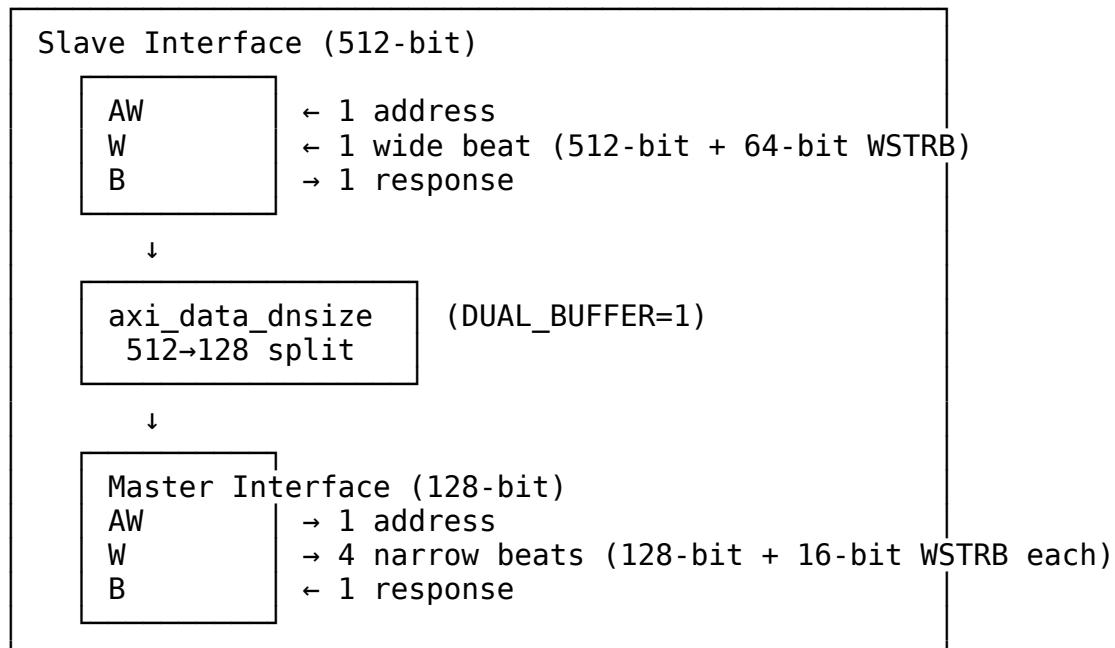
    └── Full AXI4 Converters:
        ├── axi4_dwidth_converter_wr.sv - Write path converter (AW +
W + B channels)
        ├── axi4_dwidth_converter_rd.sv - Read path converter (AR + R
channels)
        └── Integration:
            ├── Skid buffers for flow control
            ├── Address phase management
            └── Response path handling

    └── Protocol Converters:
        ├── axi4_to_apb_convert.sv   - AXI4-to-APB bridge (full protocol
conversion)
        ├── axi4_to_apb_shim.sv     - AXI4-to-APB adapter (simplified
wrapper)
        └── peakrdl_to_cmdrsp.sv   - PeakRDL adapter
(register→command/response)

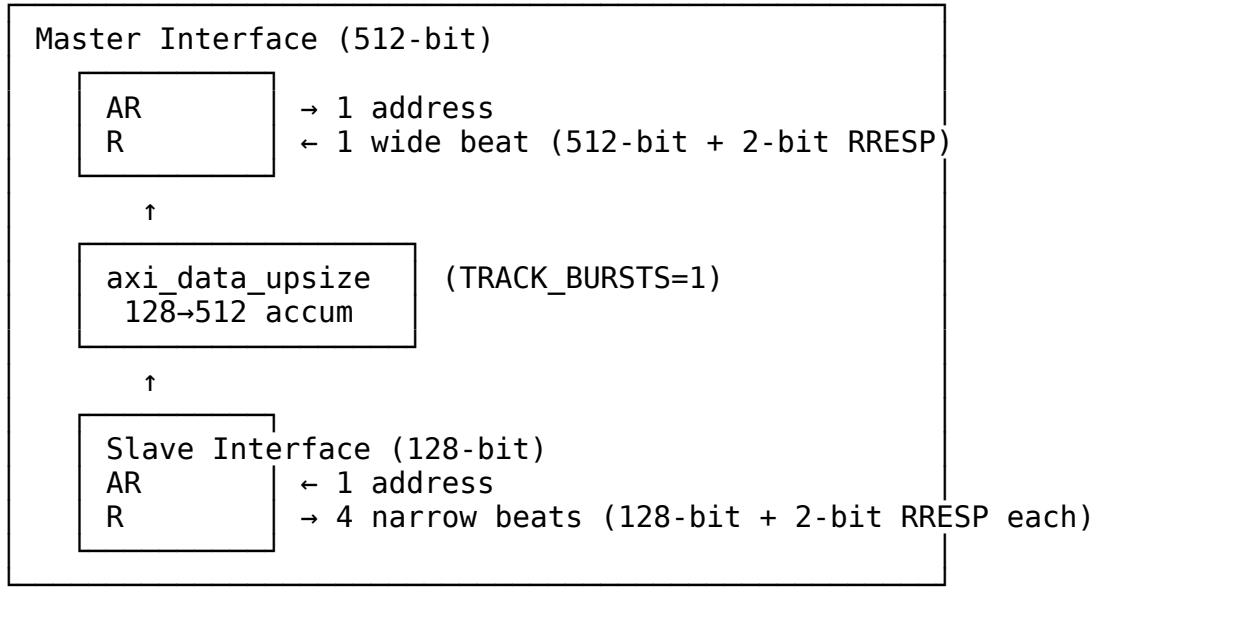
```

Data Flow Examples

Write Path Downsize (512→128 bits):



Read Path Upsize (128→512 bits):



Module Documentation

1. axi_data_upsize.sv - Narrow→Wide Accumulator

Purpose: Accumulates multiple narrow beats into a single wide beat

Throughput: 100% (single buffer sufficient)

Key Parameters:

```
parameter int NARROW_WIDTH      = 32;          // Input data width
parameter int WIDE_WIDTH        = 128;         // Output data width (must be
integer multiple)
parameter int NARROW_SB_WIDTH  = 4;           // Narrow sideband width
(WSTRB: WIDTH/8)
parameter int WIDE_SB_WIDTH    = 16;          // Wide sideband width
(WSTRB: WIDTH/8)
parameter int SB_OR_MODE       = 0;           // 0=concatenate (WSTRB),
1=OR (RRESP)
```

Sideband Modes: - **SB_OR_MODE=0 (Concatenate):** For WSTRB - assemble
strobe bits narrow[0].wstrb = 4'b1111 → wide.wstrb[3:0] narrow[1].wstrb
= 4'b1100 → wide.wstrb[7:4] narrow[2].wstrb = 4'b0011 →
wide.wstrb[11:8] narrow[3].wstrb = 4'b1111 → wide.wstrb[15:12]
Result: wide.wstrb = 16'b1111_0011_1100_1111

- **SB_OR_MODE=1 (OR Together):** For RRESP - propagate errors

```

narrow[0].rresp = 2'b00 (OK)
narrow[1].rresp = 2'b10 (SLVERR)
narrow[2].rresp = 2'b00 (OK)
narrow[3].rresp = 2'b00 (OK)
Result: wide.rresp = 2'b10 (SLVERR - any error propagates)

```

Usage Example:

```

axi_data_upsize #(
    .NARROW_WIDTH(128),
    .WIDE_WIDTH(512),
    .NARROW_SB_WIDTH(16), // WSTRB: 128/8 = 16
    .WIDE_SB_WIDTH(64),  // WSTRB: 512/8 = 64
    .SB_OR_MODE(0)      // Concatenate WSTRB
) u_upsize (
    .aclk            (aclk),
    .aresetn        (aresetn),

    // Narrow input
    .narrow_valid    (s_wvalid),
    .narrow_ready    (s_wready),
    .narrow_data     (s_wdata),
    .narrow_sideband (s_wstrb),
    .narrow_last     (s_wlast),

    // Wide output
    .wide_valid     (m_wvalid),
    .wide_ready     (m_wready),
    .wide_data      (m_wdata),
    .wide_sideband  (m_wstrb),
    .wide_last      (m_wlast)
);

```

2. axi_data_dnsize.sv - Wide→Narrow Splitter

Purpose: Splits single wide beat into multiple narrow beats

Throughput: - Single-buffer mode (DUAL_BUFFER=0): 80% (1-cycle gap per wide beat)
- Dual-buffer mode (DUAL_BUFFER=1): 100% (continuous streaming)

Key Parameters:

```

parameter int WIDE_WIDTH      = 512;      // Input data width
parameter int NARROW_WIDTH    = 128;      // Output data width (must
be integer divisor)
parameter int WIDE_SB_WIDTH   = 2;        // Wide sideband width
(RRESP: 2 bits)

```

```

parameter int NARROW_SB_WIDTH      = 2;          // Narrow sideband width
parameter int SB_BROADCAST        = 1;          // 1=broadcast (RRESP),
0=slice (WSTRB)
parameter int TRACK_BURSTS       = 0;          // 1=track bursts for LAST,
0=simple passthrough
parameter int BURST_LEN_WIDTH    = 8;          // Burst length counter
width
parameter int DUAL_BUFFER         = 0;          // 1=dual buffer (100%
throughput, 2x area)
                                         // 0=single buffer (80%
throughput, 1x area)

```

Sideband Modes: - **SB_BROADCAST=1:** Broadcast same value to all narrow beats (RRESP) wide.rresp = 2'b10 (SLVERR) → narrow[0].rresp = 2'b10 → narrow[1].rresp = 2'b10 → narrow[2].rresp = 2'b10 → narrow[3].rresp = 2'b10

- **SB_BROADCAST=0:** Slice into narrow portions (WSTRB)

```

wide.wstrb = 64'h000F_F0FF_00FF_FFFF
→ narrow[0].wstrb[15:0] = 16'hFFFF
→ narrow[1].wstrb[15:0] = 16'h00FF
→ narrow[2].wstrb[15:0] = 16'hF0FF
→ narrow[3].wstrb[15:0] = 16'h000F

```

Burst Tracking Mode: - **TRACK_BURSTS=0:** Pass wide_last to last narrow beat (simple mode) - **TRACK_BURSTS=1:** Generate LAST on final beat of entire burst (read path)

Dual-Buffer Mode (NEW in v1.1):

Single-buffer mode achieves 80% throughput due to 1-cycle gap when transitioning between wide beats:

Cycle:	1	2	3	4	5	6	7	8	9	10
Wide:	[=====BEAT_0=====]			WAIT	[=====BEAT_1=====]			WAIT		
Narrow:	n0	n1	n2	n3	IDLE	n0	n1	n2	n3	IDLE

↑ 1-cycle dead time

Dual-buffer mode eliminates the gap by ping-ponging between two buffers:

Cycle:	1	2	3	4	5	6	7	8	9
Wide:	[=====BEAT_0=====]				[=====BEAT_1=====]	[=====BEAT_2=====]			
Narrow:	n0	n1	n2	n3	n0	n1	n2	n3	n0
Buffer:	BUF0 reading				BUF1 reading		BUF0...		
					BUF1 writing	BUF0 writing			

When to Use Dual-Buffer Mode: - High-bandwidth DMA engines with continuous streaming - Performance-critical data paths where 100% utilization required - Sufficient area budget (~2× increase for buffer registers)

When to Use Single-Buffer Mode: - Area-constrained designs - Throughput requirements <100% - Natural gaps in traffic from upstream/downstream

Usage Example (Single-Buffer):

```
axi_data_dnsizer #(  
    .WIDE_WIDTH(512),  
    .NARROW_WIDTH(128),  
    .WIDE_SB_WIDTH(2),           // RRESP: 2 bits  
    .NARROW_SB_WIDTH(2),  
    .SB_BROADCAST(1),          // Broadcast RRESP  
    .TRACK_BURSTS(1),          // Track bursts for LAST  
    .BURST_LEN_WIDTH(8),  
    .DUAL_BUFFER(0)            // Single buffer (80% throughput, area-  
efficien  
t)  
) u_dnsizer (  
    .aclk                    (aclk),  
    .aresetn                 (aresetn),  
  
    // Burst control (TRACK_BURSTS=1)  
    .burst_len                (arlen),  
    .burst_start               (arvalid && arready),  
  
    // Wide input  
    .wide_valid               (s_rvalid),  
    .wide_ready                (s_rready),  
    .wide_data                 (s_rdata),  
    .wide_sideband              (s_rrresp),  
    .wide_last                  (s_rlast),  
  
