RTL Design Sherpa

Bridge Hardware Architecture Specification 1.0

January 3, 2026

Table of Contents

List of Figures

List of Tables

1 Bridge Has Index

Generated: 2026-01-04

2 Document Information

2.1 Bridge Hardware Architecture Specification

Property	Value
Document Title	Bridge Hardware Architecture Specification
Version	1.0
Date	January 3, 2026
Status	Released
Classification	Open Source - Apache 2.0 License

2.2 Revision History

Version	Date	Author	Description
1.0	2026-01-03	RTL Design Sherpa	Initial release - restructured from single spec

2.3 Document Purpose

This Hardware Architecture Specification (HAS) provides a high-level view of the Bridge component, covering:

- System-level architecture and data flow
- Protocol support (AXI4, AXI4-Lite, APB)
- Interface definitions and requirements
- Performance characteristics
- Integration guidelines

For detailed micro-architecture, FSM designs, and signal-level specifications, refer to the companion **Bridge Micro-Architecture Specification (MAS)**.

2.4 Intended Audience

- System architects defining interconnect topology
- Hardware engineers integrating Bridge into SoC designs
- Verification engineers planning test strategies
- Technical managers evaluating Bridge capabilities

2.5 Related Documents

- **Bridge MAS** - Micro-Architecture Specification (detailed block-level)
- **PRD.md** - Product requirements and overview
- **CLAUDE.md** - AI development guide

3 Purpose and Scope

3.1 What is Bridge?

Bridge is a Python-based multi-protocol crossbar generator that creates parameterized SystemVerilog RTL from human-readable configuration files. It generates AXI4 crossbars with automatic support for:

- **Protocol conversion** - AXI4, AXI4-Lite, and APB slave interfaces
- **Width conversion** - Automatic upsize/downsize for data width mismatches
- **Channel-specific masters** - Write-only, read-only, or full AXI4 masters

3.2 The Problem Bridge Solves

Manual AXI4 Crossbar Development is Error-Prone:

Creating an AXI4 crossbar manually requires: 1. Writing 5 separate channel multiplexers (AW, W, B, AR, R) 2. Implementing per-slave arbitration for each channel 3. Managing ID-based response routing for out-of-order support 4. Handling burst locking and interleaving constraints 5. Inserting width converters for data width mismatches 6. Inserting protocol converters for APB/AXI4-Lite mixed systems 7. Wiring hundreds of signals with consistent naming

Bridge Automates All of This: - Define ports and connectivity in configuration files - Run generator script - Get production-ready SystemVerilog RTL

3.3 Scope

This specification covers:

- High-level architecture and block organization

- Supported protocols and conversion capabilities
- Interface requirements and signal conventions
- Performance characteristics and resource estimates
- Integration guidelines and verification strategy

3.4 Out of Scope

The following are covered in the companion MAS:

- Detailed FSM state diagrams and transitions
- Signal-level timing diagrams
- Internal pipeline structures
- RTL implementation details

4 Document Conventions

4.1 Terminology

Table 1.1: Common Terminology

Term	Definition
Master	AXI4 initiator that issues transactions
Slave	AXI4 target that responds to transactions
Crossbar	NxM interconnect connecting masters to slaves
Channel	AXI4 communication path (AW, W, B, AR, R)
Converter	Logic that transforms width or protocol

4.2 Signal Naming

Bridge uses consistent signal prefixes:

Table 1.2: Signal Naming Prefixes

Prefix	Meaning
`s_*`	Slave-side interface (master port on bridge)
`m_*`	Master-side interface (slave port on bridge)
`axi4_*`	AXI4 protocol signals
`apb_*`	APB protocol signals
`cfg_*`	Configuration signals
`dbg_*`	Debug signals

4.3 Diagrams

4.3.1 Figure 1.1: Block Diagram Legend

Block diagrams in this specification use the following conventions:

Table 1.3: Block Diagram Symbols

Symbol	Meaning
Rectangle	Functional block
Arrow	Data/signal flow
Dashed box	Optional/configurable block
Double line	Multi-signal bus

4.4 Code Examples

SystemVerilog code examples are formatted as:

```systemverilog
// Module instantiation example
bridge_4x4 u_bridge (
    .aclk       (sys_clk),
    .aresetn    (sys_rst_n),
    // ... port connections
);
```

4.5 Parameter Notation

Parameters use the following format:

Table 1.4: Parameter Format Example

Parameter	Type	Range	Default	Description
NUM_MASTERS	int	1-32	2	Number of master ports
NUM_SLAVES	int	1-256	2	Number of slave ports

5 Definitions and Acronyms

5.1 Acronyms

Table 1.5: Acronyms and Abbreviations

Acronym	Definition
AMBA	Advanced Microcontroller Bus Architecture
APB	Advanced Peripheral Bus
AR	AXI4 Read Address channel
AW	AXI4 Write Address channel
AXI	Advanced eXtensible Interface
AXI4	AXI Protocol Version 4
B	AXI4 Write Response channel
BID	Bridge ID (internal master identifier)
CAM	Content-Addressable Memory
CDC	Clock Domain Crossing
CSV	Comma-Separated Values
DECERR	Decode Error (AXI response)
DMA	Direct Memory Access
FIFO	First-In First-Out buffer
FSM	Finite State Machine
HAS	Hardware Architecture Specification

Acronym	Definition
ID	Transaction Identifier
MAS	Micro-Architecture Specification
MUX	Multiplexer
NxM	N masters by M slaves (topology notation)
OOO	Out-of-Order
OOR	Out-of-Range
QoS	Quality of Service
R	AXI4 Read Data channel
RTL	Register-Transfer Level
SLVERR	Slave Error (AXI response)
SoC	System-on-Chip
TOML	Tom's Obvious Minimal Language
W	AXI4 Write Data channel

5.2 Definitions

Address Decode: The process of determining which slave should receive a transaction based on the address.

Arbitration: The process of selecting which master gains access to a slave when multiple masters contend.

Bridge ID (BID): Internal identifier prepended to master IDs for response routing. Width = clog2(NUM_MASTERS).

Channel-Specific Master: A master that only uses subset of AXI4 channels (write-only or read-only).

Crossbar: An NxM interconnect allowing any master to communicate with any connected slave.

Grant Locking: Mechanism that holds arbitration grant until transaction completes.

Out-of-Order Response: Responses returned in different order than requests were issued.

Protocol Conversion: Transformation between different bus protocols (e.g., AXI4 to APB).

Response Routing: Directing B/R channel responses back to the originating master.

Round-Robin Arbitration: Fair arbitration where priority rotates among requesters.

Width Conversion: Transformation between different data bus widths (upsize or downsize).

6 Use Cases

6.1 Primary Applications

6.1.1 SoC Interconnects

Bridge serves as the primary interconnect fabric for System-on-Chip designs:

- **Multi-core processor memory subsystems** - Connect multiple CPU cores to shared memory
- **Accelerator integration** - Route GPU, DSP, or custom accelerator traffic
- **DMA engine routing** - Connect DMA controllers to memory and peripherals
- **Peripheral bus bridges** - Convert AXI4 to APB for low-speed peripherals

6.1.2 Memory Systems

Bridge provides flexible memory access:

- **Multi-port DDR controllers** - Multiple masters accessing shared DDR
- **SRAM buffer sharing** - Shared packet buffers for networking
- **Cache coherency interconnects** - Connect cache controllers
- **Memory-mapped I/O** - Unified address space for peripherals

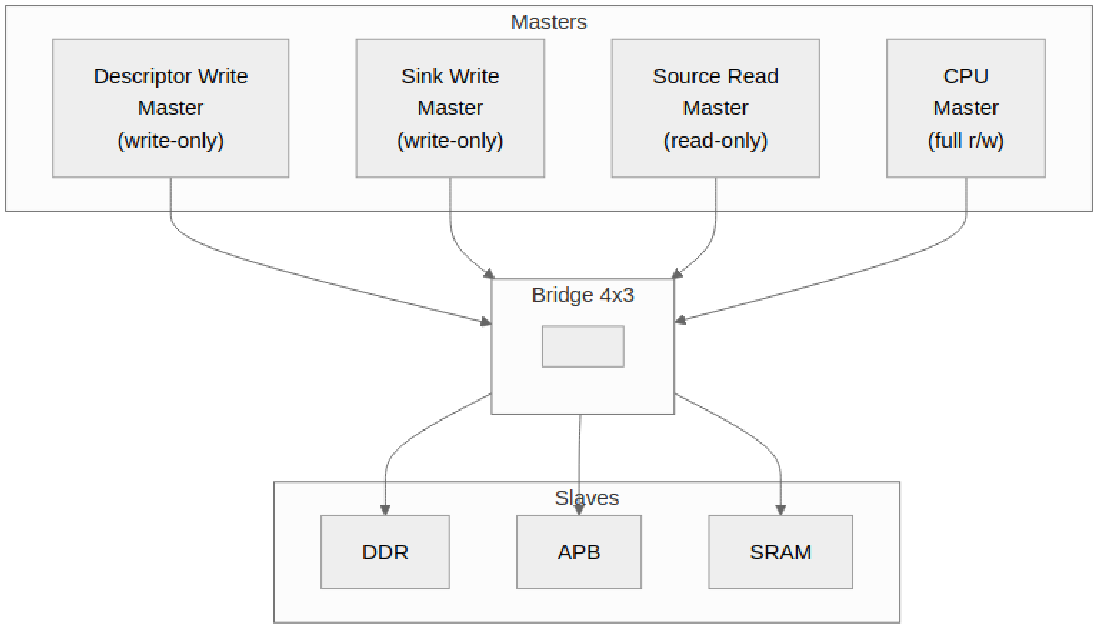
6.1.3 Accelerator Systems

Bridge supports custom accelerator architectures like RAPIDS:

- **Descriptor write masters** - Write control descriptors (write-only)
- **Sink write masters** - Write incoming data packets (write-only)
- **Source read masters** - Read outgoing data packets (read-only)
- **CPU configuration** - Full access for register configuration

6.2 Example: RAPIDS Accelerator

6.2.1 Figure 2.1: RAPIDS System Topology


Masters
Descriptor Write Master (write-only)
Sink Write Master (write-only)
Source Read Master (read-only)
CPU Master (full r/w)
Bridge 4x3
Slaves
DDR
APB
SRAM

RAPIDS System Topology

Bridge topology for RAPIDS accelerator with channel-specific masters.

6.2.2 Configuration

```
[bridge]
  name = "bridge_rapids"
  masters = [
    {name = "rapids_descr_wr", channels = "wr", data_width = 512},
    {name = "rapids_sink_wr", channels = "wr", data_width = 512},
    {name = "rapids_src_rd", channels = "rd", data_width = 512},
    {name = "cpu", channels = "rw", data_width = 64}
  ]
  slaves = [
    {name = "ddr", protocol = "axi4", data_width = 512},
    {name = "apb_periph", protocol = "apb", data_width = 32},
    {name = "sram", protocol = "axi4", data_width = 256}
  ]
```

6.2.3 Benefits

- **39% port reduction** for write-only masters (3 channels vs 5)
- **61% port reduction** for read-only masters (2 channels vs 5)
- **Automatic width conversion** for 64b CPU to 512b DDR

- **Protocol conversion** for APB peripheral access

6.3 Rapid Prototyping

Bridge accelerates design exploration:

- **Quick topology exploration** - Try different master/slave configurations
- **Performance analysis** - Evaluate arbitration and bandwidth
- **Design space exploration** - Compare resource usage
- **Educational projects** - Learn AXI4 protocol and interconnects

7 Key Features

7.1 Configuration-Driven Generation

7.1.1 CSV/TOML Configuration

Bridge uses human-readable configuration files:

- **ports configuration** - Define each master and slave port
- **connectivity matrix** - Specify which masters connect to which slaves
- **Version-control friendly** - Text-based, easy to diff and review

7.1.2 Example Configuration

Port Definition (TOML):

```
masters = [
  {name = "cpu", prefix = "cpu_m_axi", data_width = 64, channels =
"rw"},
  {name = "dma", prefix = "dma_m_axi", data_width = 256, channels =
"rw"}
]
slaves = [
  {name = "ddr", prefix = "ddr_s_axi", data_width = 512, protocol =
"axi4"},
  {name = "uart", prefix = "uart_apb", data_width = 32, protocol =
"apb"}
]
```

Connectivity Matrix (CSV):

```
master\slave,ddr,uart
cpu,1,1
dma,1,0
```

7.2 Multi-Protocol Support

Table 2.1: Supported Protocols

Master Protocol	Slave Protocol	Conversion Type
AXI4	AXI4	Direct or width convert
AXI4	AXI4-Lite	Protocol downgrade
AXI4	APB	Full protocol conversion

7.2.1 Protocol Features

- **AXI4 Full** - Complete 5-channel implementation with bursts
- **AXI4-Lite** - Simplified single-beat transactions
- **APB** - Low-power peripheral access

7.3 Channel-Specific Masters

7.3.1 Write-Only Masters (wr)

Generate only write channels: - AW (Write Address) - W (Write Data) - B (Write Response)

Use case: DMA write engines, descriptor writers

7.3.2 Read-Only Masters (rd)

Generate only read channels: - AR (Read Address) - R (Read Data)

Use case: DMA read engines, source data fetchers

7.3.3 Full Masters (rw)

Generate all five channels for complete read/write capability.

Use case: CPU interfaces, configuration masters

7.3.4 Resource Savings

Table 2.2: Channel-Specific Resource Savings

Master Type	Channels	Signal Reduction
Write-only (wr)	3 of 5	~39% fewer signals
Read-only (rd)	2 of 5	~61% fewer signals
Full (rw)	5 of 5	Baseline

7.4 Automatic Converters

7.4.1 Width Conversion

Bridge automatically inserts width converters:

- **Upsize** - Narrow master to wide slave (e.g., 64b to 512b)
- **Downsize** - Wide master to narrow slave (e.g., 512b to 32b)
- **Per-path** - Only where needed, not global conversion

7.4.2 Protocol Conversion

Bridge inserts protocol converters for mixed systems:

- **AXI4 to APB** - Full conversion with burst handling
- **AXI4 to AXI4-Lite** - Simplified conversion

7.5 Flexible Topology

7.5.1 Scalability

- **Masters:** 1-32 master ports
- **Slaves:** 1-256 slave ports
- **Partial connectivity** - Not all masters need access to all slaves

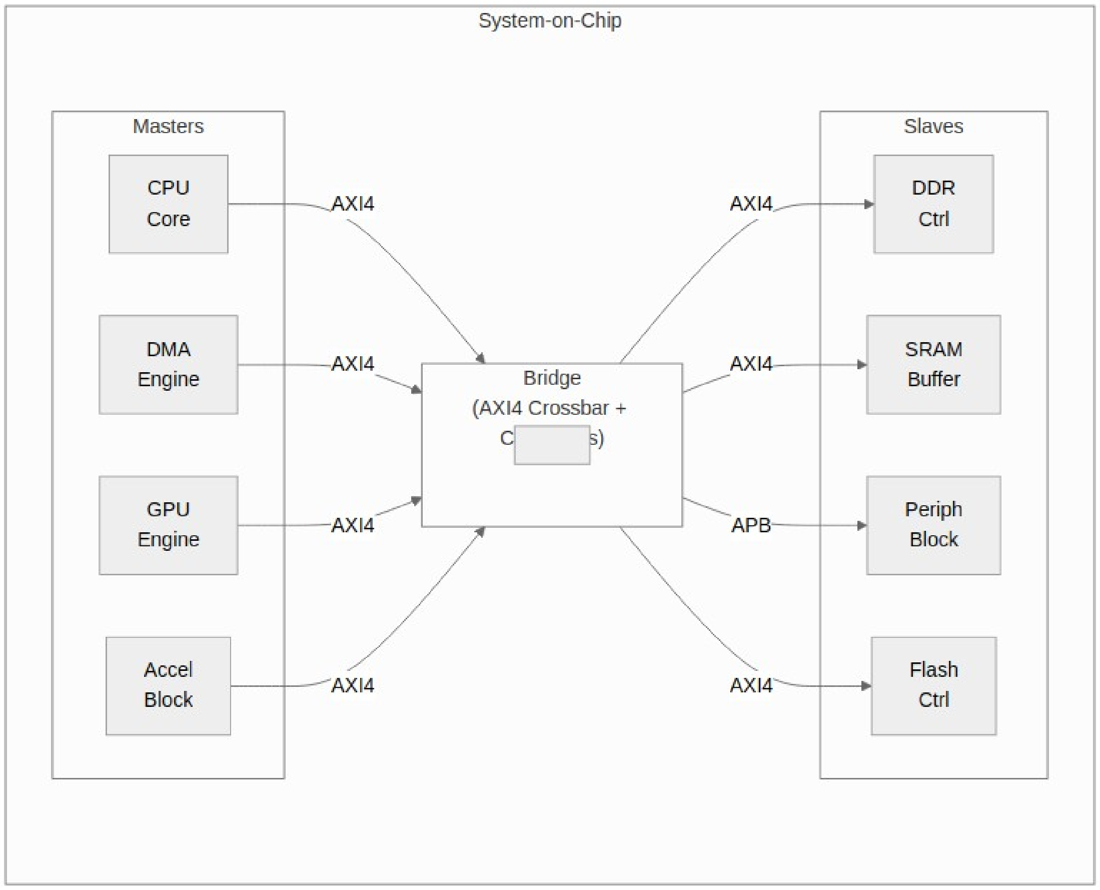
7.5.2 Mixed Configurations

- **Mixed protocols** - AXI4, AXI4-Lite, and APB slaves in same crossbar
- **Mixed data widths** - 32b, 64b, 128b, 256b, 512b
- **Mixed channels** - Write-only, read-only, and full masters

8 System Context

8.1 Bridge in SoC Architecture

8.1.1 Figure 2.2: Bridge System Context



Bridge System Context

Bridge as central interconnect connecting masters to slaves with protocol conversion.

8.2 Interface Boundaries

8.2.1 Master-Side (Upstream)

Bridge presents slave interfaces to upstream masters:

- Accepts AXI4 transactions from masters
- Provides flow control via ready signals
- Returns responses (B/R channels) to correct master

8.2.2 Slave-Side (Downstream)

Bridge presents master interfaces to downstream slaves:

- Issues AXI4/APB transactions to slaves
- Accepts responses from slaves
- Routes responses back through ID tracking

8.3 Address Space

8.3.1 Address Map Organization

Each slave occupies a configurable address region:

Table 2.3: Address Map Organization

Slave	Base Address	Address Range	Size
Slave 0	`BASE + 0x0000_0000`	Configurable	Variable
Slave 1	`BASE + addr_range_0`	Configurable	Variable
Slave N	Sum of previous ranges	Configurable	Variable

8.3.2 Address Decode

- **Parallel decode** - All slaves checked simultaneously
- **One-hot result** - Exactly one slave selected per transaction
- **Out-of-range detection** - DECERR for unmapped addresses

8.4 Clock and Reset

8.4.1 Clock Domain

- **Single clock domain** - All Bridge logic synchronous to `aclk`
- **No CDC** - Masters and slaves must be in same clock domain

8.4.2 Reset

- **Active-low asynchronous** - Standard `aresetn` convention
- **Full reset** - All internal state cleared on reset
- **Transaction safety** - Outstanding transactions aborted on reset

8.5 Dependencies

8.5.1 Required Infrastructure

- AXI4-compliant masters and slaves
- Proper clock and reset distribution
- Address map configuration matching slave address ranges

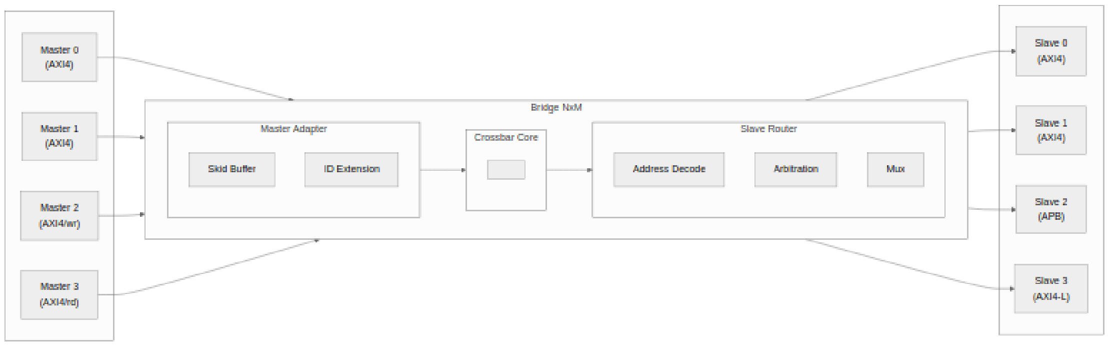
8.5.2 Optional Features

- Width converters (only if widths mismatch)
- Protocol converters (only if protocols differ)
- ID tracking CAM (only if OOO support needed)

9 Block Diagram

9.1 Bridge Architecture Overview

9.1.1 Figure 3.1: Bridge Block Diagram



Bridge Block Diagram

Bridge high-level block diagram showing master adapters, crossbar core, and slave routers.