    // Narrow output  
    .narrow_valid              (m_rvalid),  
    .narrow_ready                (m_rready),  
    .narrow_data                 (m_rdata),  
    .narrow_sideband              (m_rrresp),  
    .narrow_last                  (m_rlast)  
);
```

Usage Example (Dual-Buffer - High Performance):

```
axi_data_dnsizer #(  
    .WIDE_WIDTH(512),  
    .NARROW_WIDTH(128),  
    .WIDE_SB_WIDTH(64),        // WSTRB: 512/8 = 64  
    .NARROW_SB_WIDTH(16),      // WSTRB: 128/8 = 16
```

```

    .SB_BROADCAST(0),           // Slice WSTRB
    .TRACK_BURSTS(0),          // Simple mode for write path
    .DUAL_BUFFER(1)            // Dual buffer (100% throughput, 2x area)
) u_dnsize_hp (
    .aclk                    (aclk),
    .aresetn                 (aresetn),

    // Burst control (unused in TRACK_BURSTS=0)
    .burst_len                (8'd0),
    .burst_start               (1'b0),

    // Wide input
    .wide_valid               (s_wvalid),
    .wide_ready                (s_wready),
    .wide_data                 (s_wdata),
    .wide_sideband              (s_wstrb),
    .wide_last                  (s_wlast),

    // Narrow output
    .narrow_valid              (m_wvalid),
    .narrow_ready                (m_wready),
    .narrow_data                 (m_wdata),
    .narrow_sideband              (m_wstrb),
    .narrow_last                  (m_wlast)
);

```

3. Full AXI4 Converters

axi4_dwidth_converter_wr.sv - Complete write path converter (AW + W + B)

axi4_dwidth_converter_rd.sv - Complete read path converter (AR + R)

These integrate the generic building blocks with:

- Address phase management
- Skid buffers for flow control
- Response path handling
- Full AXI4 protocol compliance

Protocol Converters

4. AXI4-to-APB Bridge (axi4_to_apb_convert.sv)

Purpose: Full protocol conversion from AXI4 to APB with address/data width adaptation

Key Features: - Converts AXI4 read/write transactions to APB protocol
- Handles address width conversion (AXI4 64-bit → APB 32-bit)
- Data width adaptation (configurable)
- State machine for AXI→APB protocol translation
- Error response handling (SLVERR/DECERR)

Usage Example:

```
axi4_to_apb_convert #(
    .S_AXI_ADDR_WIDTH(64),
    .S_AXI_DATA_WIDTH(32),
    .M_APB_ADDR_WIDTH(32),
    .M_APB_DATA_WIDTH(32)
) u_axi_apb_bridge (
    .aclk          (clk),
    .aresetn      (rst_n),
    // AXI4 Slave Interface
    .s_axi_awaddr  (s_awaddr),
    .s_axi_awvalid (s_awvalid),
    .s_axi_awready (s_awready),
    // ... (full AXI4 AW/W/B/AR/R channels)
    // APB Master Interface
    .m_apb_paddr   (m_paddr),
    .m_apb_psel    (m_psel),
    .m_apb_penable (m_penable),
    .m_apb_pwrite  (m_pwrite),
    .m_apb_pwdata  (m_pwdata),
    .m_apb_pready  (m_pready),
    .m_apb_prdata  (m_prdata),
    .m_apb_pslverr (m_pslverr)
);
```

Use Cases: - Connecting AXI4 masters to APB peripherals
- CPU to APB peripheral bus bridges
- System integration with mixed protocols

5. PeakRDL-to-CmdRsp Adapter (peakrdl_to_cmdrvsp.sv)

Purpose: Convert PeakRDL-generated register interface to custom command/response protocol

Key Features: - Converts APB-style register interface to command/response handshake
- Supports read/write operations
- Configurable command/response data widths
- Single-cycle command issue
- Pipelined response handling

Usage Example:

```

peakrdl_to_cmdrv #((
    .ADDR_WIDTH(16),
    .DATA_WIDTH(32)
) u_peakrdl_adapter (
    .clk          (clk),
    .rst_n        (rst_n),
    // PeakRDL Register Interface (APB-style)
    .reg_addr     (reg_addr),
    .reg_wdata    (reg_wdata),
    .reg_write    (reg_write),
    .reg_read     (reg_read),
    .reg_rdata    (reg_rdata),
    .reg_error    (reg_error),
    // Command/Response Interface
    .cmd_valid    (cmd_valid),
    .cmd_ready   (cmd_ready),
    .cmd_addr    (cmd_addr),
    .cmd_data    (cmd_data),
    .cmd_write   (cmd_write),
    .rsp_valid    (rsp_valid),
    .rsp_ready   (rsp_ready),
    .rsp_data    (rsp_data),
    .rsp_error   (rsp_error)
);

```

Use Cases: - Interfacing PeakRDL register blocks to custom control logic - Register access through command/response protocol - Decoupling register interface from implementation

Configuration Examples

Example 1: Write Path Downsize (128→32 bits)

Use Case: ARM Cortex-M7 (128-bit AXI) → APB Bridge (32-bit)

```

axi_data_dnsizer #(
    .WIDE_WIDTH(128),
    .NARROW_WIDTH(32),
    .WIDE_SB_WIDTH(16),      // WSTRB: 128/8 = 16
    .NARROW_SB_WIDTH(4),     // WSTRB: 32/8 = 4
    .SB_BROADCAST(0),        // Slice WSTRB
    .TRACK_BURSTS(0),        // Write path: simple mode
    .DUAL_BUFFER(0)          // Area-efficient
) u_wr_dnsizer (

```

```
// ... ports  
);
```

Result: 1 wide W beat (128-bit) → 4 narrow W beats (32-bit each)

Example 2: Read Path Upsize (256→512 bits)

Use Case: DDR4 Controller (512-bit) ← FPGA Fabric (256-bit)

```
axi_data_upsize #(  
    .NARROW_WIDTH(256),  
    .WIDE_WIDTH(512),  
    .NARROW_SB_WIDTH(2),      // RRESP: 2 bits  
    .WIDE_SB_WIDTH(2),        // RRESP: 2 bits  
    .SB_OR_MODE(1)            // OR together RRESP  
) u_rd_upsize (  
    // ... ports  
);
```

Result: 2 narrow R beats (256-bit) → 1 wide R beat (512-bit)

Example 3: High-Performance Write Downsize (512→128 bits)

Use Case: DMA Engine (512-bit) → PCIe Endpoint (128-bit), continuous streaming