9.2 Functional Blocks

9.2.1 Master Adapter

Receives transactions from upstream masters:

- **Skid buffer** - Pipeline registration for timing
- **ID extension** - Prepend Bridge ID for response routing
- **Channel separation** - Route AW/W/AR independently

9.2.2 Crossbar Core

Central switching fabric:

- **5-channel routing** - Independent AW, W, B, AR, R paths
- **Address decode** - Determine target slave
- **Arbitration** - Per-slave round-robin selection

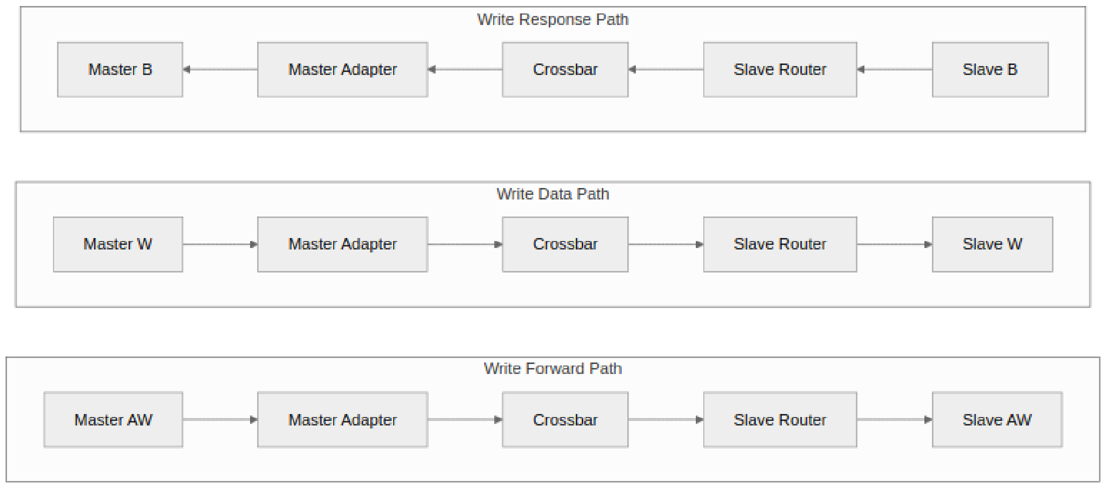
9.2.3 Slave Router

Delivers transactions to downstream slaves:

- **Protocol conversion** - AXI4 to APB/AXI4-Lite if needed
- **Width conversion** - Upsize/downsize data paths
- **Response collection** - Route B/R back to masters

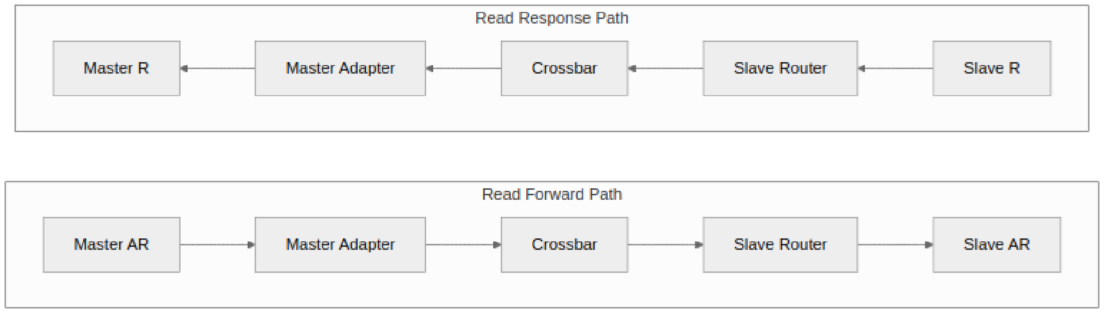
9.3 Data Flow

9.3.1 Figure 3.2: Write Path



Write Path Flow

9.3.2 Figure 3.3: Read Path



Read Path Flow

9.4 Scalability

9.4.1 Topology Parameters

Table 3.1: Topology Parameter Scaling

Parameter	Range	Impact
NUM_MASTERS	1-32	Linear MUX growth
NUM_SLAVES	1-256	Linear decoder growth
DATA_WIDTH	32-512	Linear data path growth
ID_WIDTH	1-16	Logarithmic CAM growth

9.4.2 Resource Scaling

- **Logic:** O(M x N) for crossbar core
- **Routing:** O(M + N) for address/response paths
- **Memory:** O(M x outstanding) for ID tracking

10 Data Flow

10.1 Transaction Flow Overview

10.1.1 Write Transaction Flow

1. **Master issues AW** - Address and control information
2. **Bridge decodes address** - Determines target slave

3. **Arbitration** - Grants access if multiple masters contend
4. **AW forwarded to slave** - With extended ID (Bridge ID prepended)
5. **Master sends W data** - Following the granted AW
6. **W forwarded to slave** - Using same grant
7. **Slave returns B response** - With extended ID
8. **Bridge routes B to master** - Using ID to find originator

10.1.2 Read Transaction Flow

1. **Master issues AR** - Address and control information
2. **Bridge decodes address** - Determines target slave
3. **Arbitration** - Grants access if multiple masters contend
4. **AR forwarded to slave** - With extended ID
5. **Slave returns R data** - Potentially multiple beats
6. **Bridge routes R to master** - Using ID to find originator

10.2 Channel Independence

10.2.1 AXI4 Channel Separation

Each AXI4 channel operates independently:

Table 3.2: AXI4 Channel Summary

Channel	Direction	Contents	Arbitration
AW	Master to Slave	Write address	Per-slave
W	Master to Slave	Write data	Follows AW grant
B	Slave to Master	Write response	ID-based routing
AR	Master to Slave	Read address	Per-slave
R	Slave to Master	Read data	ID-based routing

10.2.2 Parallel Operation

- Read and write transactions can proceed simultaneously
- Different masters can access different slaves in parallel
- Same slave serializes access via arbitration

10.3 ID Extension

10.3.1 Bridge ID (BID) Concept

```
Original Master ID: [ID_WIDTH-1:0]
Extended ID: [BID | Original ID]
BID Width: clog2(NUM_MASTERS)
```

10.3.2 Example: 4 Masters, 4-bit ID

```
Master 0 ID = 4'b0101
Extended ID = 6'b00_0101  (BID=00, prepended)

Master 3 ID = 4'b1100
Extended ID = 6'b11_1100  (BID=11, prepended)
```

10.3.3 ID Flow

```
Master 0: ID=5 --[Extend]--> Slave: ID=0x05 --[Response]--> B.ID=0x05
--[Extract]--> Master 0
Master 1: ID=5 --[Extend]--> Slave: ID=0x15 --[Response]--> B.ID=0x15
--[Extract]--> Master 1
```

10.4 Response Routing

10.4.1 B Channel (Write Response)

1. Slave issues B with extended ID
2. Bridge extracts BID from upper bits
3. BID determines destination master
4. Original ID (lower bits) returned to master

10.4.2 R Channel (Read Data)

1. Slave issues R with extended ID
2. Bridge extracts BID from upper bits
3. BID determines destination master
4. Original ID (lower bits) returned to master
5. RLAST indicates final beat

10.5 Out-of-Order Support

10.5.1 Transaction Interleaving

Bridge supports out-of-order completion:

- Different IDs can complete in any order
- Same ID must complete in order (AXI4 rule)
- ID tracking tables manage outstanding transactions

10.5.2 Example Sequence

```
Time 0: Master 0 issues AR (ID=1) to Slave 0
Time 1: Master 0 issues AR (ID=2) to Slave 1
Time 2: Slave 1 returns R (ID=2) - fast slave
Time 3: Slave 0 returns R (ID=1) - slow slave

Result: ID=2 response arrives before ID=1 - valid OOO
```

11 Protocol Support

11.1 Supported Protocols

11.1.1 AXI4 Full

Complete AMBA AXI4 protocol support:

- **5 channels:** AW, W, B, AR, R
- **Burst transactions:** FIXED, INCR, WRAP
- **Transaction IDs:** Configurable width
- **Out-of-order:** Response reordering supported
- **Data widths:** 32, 64, 128, 256, 512 bits

11.1.2 AXI4-Lite

Simplified AXI4 for register access:

- **5 channels:** Same as AXI4
- **Single-beat only:** AWLEN/ARLEN = 0
- **No IDs:** Transaction ordering by channel
- **Fixed width:** Typically 32 or 64 bits

11.1.3 APB (Advanced Peripheral Bus)

Low-power peripheral protocol:

- **Single channel:** Combined address and data
- **Simple handshake:** PSEL, PENABLE, PREADY
- **No bursts:** Single transfer per transaction

- **Low overhead:** Minimal logic for slow peripherals

11.2 Protocol Conversion Matrix

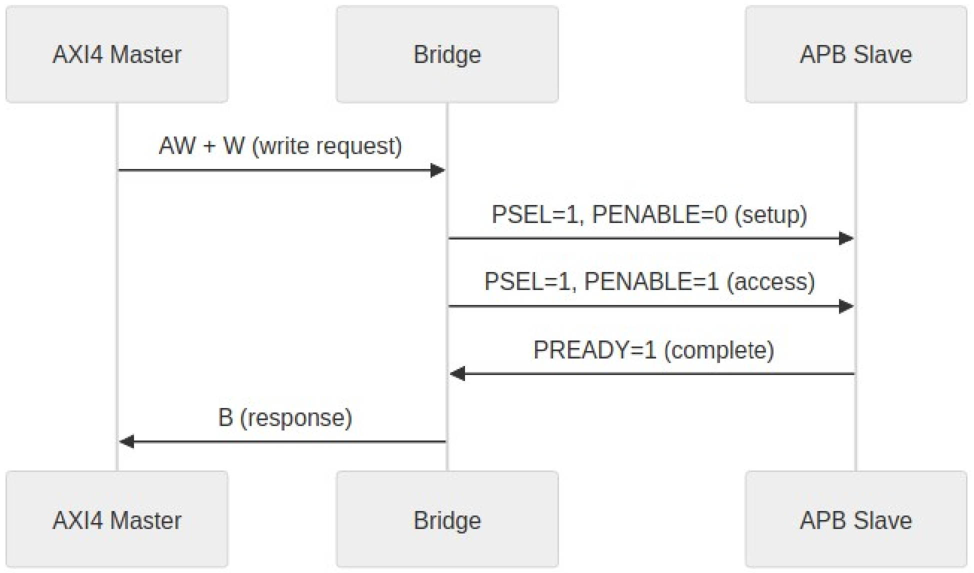
Table 3.3: Protocol Conversion Matrix

Master	Slave	Conversion Type	Notes
AXI4	AXI4	Direct	Width conversion if needed
AXI4	AXI4-Lite	Downgrade	Burst split to single beats
AXI4	APB	Full convert	Channel and timing conversion
AXI4-Lite	AXI4	Upgrade	Single-beat pass-through
AXI4-Lite	AXI4-Lite	Direct	Width conversion if needed
AXI4-Lite	APB	Convert	Simplified conversion

11.3 AXI4 to APB Conversion

11.3.1 Conversion Process

11.3.2 Figure 3.1: AXI4 to APB Conversion Sequence



AXI4 to APB Conversion

11.3.3 Burst Handling

- AXI4 bursts split into individual APB transfers
- WSTRB converted to per-byte enables
- B response generated after all beats complete

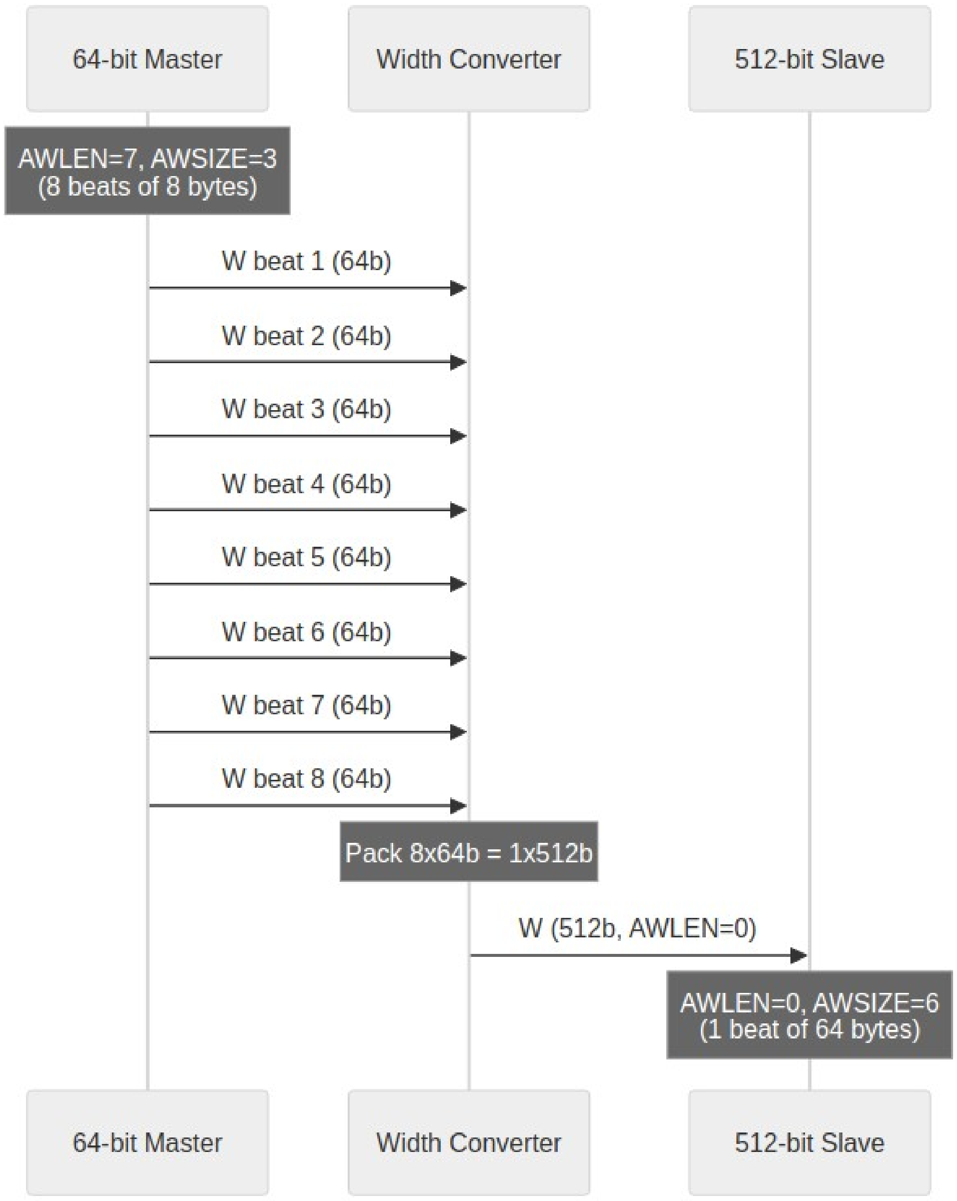
11.3.4 Timing Considerations

- APB is inherently slower (minimum 2 cycles per transfer)
- Pipeline stalls during APB access
- Consider separate APB crossbar for multiple slow peripherals

11.4 Width Conversion

11.4.1 Upsize (Narrow to Wide)

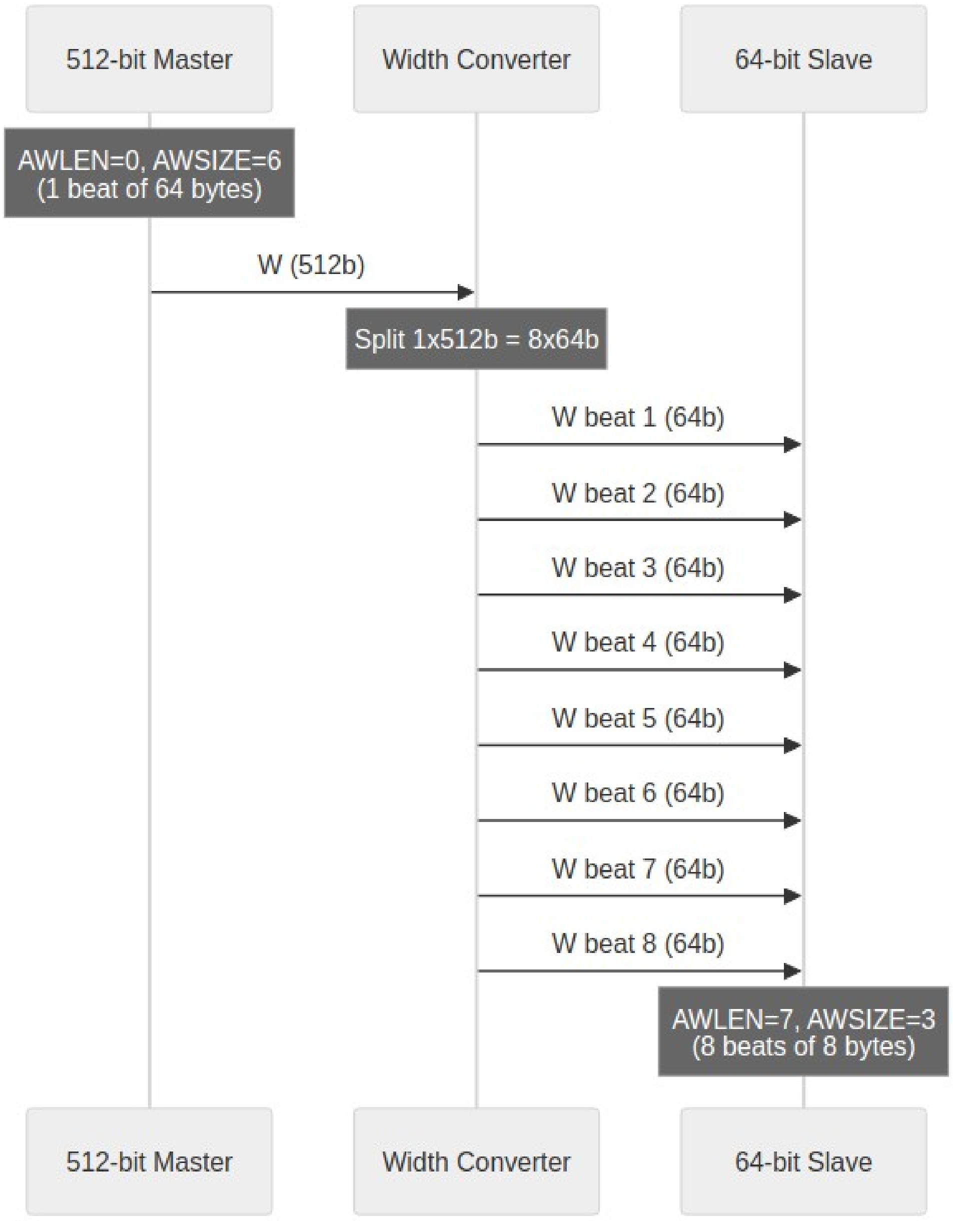
11.4.2 Figure 3.2: Width Upsize Conversion



Width Upsize

11.4.3 Downsize (Wide to Narrow)

11.4.4 Figure 3.3: Width Downsize Conversion



Width Downsize

11.5 Channel-Specific Protocol

11.5.1 Write-Only Masters

Only AW, W, B channels active:

- AR permanently deasserted
- R channel ignored (no routing logic)
- Reduced signal count and logic

11.5.2 Read-Only Masters

Only AR, R channels active:

- AW, W, B channels ignored
- No write arbitration for this master
- Reduced signal count and logic

12 AXI4 Interface Overview

12.1 Interface Organization

12.1.1 Master-Side Interfaces

Bridge presents **slave interfaces** to upstream masters. Each master port includes:

Table 4.9: Master-Side Interface Channels

Channel	Signals	Direction (to Bridge)
AW	AWVALID, AWREADY, AWADDR, AWID, AWLEN, AWSIZE, AWBURST, …	Input
W	WVALID, WREADY, WDATA, WSTRB, WLAST	Input
B	BVALID, BREADY, BID,	Output

Channel	Signals	Direction (to Bridge)
	BRESP	
AR	ARVALID, ARREADY, ARADDR, ARID, ARLEN, ARSIZE, ARBURST, …	Input
R	RVALID, RREADY, RDATA, RID, RRESP, RLAST	Output

12.1.2 Slave-Side Interfaces

Bridge presents **master interfaces** to downstream slaves. Each slave port includes:

Table 4.10: Slave-Side Interface Channels

Channel	Signals	Direction (from Bridge)
AW	AWVALID, AWREADY, AWADDR, AWID, AWLEN, AWSIZE, AWBURST, …	Output
W	WVALID, WREADY, WDATA, WSTRB, WLAST	Output
B	BVALID, BREADY, BID, BRESP	Input
AR	ARVALID, ARREADY, ARADDR, ARID, ARLEN, ARSIZE,	Output

Channel	Signals	Direction (from Bridge)
	ARBURST, …	
R	RVALID, RREADY, RDATA, RID, RRESP, RLAST	Input

12.2 Signal Naming Convention

12.2.1 Custom Prefixes

Each port uses a custom prefix for signal naming:

```
// Example: Master 0 with prefix "cpu_m_axi_"
input  logic [31:0] cpu_m_axi_awaddr,
input  logic [3:0]  cpu_m_axi_awid,
input  logic        cpu_m_axi_awvalid,
output logic        cpu_m_axi_awready,
// ... remaining AW signals

// Example: Slave 0 with prefix "ddr_s_axi_"
output logic [31:0] ddr_s_axi_awaddr,
output logic [7:0]  ddr_s_axi_awid,  // Extended ID
output logic        ddr_s_axi_awvalid,
input  logic        ddr_s_axi_awready,
```

12.2.2 ID Width Extension

Slave-side IDs are wider than master-side:

```
Master ID Width: ID_WIDTH (configuration parameter)
Bridge ID Width: clog2(NUM_MASTERS)
Slave ID Width: ID_WIDTH + clog2(NUM_MASTERS)
```

12.3 Channel-Specific Interfaces

12.3.1 Write-Only Master (wr)

Only AW, W, B channels generated:

```
// Write Address Channel
input  logic [ADDR_WIDTH-1:0] descr_m_axi_awaddr,
input  logic [ID_WIDTH-1:0]   descr_m_axi_awid,
// ... full AW channel

// Write Data Channel
```

```
input  logic [DATA_WIDTH-1:0] descr_m_axi_wdata,
input  logic [STRB_WIDTH-1:0] descr_m_axi_wstrb,
// ... full W channel

// Write Response Channel
output logic [ID_WIDTH-1:0]   descr_m_axi_bid,
output logic [1:0]            descr_m_axi_bresp,
// ... full B channel

// NO AR or R channels
```

12.3.2 Read-Only Master (rd)

Only AR, R channels generated:

```
// Read Address Channel
input  logic [ADDR_WIDTH-1:0] src_m_axi_araddr,
input  logic [ID_WIDTH-1:0]   src_m_axi_arid,
// ... full AR channel

// Read Data Channel
output logic [DATA_WIDTH-1:0] src_m_axi_rdata,
output logic [ID_WIDTH-1:0]   src_m_axi_rid,
// ... full R channel

// NO AW, W, or B channels
```

12.4 Handshake Protocol

12.4.1 Valid/Ready Handshake

All channels use AXI4 valid/ready handshake:

```
Transfer occurs when: VALID && READY
```

12.4.2 Backpressure

- Bridge asserts READY when able to accept
- Masters must hold VALID until READY
- Bridge does not drop transactions

12.5 Burst Support

12.5.1 Supported Burst Types

Table 4.11: Supported Burst Types

AWBURST/ARBURST	Type	Description
2’b00	FIXED	Same address each beat
2’b01	INCR	Incrementing address
2’b10	WRAP	Wrapping at boundary
2’b11	Reserved	Not used

12.5.2 Burst Length

- AWLEN/ARLEN: 8-bit field
- Length = AXLEN + 1 (1 to 256 beats)
- Burst does not cross 4KB boundary (AXI4 rule)

13 APB Interface Overview

13.1 APB Protocol Summary

13.1.1 Signal Definition

Table 4.1: APB Signal Definitions

Signal	Width	Direction	Description
PSEL	1	Output	Slave select
PENABLE	1	Output	Transaction phase
PADDR	ADDR_WIDTH	Output	Address
PWRITE	1	Output	Write enable
PWDATA	DATA_WIDTH	Output	Write data
PSTRB	DATA_WIDTH/8	Output	Byte strobes
PPROT	3	Output	Protection

Signal	Width	Direction	Description
PRDATA	DATA_WIDTH	Input	Read data
PSLVERR	1	Input	Slave error
PREADY	1	Input	Slave ready

13.2 APB Slave Interface

13.2.1 Signal Naming

APB slave ports use custom prefixes:

```
// Example: APB slave with prefix "uart_apb_"
output logic        uart_apb_psel,
output logic        uart_apb_penable,
output logic [31:0] uart_apb_paddr,
output logic        uart_apb_pwrite,
output logic [31:0] uart_apb_pwdata,
output logic [3:0]  uart_apb_pstrb,
output logic [2:0]  uart_apb_pprot,
input  logic [31:0] uart_apb_prdata,
input  logic        uart_apb_pslverr,
input  logic        uart_apb_pready
```

13.3 APB Transaction Timing

13.3.1 Write Transaction

```
        ___     ___     ___     ___
PCLK   |   |___|   |___|   |___|   |___
            _________
PSEL   ____|         |________________
                ____
PENABLE ________|    |________________
            _________
PWRITE  ____|        |________________
       xxxxxx|  ADDR |xxxxxxxxxxxxxxxx
PADDR  xxxxxx|_______|xxxxxxxxxxxxxxxx
       xxxxxx| WDATA |xxxxxxxxxxxxxxxx
PWDATA xxxxxx|_______|xxxxxxxxxxxxxxxx
                ____
PREADY  ________|    |________________
```

13.3.2 Read Transaction

```
        ___     ___     ___     ___
PCLK   |   |___|   |___|   |___|   |___
            _________
```

```
PSEL   ____|         |________________

                _____
PENABLE ________|    |________________
PWRITE  ______________________________
       xxxxxx|  ADDR |xxxxxxxxxxxxxxxx
PADDR  xxxxxx|_______|xxxxxxxxxxxxxxxx

                _____
PREADY  ________|    |________________
       xxxxxxxx|RDATA|xxxxxxxxxxxxxxxx
PRDATA xxxxxxxx|_____|xxxxxxxxxxxxxxxx
```

13.4 AXI4 to APB Conversion

13.4.1 Conversion Requirements

When an AXI4 master accesses an APB slave, Bridge performs:

1. **Burst splitting** - AXI4 bursts become multiple APB transfers
2. **Channel mapping** - AW+W combined into PADDR+PWDATA
3. **Response generation** - APB PSLVERR maps to AXI4 BRESP/RRESP
4. **Timing adaptation** - Insert wait states as needed

13.4.2 Burst Handling

Table 4.2: AXI4 to APB Burst Conversion

AXI4 AWLEN	APB Transfers
0 (1 beat)	1 transfer
1 (2 beats)	2 transfers
N (N+1 beats)	N+1 transfers

13.4.3 Error Mapping

Table 4.3: APB to AXI4 Error Mapping

APB PSLVERR	AXI4 BRESP/RRESP
0 (OK)	2’b00 (OKAY)
1 (Error)	2’b10 (SLVERR)

13.5 Performance Implications

13.5.1 APB Limitations

- **Minimum 2 cycles per transfer** (setup + access phase)

- **No pipelining** (sequential transfers only)
- **No bursts** (each beat requires full handshake)

13.5.2 Design Recommendations

- Use APB only for slow peripherals (UART, GPIO, config registers)
- Group APB slaves behind dedicated sub-crossbar
- Use AXI4/AXI4-Lite for higher-bandwidth slaves

14 Clock and Reset

14.1 Clock Requirements

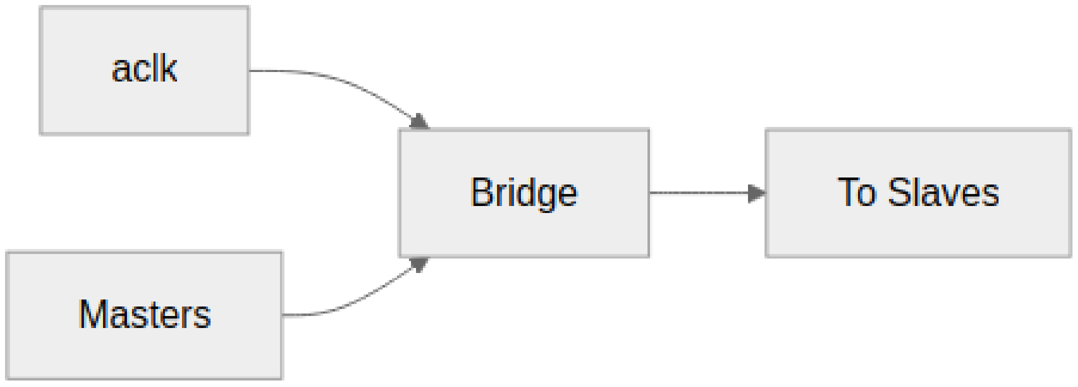
14.1.1 Single Clock Domain

Bridge operates in a single synchronous clock domain:

Table 4.4: Clock Requirements

Signal	Description	Requirements
aclk	System clock	All interfaces synchronous
Frequency	Operating frequency	Design-dependent
Duty cycle	Clock duty cycle	40-60% typical

14.1.2 Figure 4.1: Clock Distribution



Clock Distribution

All masters and slaves must be in the same clock domain as Bridge.