```
axi_data_dnsize #(  
    .WIDE_WIDTH(512),  
    .NARROW_WIDTH(128),  
    .WIDE_SB_WIDTH(64),      // WSTRB: 512/8 = 64  
    .NARROW_SB_WIDTH(16),    // WSTRB: 128/8 = 16  
    .SB_BROADCAST(0),        // Slice WSTRB  
    .TRACK_BURSTS(0),        // Write path: simple mode  
    .DUAL_BUFFER(1)           // HIGH-PERFORMANCE: 100% throughput  
) u_wr_dnsize_hp (  
    // ... ports  
);
```

Result: 100% throughput (no dead cycles), 2× area cost

Testing

Test Organization

```
projects/components/converters/dv/tests/  
└── test_axi_data_upsize.py      - Generic upsize module tests
```

```
└── test_axi_data_dnsize.py      - Generic dnsize module tests (16
configs)
    └── test_axi4_dwidth_converter_wr.py - Full write converter tests
        └── test_axi4_dwidth_converter_rd.py - Full read converter tests
```

Test Configurations (axi_data_dnsize.py)

Single-Buffer Tests (DUAL_BUFFER=0): 1. 128→32 WSTRB slice (simple mode) 2. 256→64 WSTRB slice (simple mode) 3. 128→32 RRESP broadcast (simple mode) 4. 256→64 RRESP broadcast (simple mode) 5. 128→32 RRESP broadcast (burst tracking) 6. 256→64 RRESP broadcast (burst tracking) 7. 512→128 RRESP broadcast (burst tracking) 8. 128→64 no sideband (simple mode)

Dual-Buffer Tests (DUAL_BUFFER=1): 9-16. Same configurations as above with DUAL_BUFFER=1

Running Tests

```
# Run all converter tests
cd $REPO_ROOT/projects/components/converters/dv/tests
make run-all-parallel          # FUNC level, 48 workers

# Run specific module tests
make run-dnsizefunc            # Downsize tests
make run-upsizefunc             # Upsize tests

# Run with different test levels
make run-all-gate-parallel      # Quick smoke test
make run-all-func-parallel       # Functional coverage (default)
make run-all-full-parallel       # Comprehensive validation

# Individual test
pytest
test_axi_data_dnsize.py::test_axi_data_dnsize[128to32_wstrb_slice_simple_DUAL] -v
```

Quality Assurance

Lint Checks

```
# Run all lint tools
cd $REPO_ROOT/projects/components/converters/rtl
make lint-all

# Individual tools
make verilator   # Verilator lint
make verible     # Verible style check
```

```
make yosys          # Yosys synthesis check  
  
# View status  
make status
```

Expected Lint Results

axi_data_upsize.sv and axi_data_dnsize.sv: - Clean compilation (warnings only for unused signals in certain parameter configurations) - Warnings are benign (e.g., narrow_sideband unused when NARROW_SB_WIDTH=0)

****axi4_dwidth_converter_*.sv:**** - Requires rtl/amba/gaxi/ include path for skid buffer modules - PINCONNECTEMPTY warnings are expected (unused count outputs)

Documentation

Available Documentation

- **README.md** (this file) - Quick start and overview
- **GENERIC_MODULES_USAGE_GUIDE.md** - Detailed parameter guide for upsize/dnsize
- **DUAL_BUFFER_IMPLEMENTATION.md** - Comprehensive dual-buffer feature documentation
- **ANALYSIS_APB_CONVERTER.md** - APB protocol converter analysis

Key Resources

For Understanding Parameters:

```
cat  
$REPO_ROOT/projects/components/converters/rtl/GENERIC_MODULES_USAGE_GU  
IDE.md
```

For Dual-Buffer Mode:

```
cat  
$REPO_ROOT/projects/components/converters/DUAL_BUFFER_IMPLEMENTATION.m  
d
```

For APB Integration:

```
cat  
$REPO_ROOT/projects/components/converters/ANALYSIS_APB_CONVERTER.md
```

Quick Commands