14.1.3 No Clock Domain Crossing

Bridge does not include CDC logic:

- All inputs sampled on aclk rising edge
- All outputs generated on aclk rising edge
- External CDC required if masters/slaves use different clocks

14.2 Reset Requirements

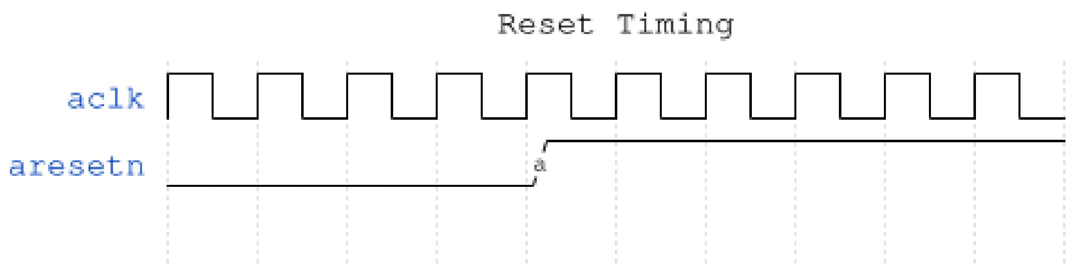
14.2.1 Asynchronous Active-Low Reset

Table 4.5: Reset Signal

Signal	Description	Polarity
aresetn	Async reset	Active-low

14.2.2 Reset Behavior

14.2.3 Figure 4.3: Reset Timing



Reset Timing

The diagram shows the relationship between clock and reset signals. Reset is active-low, meaning the system is in reset when `aresetn = 0`.

14.2.4 Reset Effects

During reset (aresetn = 0):

Table 4.6: Reset Effects

Component	State
Arbiters	Cleared (no grants)
ID tables	Cleared (no outstanding)
FIFOs	Emptied

Component	State
Outputs	Deasserted (VALID = 0)

14.2.5 Outstanding Transactions

Warning: Transactions in flight during reset are lost:

- Master may see timeout (no response)
- Slave may receive partial transaction
- System must ensure quiescence before reset

14.2.6 Reset Release

After reset release (aresetn = 1):

1. Bridge accepts new transactions after 1 cycle
2. All state machines start in IDLE
3. No residual grants or locks

14.3 Timing Constraints

14.3.1 Setup and Hold

All inputs must meet setup/hold relative to aclk:

Table 4.7: Timing Constraints

Constraint	Typical Value
Setup time	Process-dependent
Hold time	Process-dependent

14.3.2 Clock-to-Output

All outputs appear after aclk rising edge:

Table 4.8: Clock-to-Output Delays

Path	Typical Delay
aclk to VALID	1 cycle (registered)
aclk to READY	Combinational
aclk to DATA	1 cycle (registered)

14.3.3 Recommended Constraints

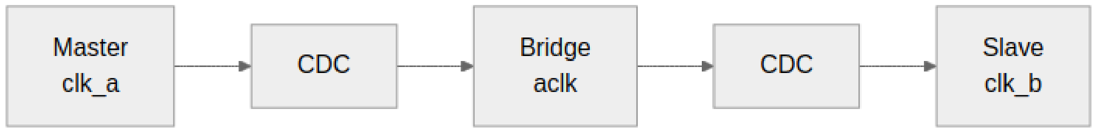
```
# Example SDC constraints
create_clock -period 10 [get_ports aclk]  ;# 100 MHz

# Reset is async - synchronize externally
set_false_path -from [get_ports aresetn]
```

14.4 Multi-Clock Systems

14.4.1 External CDC Required

For systems with multiple clock domains:

14.4.2 Figure 4.2: Multi-Clock CDC



Multi-Clock CDC

14.4.3 CDC Recommendations

- Use AXI4 async FIFO bridges
- Consider AXI4 clock converter IP
- Ensure proper gray-coding for pointers

15 Throughput Characteristics

15.1 Peak Throughput

15.1.1 Single Master to Single Slave

Table 5.1: Peak Throughput by Data Width

Data Width	Frequency	Peak Throughput
32-bit	100 MHz	400 MB/s
64-bit	100 MHz	800 MB/s
128-bit	100 MHz	1.6 GB/s
256-bit	100 MHz	3.2 GB/s
512-bit	100 MHz	6.4 GB/s

15.1.2 Formula

```
Peak Throughput = DATA_WIDTH (bits) × Frequency (Hz) / 8 bytes/bit
```

15.2 Sustained Throughput

15.2.1 Factors Affecting Throughput

Table 5.2: Factors Affecting Throughput

Factor	Impact	Mitigation
Arbitration contention	Reduces per-master throughput	Increase slave ports
Width conversion	1-cycle penalty per direction	Match widths where possible
Protocol conversion	2+ cycles for APB	Use AXI4 for high-bandwidth
Response routing	Minimal (pipelined)	N/A

15.2.2 Multi-Master Scaling

With fair round-robin arbitration:

```
Per-Master Throughput = Peak Throughput / Active Masters (to same
slave)
```

15.2.3 Example: 4 Masters, 2 Slaves

```
All 4 masters accessing Slave 0:
  Per-master = Peak / 4

2 masters to Slave 0, 2 masters to Slave 1:
  Per-master = Peak / 2 (full parallelism)
```

15.3 Burst Efficiency

15.3.1 Burst vs Single-Beat

Table 5.3: Burst Efficiency Comparison

Transaction Type	Overhead	Efficiency
Single beat	1 cycle address + 1 cycle data	50%
4-beat burst	1 cycle address + 4 cycle data	80%

Transaction Type	Overhead	Efficiency
16-beat burst	1 cycle address + 16 cycle data	94%
256-beat burst	1 cycle address + 256 cycle data	99.6%

15.3.2 Recommendation

- Use bursts whenever possible
- AXI4 supports up to 256 beats per burst
- APB does not support bursts (split internally)

15.4 Width Conversion Impact

15.4.1 Upsize (Narrow to Wide)

```
64-bit to 512-bit (8:1 ratio):
  Input: 8 beats × 64 bits = 512 bits
  Output: 1 beat × 512 bits = 512 bits

Throughput: Preserved (same data, fewer beats)
Latency: +1 cycle (packing delay)
```

15.4.2 Downsize (Wide to Narrow)

```
512-bit to 64-bit (8:1 ratio):
  Input: 1 beat × 512 bits = 512 bits
  Output: 8 beats × 64 bits = 512 bits

Throughput: Preserved (same data, more beats)
Latency: +7 cycles (sequential output)
```

15.5 Protocol Conversion Impact

15.5.1 AXI4 to APB

Table 5.4: AXI4 vs APB Protocol Impact

Metric	AXI4 Direct	AXI4 to APB
Min cycles/transfer	1	2
Burst support	Yes	No (split)
Pipeline depth	1+	1
Peak throughput	DATA_WIDTH/cycle	DATA_WIDTH/2

Metric	AXI4 Direct	AXI4 to APB
		cycles

15.5.2 Recommendation

- Keep high-bandwidth traffic on AXI4 paths
- Use APB only for low-bandwidth peripherals
- Group APB slaves to avoid impacting AXI4 traffic

16 Latency Analysis

16.1 Latency Components

16.1.1 Address Path Latency

Table 5.5: Address Path Latency

Stage	Cycles	Notes
Master Adapter (skid)	1	Pipeline registration
Address Decode	0	Combinational
Arbitration	0-1	0 if no contention, 1 if arbitrate
Slave Router	1	Pipeline registration
Total Address	**2-3**	Best/worst case

16.1.2 Data Path Latency

Table 5.6: Data Path Latency

Stage	Cycles	Notes
W follows AW	0	Same grant
Width conversion	0-1	0 if match, 1 if convert
Protocol conversion	0-2+	0 for AXI4, 2+ for APB
Total Data	**0-3+**	Best/worst case

16.1.3 Response Path Latency

Table 5.7: Response Path Latency

Stage	Cycles	Notes
Slave response	Variable	Slave-dependent
ID extraction	0	Combinational
Response routing	1	Pipeline registration
Master delivery	0	Direct connection
Total Response	**1 + slave**	Plus slave latency

16.2 End-to-End Latency

16.2.1 Write Transaction (Best Case)

```
Cycle 0: AW arrives at Bridge
Cycle 1: AW exits to Slave (skid buffer)
Cycle 2: Slave receives AW
Cycle 1: W arrives (same cycle as AW skid)
Cycle 2: W exits to Slave
Cycle 3: Slave receives W, generates B
Cycle 4: B routed back to Master

Total: 4 cycles (minimum)
```

16.2.2 Read Transaction (Best Case)

```
Cycle 0: AR arrives at Bridge
Cycle 1: AR exits to Slave (skid buffer)
Cycle 2: Slave receives AR, generates R
Cycle 3: R routed back to Master

Total: 3 cycles (minimum)
```

16.2.3 Contention Latency

When multiple masters contend for same slave:

```
Arbitration adds 1 cycle per contending master

Example: 4 masters requesting same slave
  Worst case: Wait for 3 other masters = 3 extra cycles
  Average case: Wait for 1.5 masters = 1-2 extra cycles
```

16.3 Width Conversion Latency

16.3.1 Upsize Latency

Table 5.8: Upsize Latency by Ratio

Ratio	Extra Cycles	Notes
1:1	0	No conversion
2:1	0	Pack on fly
4:1	0	Pack on fly
8:1	0-1	May need accumulation

16.3.2 Downsize Latency

Table 5.9: Downsize Latency by Ratio

Ratio	Extra Cycles	Notes
1:1	0	No conversion
1:2	+1	2 output beats
1:4	+3	4 output beats
1:8	+7	8 output beats

16.4 Protocol Conversion Latency

16.4.1 AXI4 to APB

Table 5.10: AXI4 to APB Conversion Latency

Phase	Cycles
AXI4 AW decode	1
APB setup	1
APB access	1+ (PREADY)
B generation	1
Total	**4+ cycles**

16.4.2 APB Wait States

APB slaves may insert wait states:

```
PREADY deasserted: +1 cycle per wait
Typical UART/GPIO: 0-2 wait states
Slow peripherals: May have many wait states
```

16.5 Latency Optimization

16.5.1 Design Recommendations

1. **Match data widths** - Avoid conversion latency
2. **Use AXI4 for fast paths** - Avoid APB latency
3. **Minimize contention** - Spread traffic across slaves
4. **Use bursts** - Amortize address latency
5. **Pipeline responses** - Don't block on slow slaves

17 Resource Estimates

17.1 FPGA Resource Summary

17.1.1 Baseline Configuration (2x2, 64-bit)

Table 5.11: Baseline Resource Requirements

Resource	Count	Notes
LUTs	~2,000-3,000	Logic and routing
Registers	~1,500-2,000	Pipeline stages
Block RAM	0-1	Optional ID CAM
DSP	0	No arithmetic

17.1.2 Scaling Factors

Table 5.12: Resource Scaling Factors

Component	Scaling
Crossbar core	O(M x N)
Master adapters	O(M)
Slave routers	O(N)
ID tracking	O(M x Outstanding)
Width converters	O(per-path ratio)

17.2 Component Breakdown

17.2.1 Per-Master Resources

Table 5.13: Per-Master Resource Breakdown

Component	LUTs	Registers
Skid buffer	~50-100	~DATA_WIDTH
ID extension	~20	~ID_WIDTH
Channel mux	~100	~50
Per Master Total	~200-300	~DATA_WIDTH + 100

17.2.2 Per-Slave Resources

Table 5.14: Per-Slave Resource Breakdown

Component	LUTs	Registers
Address decode	~50	0
Arbiter	~100-200	~50
Response mux	~100	~50
Per Slave Total	~250-350	~100

17.2.3 Crossbar Core

Table 5.15: Crossbar Core Resources

Component	LUTs	Registers
AW mux (N-way)	~100 x N	~50 x N
W mux (N-way)	~100 x N	~50 x N
B demux (M-way)	~50 x M	~50 x M
AR mux (N-way)	~100 x N	~50 x N
R demux (M-way)	~50 x M	~50 x M

17.3 Width Converter Resources

Table 5.16: Width Converter Resources

Ratio	LUTs	Registers	Notes
1:2 upsize	~200	~DATA_IN	Pack logic
1:4 upsize	~300	~DATA_IN	Pack logic

Ratio	LUTs	Registers	Notes
1:8 upsize	~400	~DATA_IN	Pack logic
2:1 downsize	~200	~DATA_OUT	Split logic
4:1 downsize	~300	~DATA_OUT	Split logic
8:1 downsize	~400	~DATA_OUT	Split logic

17.4 Protocol Converter Resources

17.4.1 AXI4 to APB

Table 5.17: AXI4 to APB Converter Resources

Component	LUTs	Registers
FSM	~100	~20
Burst counter	~50	~8
Data buffer	~100	~DATA_WIDTH
Response logic	~50	~10
Total	~300	~DATA_WIDTH + 50

17.5 Example Configurations

Table 5.18: Complete Bridge Resource Estimates

Config	Masters	Slaves	Data	LUTs	Registers
2x2 Basic	2	2	64	~2,500	~1,500
4x4 Standard	4	4	64	~5,000	~3,000
4x4 Wide	4	4	256	~8,000	~5,000
8x8 Large	8	8	128	~12,000	~8,000
RAPIDS (4x3)	4	3	512	~10,000	~6,000

17.5.1 Notes

- Estimates include all converters and adapters
- Actual results vary by synthesis tool and FPGA family
- Block RAM usage depends on ID table implementation

- DSP usage is zero (no multiplication/division)

17.6 Optimization Strategies

17.6.1 Resource Reduction

1. **Use channel-specific masters** - 40-60% port reduction
2. **Match data widths** - Avoid converter logic
3. **Use APB sparingly** - Reduces AXI4 overhead
4. **Limit outstanding transactions** - Smaller ID tables

17.6.2 Performance/Resource Trade-off

Table 5.19: Performance/Resource Trade-offs

Feature	Resource Impact	Performance Impact
Deeper skid buffers	+registers	+timing margin
Larger ID tables	+BRAM	+OOO depth
Pipeline stages	+registers	+frequency
Wider data paths	+routing	+throughput

18 System Requirements

18.1 Hardware Requirements

18.1.1 Clock Infrastructure

Table 6.1: Clock Requirements

Requirement	Specification
Clock source	Single clock domain (aclk)
Clock quality	Low jitter, stable frequency
Clock distribution	Must reach all Bridge ports

18.1.2 Reset Infrastructure

Table 6.2: Reset Requirements

Requirement	Specification
Reset type	Asynchronous, active-low (aresetn)

Requirement	Specification
Reset synchronization	External synchronizer recommended
Reset distribution	Must reach all Bridge ports

18.1.3 Power

Table 6.3: Power Requirements

Requirement	Specification
Power domains	Single domain (no power gating)
Voltage	FPGA/ASIC process-dependent

18.2 Interface Requirements

18.2.1 Master Requirements

Masters connecting to Bridge must:

1. **Comply with AXI4 protocol** - Valid/ready handshake
2. **Use correct ID width** - Match configured ID_WIDTH
3. **Use correct data width** - Match configured DATA_WIDTH
4. **Use correct address width** - Match configured ADDR_WIDTH
5. **Stay within address range** - Target valid slave addresses

18.2.2 Slave Requirements

Slaves connecting to Bridge must:

1. **Comply with AXI4/APB protocol** - Based on configuration
2. **Accept extended IDs** - ID_WIDTH + BID_WIDTH
3. **Match configured data width** - Or use width conversion
4. **Respond to all transactions** - No hanging

18.3 Configuration Requirements

18.3.1 Address Map

Table 6.4: Address Map Requirements

Requirement	Specification
Slave ranges	Must not overlap

Requirement	Specification
Range alignment	Power-of-2 recommended
Coverage	All master addresses must map to slaves or OOR

18.3.2 Connectivity

Table 6.5: Connectivity Requirements

Requirement	Specification
Matrix completeness	All masters must access at least one slave
Reachability	Consider traffic patterns for performance

18.4 Timing Requirements

18.4.1 Setup/Hold

Table 6.6: Timing Requirements

Constraint	Requirement
Input setup	Meet FPGA/ASIC requirements
Input hold	Meet FPGA/ASIC requirements
Clock-to-output	Per process characterization

18.4.2 Recommended Margins

Table 6.7: Recommended Timing Margins

Margin	Value	Purpose
Setup margin	10-20%	Variation tolerance
Hold margin	Positive	No hold violations
Clock uncertainty	Include in constraints	PLL/MMCM jitter

18.5 Verification Requirements

18.5.1 Pre-Integration Checks

1. **RTL lint clean** - No Verilator warnings

2. **Parameter validation** - All parameters in valid range
3. **Address map review** - No overlaps, complete coverage
4. **ID width calculation** - Sufficient for NUM_MASTERS

18.5.2 Integration Checks

1. **Signal connectivity** - All ports connected
2. **Width matching** - DATA_WIDTH, ADDR_WIDTH, ID_WIDTH
3. **Protocol matching** - AXI4 vs APB correctly configured
4. **Clock/reset connectivity** - All ports share same domain

18.5.3 Post-Integration Checks

1. **Basic transaction test** - Read/write to each slave
2. **Arbitration test** - Multi-master contention
3. **Error handling test** - OOR address response
4. **Performance validation** - Meet throughput targets

19 Parameter Configuration

19.1 Core Parameters

Table 6.8: Bridge Core Parameters

Parameter	Type	Range	Default	Description
NUM_MASTERS	int	1-32	2	Number of master ports
NUM_SLAVES	int	1-256	2	Number of slave ports
ADDR_WIDTH	int	12-64	32	Address bus width
DATA_WIDTH	int	32-512	64	Data bus width
ID_WIDTH	int	1-16	4	Master ID width
USER_WIDTH	int	1-16	1	User signal width

19.2 Derived Parameters

19.2.1 Calculated by Generator

Table 6.9: Derived Parameters

Parameter	Formula	Example
BID_WIDTH	clog2(NUM_MASTERS)	4 masters = 2 bits
TOTAL_ID_WIDTH	ID_WIDTH + BID_WIDTH	4 + 2 = 6 bits
STRB_WIDTH	DATA_WIDTH / 8	64b = 8 strobes

19.3 Per-Port Configuration

19.3.1 Master Port Configuration

Table 6.10: Master Port Configuration

Field	Type	Description
name	string	Unique identifier
prefix	string	Signal prefix
channels	enum	“rw”, “wr”, or “rd”
data_width	int	Port data width
id_width	int	Port ID width

19.3.2 Slave Port Configuration

Table 6.11: Slave Port Configuration

Field	Type	Description
name	string	Unique identifier
prefix	string	Signal prefix
protocol	enum	“axi4”, “axi4lite”, “apb”
data_width	int	Port data width
base_addr	hex	Base address
addr_range	hex	Address range size

19.4 Configuration File Format

19.4.1 TOML Configuration

```
[bridge]
  name = "my_bridge"
  description = "Example bridge configuration"

  defaults.skid_depths = {ar = 2, r = 4, aw = 2, w = 4, b = 2}

  masters = [
    {name = "cpu", prefix = "cpu_m_axi", channels = "rw",
     id_width = 4, addr_width = 32, data_width = 64, user_width = 1},
    {name = "dma", prefix = "dma_m_axi", channels = "rw",
     id_width = 4, addr_width = 32, data_width = 256, user_width = 1}
  ]

  slaves = [
    {name = "ddr", prefix = "ddr_s_axi", protocol = "axi4",
     id_width = 6, addr_width = 32, data_width = 512, user_width = 1,
     base_addr = 0x80000000, addr_range = 0x40000000},
    {name = "uart", prefix = "uart_apb", protocol = "apb",
     addr_width = 12, data_width = 32,
     base_addr = 0x00000000, addr_range = 0x00001000}
  ]
```

19.4.2 CSV Connectivity

```
master\slave,ddr,uart
cpu,1,1
dma,1,0
```

19.5 Parameter Validation

19.5.1 Generator Checks

Table 6.12: Generator Validation Checks

Check	Error Condition
NUM_MASTERS	< 1 or > 32
NUM_SLAVES	< 1 or > 256
DATA_WIDTH	Not power of 2
Address overlap	Slave ranges intersect
Connectivity	Master with no slaves
ID width	Insufficient for masters

19.5.2 Runtime Validation

Bridge includes optional assertions for:

- Address alignment
- Burst boundary crossing
- Protocol violations
- ID mismatch