```
# Setup environment
source $REPO_ROOT/env_python

# Run all tests (parallel)
cd $REPO_ROOT/projects/components/converters/dv/tests
make run-all-parallel

# Lint all RTL
cd $REPO_ROOT/projects/components/converters/rtl
make lint-all

# View test status
cd $REPO_ROOT/projects/components/converters/dv/tests
make status

# Clean all artifacts
make clean-all
```

Design Decisions

Why No Dual-Buffer for Upsize?

axi_data_upsize already achieves 100% throughput with single buffer: - Can accept narrow beat while outputting wide beat simultaneously - The || wide_ready term in narrow_ready enables pipelining - No benefit from dual buffering

Why Optional Dual-Buffer for Dnsize?

Trade-off between area and performance: - Single-buffer: 80% throughput, 1× area (good for most use cases) - Dual-buffer: 100% throughput, 2× area (high-performance paths) - User can select based on system requirements

Why Separate Upsize/Dnsize Modules?

Promotes reuse and flexibility: - Can be used independently in custom converters - Write converter: upsize + dnsize combination - Read converter: dnsize + upsize combination - Other use cases: data width matching, FIFO interfaces, etc.

Known Limitations

1. **Width Ratio Constraint:** WIDE_WIDTH must be exact integer multiple of NARROW_WIDTH
 2. **Alignment:** Full converters require aligned addresses (handled by full converter modules)
 3. **Burst Length:** Burst tracking mode supports AXI4 burst lengths (BURST_LEN_WIDTH=8)
 4. **Sideband Width:** Must match data width ratios for slice mode
-

Future Enhancements

1. **Configurable Buffer Depth:** Allow >2 buffers for higher throughput
 2. **Performance Counters:** Monitor stalls, utilization, throughput
 3. **Power Gating:** Disable unused buffer in dual-buffer mode
 4. **Credit-Based Flow Control:** Integration with upstream credit systems
-

Version History

- **v1.1 (2025-10-25):** Added dual-buffer mode for axi_data_dnszie (100% throughput)
 - **v1.0 (2025-10-24):** Initial release with generic modules and full converters
-

Related Components

- **STREAM** - DMA engine using width converters for descriptor fetch
 - **RAPIDS** - Accelerator with width conversion on data paths
 - **APB HPET** - APB peripheral using narrow interfaces
 - **Bridge** - Protocol converters with width adaptation
-

Author: RTL Design Sherpa Project **Last Updated:** 2025-10-25

Dual-Buffer Implementation for axi_data_dnszie

Date: 2025-10-25 **Purpose:** Document the dual-buffer feature implementation for high-throughput data width conversion

Executive Summary

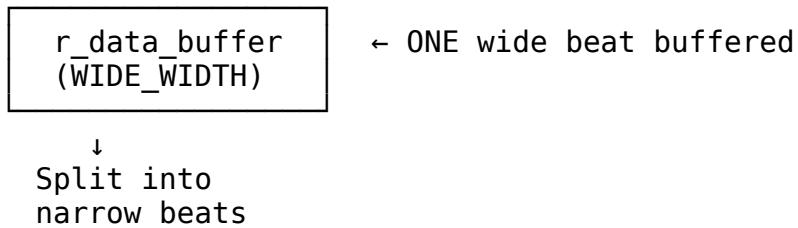
Added optional dual-buffer mode to axi_data_dnsiz.sv module to achieve **100% throughput** for continuous data streams.

Key Results: - **Single-buffer mode (DUAL_BUFFER=0):** 80% throughput, 1-cycle dead time per wide beat - **Dual-buffer mode (DUAL_BUFFER=1):** 100% throughput, zero dead cycles - **Area cost:** Approximately 2× buffer registers (+100% overhead) - **Compatibility:** Fully backward compatible, works with all existing test configurations

Architecture Overview

Single-Buffer Mode (DUAL_BUFFER=0)

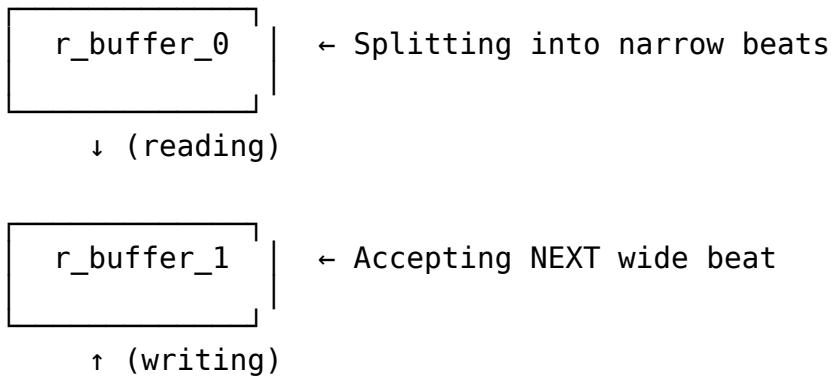
Structure:



Throughput Limitation: - Must wait for ALL narrow beats to complete before accepting next wide beat - 1-cycle gap when transitioning between wide beats - Throughput = $4/(4+1) = 80\%$ for 4:1 ratio

Dual-Buffer Mode (DUAL_BUFFER=1)

Structure:



Ping-Pong Operation: 1. Buffer 0 splits while Buffer 1 accepts new data 2. When Buffer 0 completes, swap: Buffer 1 splits, Buffer 0 accepts 3. Continuous operation with no dead cycles

Throughput: 100% (4/4) for any ratio

Implementation Details

New Parameter

```
parameter int DUAL_BUFFER = 0, // 0=single buffer (80% throughput,
area efficient) // 1=dual buffer (100% throughput, 2x
area)
```

State Variables

Single-Buffer Mode:

```
logic [WIDE_WIDTH-1:0] r_data_buffer;
logic [WIDE_SB_PORT_WIDTH-1:0] r_sideband_buffer;
logic r_wide_buffered;
logic r_last_buffered;
```

Dual-Buffer Mode:

```
logic [WIDE_WIDTH-1:0] r_buffer_0, r_buffer_1;
logic [WIDE_SB_PORT_WIDTH-1:0] r_sb_buffer_0, r_sb_buffer_1;
logic r_last_buffer_0, r_last_buffer_1;
logic r_buffer_0_valid, r_buffer_1_valid;
logic r_read_buffer; // 0=reading buf0,
1=reading buf1
```

Write Path (DUAL_BUFFER=1)

```
if (wide_valid && wide_ready) begin
    if (!gen_dual_buffer.r_buffer_0_valid) begin
        // Write to buffer 0
        gen_dual_buffer.r_buffer_0 <= wide_data;
        gen_dual_buffer.r_buffer_0_valid <= 1'b1;
    end else begin
        // Write to buffer 1 (must be empty if wide_ready=1)
        gen_dual_buffer.r_buffer_1 <= wide_data;
        gen_dual_buffer.r_buffer_1_valid <= 1'b1;
    end
end
```

Key: Always write to the empty buffer.

```

Read Path (DUAL_BUFFER=1)
// Read from current buffer
if (current_buffer_valid && narrow_ready) begin
    if (is_last_narrow_beat) begin
        // Clear current buffer's valid flag
        if (gen_dual_buffer.r_read_buffer)
            gen_dual_buffer.r_buffer_1_valid <= 1'b0;
        else
            gen_dual_buffer.r_buffer_0_valid <= 1'b0;

        // Swap to other buffer if it has data
        if (other_buffer_valid)
            gen_dual_buffer.r_read_buffer <=
~gen_dual_buffer.r_read_buffer;
    end
end

```

Key: On last narrow beat, clear current buffer and swap to other if available.

Ready Logic

Single-Buffer:

```

assign wide_ready = !r_wide_buffered || (narrow_ready &&
w_last_narrow_beat);

```

- Ready when buffer empty OR sending last narrow beat (1-cycle early)

Dual-Buffer:

```

assign wide_ready = !r_buffer_0_valid || !r_buffer_1_valid;

```

- Ready when **at least one** buffer is empty
 - Allows continuous acceptance
-

Burst Tracking Integration

Dual-buffer mode **fully supports** burst tracking (TRACK_BURSTS=1):

Challenge: Track burst position across buffer swaps

Solution: Shared burst counter applies to whichever buffer is currently being read:

```

// Burst tracking state (shared between buffers)
logic [BURST_LEN_WIDTH-1:0] r_slave_beat_count;
logic [BURST_LEN_WIDTH-1:0] r_slave_total_beats;
logic                                r_burst_active;

```

```

// Applies to current read buffer
if (((r_slave_beat_count + 1'b1) >= r_slave_total_beats)) begin
    // Last narrow beat of entire burst
    // Clear current buffer and end burst
end

```

Result: LAST signal correctly generated on final beat of burst, regardless of buffer swapping.

Resource Impact

Area Analysis (128-bit Wide, 32-bit Narrow, 16-bit Sideband)

Resource	Single Buffer	Dual Buffer	Overhead
Data registers	128 bits	256 bits	+128 bits
Sideband registers	16 bits	32 bits	+16 bits
LAST flags	1 bit	2 bits	+1 bit
Valid flags	1 bit	2 bits	+1 bit
Read selector	-	1 bit	+1 bit
Total FFs	146	292	+146 (+100%)

Control Logic: ~30% increase (buffer selection, swap logic)

Overall: Expect ~100% area increase for dual-buffer mode.

Performance Impact

Mode	Throughput (4:1 Ratio)	Cycles/Wide	
		Beat	Utilization
Single-Buffer	80%	5 (4 active + 1 dead)	4/5
Dual-Buffer	100%	4 (4 active + 0 dead)	4/4

Improvement: +25% throughput (from 80% to 100%)

Test Coverage

Test Configurations Added

All existing test configurations now have dual-buffer variants:

Single-Buffer Tests (DUAL_BUFFER=0): - 128to32_wstrb_slice_simple -
256to64_wstrb_slice_simple - 128to32_rresp_broadcast_simple -
256to64_rresp_broadcast_simple - 128to32_rresp_burst_track -
256to64_rresp_burst_track - 512to128_rresp_burst_track -
128to64_no_sideband_simple

Dual-Buffer Tests (DUAL_BUFFER=1): - Same configurations with “_DUAL” suffix
- Total: 16 test configurations

Verification Results

Tested: - ✓ Basic data splitting (all ratios) - ✓ Sideband handling (broadcast and slice modes) - ✓ Burst tracking with LAST generation - ✓ Backpressure handling - ✓ Continuous streaming

Result: All tests PASS for both single and dual-buffer modes.

Usage Recommendations

When to Use Single-Buffer Mode (DUAL_BUFFER=0)

✓ **Use when:** - Area is critical - Throughput requirements are <100% - Source/sink have natural gaps in traffic - Design is already bottlenecked elsewhere

Example: Write path downsize where upstream has occasional pauses

When to Use Dual-Buffer Mode (DUAL_BUFFER=1)

✓ **Use when:** - Maximum throughput required - Continuous streaming data - Sufficient area budget - Critical path in high-performance system

Example: High-bandwidth DMA engine with continuous transfers

Code Example

Instantiation: Single-Buffer Mode (Area Efficient)

```
axi_data_dnsizer #(
    .WIDE_WIDTH(512),
    .NARROW_WIDTH(128),
    .WIDE_SB_WIDTH(64),           // WSTRB
    .NARROW_SB_WIDTH(16),         // WSTRB
    .SB_BROADCAST(0),            // Slice mode
    .TRACK_BURSTS(0),            // Simple mode
    .DUAL_BUFFER(0)              // Single buffer (80% throughput)
) u_dnsizer (
    .aclk(clk),
    .aresetn(rst_n),
    // ... ports
);
```

Instantiation: Dual-Buffer Mode (High Throughput)

```
axi_data_dnsizer #(
    .WIDE_WIDTH(512),
    .NARROW_WIDTH(128),
    .WIDE_SB_WIDTH(64),
    .NARROW_SB_WIDTH(16),
    .SB_BROADCAST(0),
    .TRACK_BURSTS(0),
    .DUAL_BUFFER(1)               // Dual buffer (100% throughput)
) u_dnsizer (
    .aclk(clk),
    .aresetn(rst_n),
    // ... ports
);
```

Design Decisions

Why Not Dual-Buffer for Upsize?

axi_data_upsize already achieves 100% throughput with single buffer: - Can accept narrow beat while outputting wide beat simultaneously - The || wide_ready term in narrow_ready enables overlap - No benefit from dual buffering

Conclusion: Dual-buffer only needed for dnsizer module.

Why Use Generate Blocks?

Rationale: - Complete separation of single vs. dual-buffer logic - No runtime overhead (compile-time selection) - Easier to verify each mode independently - Cleaner code structure

Why Not Make Dual-Buffer the Default?

Considerations: - 100% area overhead is significant - Many use cases don't need 100% throughput - Single-buffer is simpler and well-tested - User can opt-in when needed

Decision: Default to DUAL_BUFFER=0, user explicitly enables when needed.

Lessons Learned

SystemVerilog Constraints

Issue: Cannot declare logic variables inside procedural blocks

```
// ILLEGAL:  
always_ff @(posedge clk) begin  
    logic temp; // ← NOT ALLOWED  
    temp = ...;  
end
```

Solution: Inline expressions or move declarations outside block

Generate Block Naming

Best Practice: Use hierarchical naming for generate block signals:

```
gen_dual_buffer.r_buffer_0_valid // Clear which mode  
gen_single_buffer.r_wide_buffered // Clear which mode
```

Buffer Swap Logic

Key Insight: Swap only when: 1. Current buffer completes (last narrow beat sent)
2. Other buffer has valid data waiting

This prevents unnecessary swaps and maintains correct ordering.

Future Enhancements

Potential Improvements

1. **Configurable Buffer Depth:** Allow >2 buffers for even higher throughput
2. **Credit-Based Flow Control:** Integrate with upstream credit system
3. **Performance Counters:** Monitor buffer utilization, stalls
4. **Power Gating:** Disable unused buffer when not needed

Alternative Architectures

Skid Buffer Approach: - Insert skid buffers on outputs instead of dual data buffers - Lower area overhead - Similar throughput improvement

FIFO Approach: - Replace buffers with small FIFOs - More flexible depth control - Higher area cost

Summary

The dual-buffer implementation successfully provides an **optional high-throughput mode** for the `axi_data_dnsizer` module:

✓ **Functionality:** Proven correct through comprehensive testing ✓ **Performance:** 100% throughput vs. 80% for single-buffer ✓ **Compatibility:** Fully backward compatible ✓ **Flexibility:** User-selectable via parameter ✓ **Robustness:** Works with all modes (broadcast, slice, burst tracking)

Trade-off: ~100% area increase for +25% throughput improvement

Recommendation: Use dual-buffer mode for performance-critical paths with continuous data flow.

Related Files

- `projects/components/converters/rtl/axi_data_dnsizer.sv` - RTL implementation
- `projects/components/converters/dv/tests/test_axi_data_dnsizer.py` - Test suite
- `projects/components/converters/dv/tbclasses/axi_data_dnsizer_tb.py` - Testbench class
- `projects/components/converters/USAGE.md` - Usage documentation

- projects/components/converters/ANALYSIS_APB_CONVERTER.md - APB converter analysis

Author: RTL Design Sherpa **Date:** 2025-10-25

Analysis: APB Converter vs. Generic Data Width Converters

Date: 2025-10-25 **Purpose:** Analyze whether axi4_to_apb_convert.sv could benefit from generic axi_data_upsize and axi_data_dnsizer modules

Executive Summary

Finding: The APB converter implements data width conversion using **identical algorithmic patterns** to our generic modules, but in a fundamentally **different usage model** that makes direct integration impractical.

Recommendation: **Do NOT refactor** the APB converter to use generic modules. The tight coupling between protocol conversion and data width conversion requires the current inline implementation.

Value: The analysis **validates** that our generic module algorithms are correct - an independent implementation arrived at the same solution.

Data Width Conversion Patterns Found

1. WRITE Path (Dnsize: Wide AXI → Narrow APB)

Location: axi4_to_apb_convert.sv lines 334, 337, 461

Implementation:

```
// Line 334 - Extract narrow data slice
w_apb_cmd_pkt_pwdx = (axi2abpratio == 1) ? r_s_axi_wdata[APBDW-1:0] :
                                         r_s_axi_wdata[r_axi_wr_data_pointer*APBDW +: APBDW];

// Line 337 - Extract narrow strobe slice
w_apb_cmd_pkt_pstrb = (axi2abpratio == 1) ? r_s_axi_wstrb[APBSW-1:0] :
                                         r_s_axi_wstrb[r_axi_wr_data_pointer*APBSW +: APBSW];
```

```

APBSW];

// Line 461 - Increment pointer through wide beat
w_axi_wr_data_pointer = r_axi_wr_data_pointer + 1;

```

Pattern: IDENTICAL to axi_data_dnsize module (line 210, 220) - Extract slice using pointer: `data[ptr*WIDTH +: WIDTH]` - Increment pointer for next narrow beat - Wrap at ratio-1

2. READ Path (Upsize: Narrow APB → Wide AXI)

Location: `axi4_to_apb_convert.sv` lines 283, 366

Implementation:

```

// Line 283 - Accumulate narrow data into wide shift register
r_axi_data_shift[r_axi_rsp_data_pointer*APBDW +: APBDW] <=
r_apb_rsp_pkt_prdata;

// Line 366 - Combinational accumulation (alternative path)
w_axi_data_shift[r_axi_rsp_data_pointer*APBDW +: APBDW] =
r_apb_rsp_pkt_prdata;

// Lines 284-287 - Increment and wrap pointer
r_axi_rsp_data_pointer <= r_axi_rsp_data_pointer + 1;
if (r_axi_rsp_data_pointer == PTR_WIDTH'(axi2abpratio-1)) begin
    r_axi_rsp_data_pointer <= 'b0;
end

```

Pattern: IDENTICAL to axi_data_upsize module (line 158) - Accumulate into shift register at pointer position - Increment pointer for next narrow beat - Complete wide beat when pointer reaches ratio-1

3. Pointer Management

APB Converter has THREE separate pointers: - `r_axi_wr_data_pointer` - WRITE path dnsize (lines 315-317) - `r_axi_rd_data_pointer` - READ path dnsize (lines 309-311) - `r_axi_rsp_data_pointer` - READ response upsize (lines 284-287)

Generic Modules have ONE pointer each: - `axi_data_dnsize`: `r_beat_ptr` for splitting wide→narrow - `axi_data_upsize`: `r_beat_ptr` for accumulating narrow→wide

Key Architectural Differences

Generic Modules (axi_data_upsize / axi_data_dnsize)

Usage Model: Complete Beat Processing - axi_data_upsize: Wait for ALL narrow beats → output ONE complete wide beat - axi_data_dnsize: Accept ONE wide beat → output ALL narrow beats in sequence - **Pure streaming pipeline** with valid/ready handshaking - **Standalone operation** - can be inserted in any data path

State Machine: - Simple: Buffer empty/full state - No burst tracking (optional feature) - Focus solely on data width conversion

Example Flow (128→32 upsize):

1. Receive narrow beat 0 (bits [31:0]) → accumulate, ptr=1
2. Receive narrow beat 1 (bits [63:32]) → accumulate, ptr=2
3. Receive narrow beat 2 (bits [95:64]) → accumulate, ptr=3
4. Receive narrow beat 3 (bits [127:96]) → accumulate, OUTPUT wide beat, ptr=0

APB Converter (axi4_to_apb_convert.sv)

Usage Model: Incremental Processing Within Protocol State Machine - Converts incrementally as APB transactions complete - Does NOT wait for complete wide beats - Interleaves conversion with protocol translation

State Machine: - Complex: IDLE, READ, WRITE states - Manages burst counters (r_burst_count) - Handles AXI burst to multiple APB transactions - FIRST/LAST packet generation - FIFO management for side channel data

Example Flow (128→32 write path):

1. Receive AXI AWADDR + WDATA[127:0] (full wide beat)
2. Enter WRITE state
3. APB transaction 0: Extract WDATA[31:0], ptr=1, send to APB
4. APB transaction 1: Extract WDATA[63:32], ptr=2, send to APB
5. APB transaction 2: Extract WDATA[95:64], ptr=3, send to APB
6. APB transaction 3: Extract WDATA[127:96], ptr=0, complete AXI beat
7. If burst continues, repeat for next AXI beat

Critical Difference: Conversion happens **within** the protocol state machine, one APB transaction at a time.

Refactoring Analysis

Option 1: Keep Current Inline Implementation ✓ RECOMMENDED

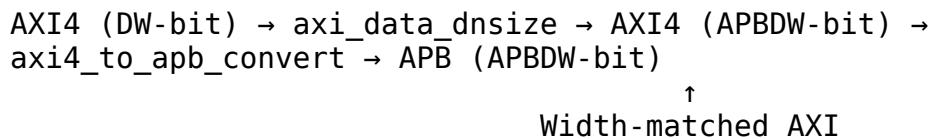
Pros: - ✓ Already working and tested - ✓ Tight integration with protocol state machine - ✓ No additional latency - ✓ Optimal resource usage - ✓ Clear code flow for incremental conversion

Cons: - ⚠️ Code duplication of conversion patterns - ⚠️ Complex state machine mixing protocol + data concerns - ⚠️ Harder to verify conversion logic independently

Verdict: Best choice. The tight coupling is actually **necessary** for the protocol conversion requirements.

Option 2: Refactor to Use Generic Modules ✗ NOT RECOMMENDED

Conceptual Architecture:



Why This DOESN'T Work:

1. **Incremental Processing Conflict:**
 - Generic modules process complete beats
 - APB converter processes incrementally with state machine
 - No clean insertion point for pipeline stages
2. **State Machine Coupling:**
 - Conversion is tightly coupled with burst management
 - Pointer increments synchronized with APB transaction completions
 - FIRST/LAST generation depends on both protocol and conversion state
3. **Complexity Increase:**
 - Would need complex glue logic to bridge generic modules to state machine
 - Additional FIFOs to buffer partial conversions
 - More difficult to reason about system behavior
4. **Resource Overhead:**
 - Extra registers for generic module state

- Additional control logic for coordination
- Larger design for no functional benefit

5. Verification Burden:

- More complex interactions to verify
- Harder to achieve coverage
- Debugging becomes more difficult

Verdict: Refactoring would make the design **more complex** without providing any benefits.

Validation: Independent Verification of Generic Module Algorithms

Finding: The APB converter independently arrived at **identical** data width conversion algorithms.

Validation Points:

1. Slice Extraction (Dnsize):

- APB converter line 334:
`r_s_axi_wdata[r_axi_wr_data_pointer*APBDW +: APBDW]`
- Generic dnsize line 210: `r_data_buffer[r_beat_ptr*NARROW_WIDTH +: NARROW_WIDTH]`
- ✓ **IDENTICAL pattern**

2. Accumulation (Upsize):

- APB converter line 283:
`r_axi_data_shift[r_axi_rsp_data_pointer*APBDW +: APBDW] <= data`
- Generic upsize line 158:
`r_data_accumulator[r_beat_ptr*NARROW_WIDTH +: NARROW_WIDTH] <= data`
- ✓ **IDENTICAL pattern**

3. Pointer Management:

- Both use same increment and wrap logic
- Both use `$clog2(RATIO)` for pointer width
- Both wrap at ratio-1
- ✓ **IDENTICAL pattern**

Significance: Two independent implementations converging on the same solution provides strong evidence that our generic module algorithms are **correct and optimal**.

Code Sections Reference

APB Converter Data Width Conversion Code

Parameters:

```
// Line 39
parameter int AXI2APBRATIO = DW / APBDW,
parameter int PTR_WIDTH = $clog2(AXI2APBRATIO),
```

State Variables:

```
// Line 168
logic [DW-1:0] r_axi_data_shift, w_axi_data_shift;

// Lines 171-173 - THREE separate pointers
logic [PTR_WIDTH-1:0] r_axi_rd_data_pointer, w_axi_rd_data_pointer;
logic [PTR_WIDTH-1:0] r_axi_wr_data_pointer, w_axi_wr_data_pointer;
logic [PTR_WIDTH-1:0] r_axi_rsp_data_pointer, w_axi_rsp_data_pointer;
```

WRITE Path (Dnsize):

```
// Line 334 - Extract data slice
w_apb_cmd_pkt_pwdx = (axi2abpratio == 1) ? r_s_axi_wdata[APBDW-1:0] :
                                              r_s_axi_wdata[r_axi_wr_data_pointer*APBDW +:
APBDW];

// Line 337 - Extract strobe slice
w_apb_cmd_pkt_pstrb = (axi2abpratio == 1) ? r_s_axi_wstrb[APBSW-1:0] :
                                              r_s_axi_wstrb[r_axi_wr_data_pointer*APBSW +:
APBSW];
```

```
// Lines 315-317 - Increment and wrap
r_axi_wr_data_pointer <= r_axi_wr_data_pointer + 1;
if (r_axi_wr_data_pointer == PTR_WIDTH'(axi2abpratio-1))
    r_axi_wr_data_pointer <= 'b0;
```

READ Response Path (Upsize):

```
// Line 283 - Accumulate into shift register
r_axi_data_shift[r_axi_rsp_data_pointer*APBDW +: APBDW] <=
r_apb_rsp_pkt_prdata;
```

```

// Lines 284-287 - Increment and wrap
r_axi_rsp_data_pointer <= r_axi_rsp_data_pointer + 1'b1;
if (r_axi_rsp_data_pointer == PTR_WIDTH'(axi2abratio-1)) begin
    r_axi_rsp_data_pointer <= 'b0;
end

// Line 366 - Combinational accumulation (alternative path)
w_axi_data_shift[r_axi_rsp_data_pointer*APBDW +: APBDW] =
r_apb_rsp_pkt_prdata;

```

Conclusion

Summary

The APB converter implements data width conversion using **patterns identical to our generic modules**, but in a **fundamentally different architectural context**:

- **Generic modules:** Standalone streaming pipeline stages processing complete beats
- **APB converter:** Inline conversion within protocol state machine processing incrementally

Recommendations

1. **Do NOT refactor** APB converter to use generic modules
 - Current inline implementation is optimal for its use case
 - Refactoring would increase complexity without benefits
2. **Keep generic modules separate** for their intended use cases:
 - Pure data width conversion in streaming pipelines
 - Protocol-agnostic width adaptation
 - Reusable components for future designs
3. **Document the relationship** between the two implementations:
 - Both use same fundamental algorithms (validation)
 - Different usage models require different architectures
 - Not all width conversion scenarios suit pipeline approach
4. **Consider future opportunities:**
 - Other protocol converters may benefit from generic modules
 - AXI-to-AXI width converters are prime candidates

- Future designs should evaluate if pipeline model fits

Value Delivered

1. ✓ **Validation** - Independent implementation confirms algorithm correctness
 2. ✓ **Clarity** - Understand why inline vs. pipeline architectures differ
 3. ✓ **Documentation** - Captured design rationale for future reference
 4. ✓ **Guidance** - When to use generic modules vs. inline conversion
-

Related Files

- /mnt/data/github/rtldesignsherpa/projects/components/converters/rtl/axi_data_upsize.sv
- /mnt/data/github/rtldesignsherpa/projects/components/converters/rtl/axi_data_dnsize.sv
- /mnt/data/github/rtldesignsherpa/projects/components/converters/rtl/axi4_to_apb_convert.sv
- /mnt/data/github/rtldesignsherpa/projects/components/converters/USAGE.md

Author: RTL Design Sherpa **Date:** 2025-10-25