19.6 Example Configurations

19.6.1 Simple 2x2

```
[bridge]
  name = "bridge_simple"
  masters = [
    {name = "m0", prefix = "m0_axi", data_width = 64},
    {name = "m1", prefix = "m1_axi", data_width = 64}
  ]
  slaves = [
    {name = "s0", prefix = "s0_axi", base_addr = 0x0, addr_range =
0x10000000},
    {name = "s1", prefix = "s1_axi", base_addr = 0x10000000,
addr_range = 0x10000000}
  ]
```

19.6.2 Mixed Protocol

```
[bridge]
  name = "bridge_mixed"
  masters = [
    {name = "cpu", prefix = "cpu_axi", data_width = 64, channels =
"rw"}
  ]
  slaves = [
    {name = "mem", prefix = "mem_axi", protocol = "axi4", data_width =
512},
    {name = "gpio", prefix = "gpio_apb", protocol = "apb", data_width
= 32},
    {name = "uart", prefix = "uart_apb", protocol = "apb", data_width
= 32}
  ]
```

20 Verification Strategy

20.1 Verification Levels

20.1.1 Unit Level

Test individual Bridge components:

Table 6.13: Unit Level Test Focus

Component	Test Focus
Master Adapter	Skid buffer, ID extension
Slave Router	Address decode, arbitration
Width Converter	Upsize/downsize correctness
Protocol Converter	AXI4 to APB timing
Response Router	ID extraction, routing

20.1.2 Integration Level

Test complete Bridge functionality:

Table 6.14: Integration Test Coverage

Test Category	Coverage
Address routing	All master-slave paths
Arbitration	Contention, fairness
ID tracking	Response routing
Error handling	OOR, timeouts
Performance	Throughput, latency

20.1.3 System Level

Test Bridge in target system:

Table 6.15: System Level Tests

Test Category	Focus
End-to-end	CPU to memory
Multi-master	DMA + CPU
Mixed protocol	AXI4 + APB

Test Category	Focus
Stress	Maximum traffic

20.2 Test Infrastructure

20.2.1 CocoTB Framework

Bridge uses CocoTB for verification:

```
from CocoTBFramework.components.axi4 import AXI4Master, AXI4Slave

@cocotb.test()
async def test_basic_write(dut):
    tb = BridgeTB(dut)
    await tb.setup_clocks_and_reset()

    # Write transaction
    await tb.master[0].write(addr=0x1000, data=0xDEADBEEF)

    # Verify
    assert tb.slave[0].mem[0x1000] == 0xDEADBEEF
```

20.2.2 Test Categories

Table 6.16: Test Categories

Category	Description	Example
Basic	Single transactions	Read/write
Burst	Multi-beat transactions	16-beat burst
Concurrent	Multiple masters active	M0 + M1
Stress	Maximum utilization	Back-to-back
Error	Error conditions	OOR address

20.3 Coverage Goals

20.3.1 Functional Coverage

Table 6.17: Functional Coverage Goals

Category	Target
Address paths	100% master-slave

Category	Target
	combinations
Transaction types	Read, write, burst
ID values	Full ID range
Data patterns	Walking 1s/0s, random

20.3.2 Code Coverage

Table 6.18: Code Coverage Goals

Metric	Target
Line coverage	>95%
Branch coverage	>90%
FSM coverage	100% state transitions
Toggle coverage	>80% signals

20.4 Debug Features

20.4.1 Waveform Capture

Bridge tests support VCD/FST waveform dumps:

```
pytest test_bridge.py --vcd=waves.vcd
gtkwave waves.vcd
```

20.4.2 Debug Signals

Bridge includes optional debug signals:

Table 6.19: Debug Signals

Signal	Description
dbg_aw_grant	Current AW grant
dbg_ar_grant	Current AR grant
dbg_outstanding	Outstanding transaction count
dbg_state	FSM states

20.5 Regression Testing

20.5.1 Continuous Integration

Table 6.20: CI Test Schedule

Stage	Tests	Frequency
Pre-commit	Basic sanity	Each commit
Nightly	Full regression	Daily
Weekly	Performance	Weekly

20.5.2 Test Matrix

Table 6.21: Configuration Test Matrix

Config	Data Width	Protocol	Tested
2x2	64	AXI4	Yes
4x4	64	AXI4	Yes
4x4	256	AXI4	Yes
2x2	64	AXI4+APB	Yes
RAPIDS	512	AXI4+APB	Yes

20.6 Known Limitations

20.6.1 Test Gaps

Table 6.22: Known Test Gaps

Gap	Status	Plan
APB burst stress	Partial	Phase 3
Width downsize stress	Partial	Planned
32-master config	Not tested	Low priority

20.6.2 Simulator Support

Table 6.23: Simulator Support Status

Simulator	Status
Verilator	Fully supported
Icarus Verilog	Supported

Simulator	Status
Commercial (VCS, Questa)	Not tested

21 Bridge Micro-Architecture Specification Index

Component: Bridge (Multi-Protocol Crossbar Generator) **Version:** 1.0 **Date:** 2026-01-03 **Purpose:** Detailed micro-architecture specification for Bridge component

---

21.1 Document Organization

This specification covers the Bridge component - a CSV-driven generator that creates AXI4 crossbars with automatic width conversion, protocol conversion (AXI4, AXI4-Lite, APB), and channel-specific master support (read-only, write-only, or full).

21.2 Quick Navigation

21.2.1 For New Users

1. Start with Overview for understanding Bridge capabilities
2. Read Block Diagram to understand the architecture
3. Study Arbitration for operational details
4. Reference Module Structure for generated RTL

21.2.2 For Integration

- **Protocol support:** See Protocol Conversion for AXI4/APB/AXI-Lite
- **Width handling:** See Width Conversion for data width mismatches
- **ID management:** See ID Management for transaction tracking
- **Error handling:** See Error Handling for OOR and timeout

21.3 Visual Assets

All diagrams referenced in the documentation are available in:

- **Source Files:**
  - `assets/graphviz/*.gv` - Graphviz source diagrams
  - `assets/puml/*.puml` - PlantUML FSM diagrams
- **Rendered Files:**

- `assets/graphviz/*.png` - Rendered block diagrams
- `assets/puml/*.png` - Rendered FSM diagrams

21.3.1 Architecture Diagrams

1. **Master Adapter** - assets/graphviz/master_adapter.png
2. **Slave Router** - assets/graphviz/slave_router.png
3. **Crossbar Core** - assets/graphviz/crossbar_core.png
4. **ID Management** - assets/graphviz/id_management.png
5. **Width Conversion** - assets/graphviz/width_conversion.png
6. **Protocol Conversion** - assets/graphviz/protocol_conversion_apb.png
7. **Response Routing** - assets/graphviz/response_routing.png
8. **Error Handling** - assets/graphviz/error_handling.png

21.3.2 FSM Diagrams

1. **AW Arbiter FSM** - assets/puml/aw_arbiter_fsm.png
2. **AR Arbiter FSM** - assets/puml/ar_arbiter_fsm.png

21.4 Component Overview

21.4.1 Key Features

- **Multi-Protocol Support:** AXI4, AXI4-Lite, and APB slave conversion
- **CSV-Driven Configuration:** Human-readable port and connectivity definitions
- **Channel-Specific Masters:** Write-only (wr), read-only (rd), or full (rw) support
- **Automatic Width Conversion:** Upsize/downsize for data width mismatches
- **Out-of-Order Support:** ID-based response routing with CAM tracking
- **Custom Signal Prefixes:** Unique prefixes per port for clean integration

21.4.2 Protocol Conversion Matrix

Master Protocol	Slave Protocol	Conversion
AXI4	AXI4	Direct or width convert
AXI4	AXI4-Lite	Protocol downgrade

Master Protocol	Slave Protocol	Conversion
AXI4	APB	Full protocol conversion

21.4.3 Design Philosophy

Efficient Multi-Width Routing: - Direct connections where widths match (zero conversion overhead) - Width converters only where needed - No fixed internal crossbar width

Resource Optimization: - Channel-specific masters reduce port count by 40-60% - Per-path width converters instead of global conversion - Minimal logic for matching-width connections

---

21.5 Related Documentation

21.5.1 Companion Specifications

- Bridge HAS - Hardware Architecture Specification (high-level)

21.5.2 Project-Level

- **PRD.md:** ../../PRD.md - Complete product requirements document
- **CLAUDE.md:** ../../CLAUDE.md - AI assistant integration guide

21.5.3 Generator

- **Generator:** `../../bin/bridge_csv_generator.py` - CSV-based generator script
- **Test Configs:** `../../bin/test_configs/` - Example TOML/CSV configurations

21.6 Version History

Version 1.0 (2026-01-03): - Initial specification release - Restructured from single spec to HAS/MAS format - Complete block-level documentation - FSM and ID management details

---

Last Updated: 2026-01-03 **Maintained By:** RTL Design Sherpa Project

22 Bridge: AXI4 Full Crossbar Generator - Product Requirements Document

Project: Bridge **Version:** 2.1 **Status:** 🟢 Phase 2 Complete - Simplified Architecture with Hard Limits **Created:** 2025-10-18 **Last Updated:** 2025-11-02

22.1 Executive Summary

Bridge is a Python-based AXI4 crossbar generator that produces simple, performant SystemVerilog RTL for connecting multiple AXI4 masters to multiple AXI4 slaves. The name follows the infrastructure theme - bridges connect different regions, enabling communication across divides, just like crossbars connect masters and slaves.

Design Philosophy: A simple AMBA fabric that is performant, but makes no attempt to support all features. We enforce hard limits (8-bit ID width, 64-bit address width) to eliminate unnecessary complexity, focusing only on what real hardware needs.

Key Differentiator from Delta: - **Delta:** AXI-Stream crossbar (streaming data, single channel, simple routing) - **Bridge:** AXI4 full crossbar (memory-mapped, 5 channels, burst support, ID-based routing)

Key Differentiator from Commercial IP: - **Commercial AXI4 Crossbars:** Feature-complete, support every AXI4 edge case, complex configurability - **Bridge:** Simple and performant, supports common use cases, hard limits for simplicity

Target Use Case: High-performance memory-mapped interconnects for multi-core processors, accelerators, and memory controllers where simplicity and performance matter more than spec compliance.

22.2 Design Philosophy

Vision: A simple AMBA fabric that is performant, but makes no attempt to support all features.

22.2.1 Core Principles

1. Simplicity Over Completeness - Focus on common use cases, not edge cases - Implement what's needed for real hardware, not AXI4 spec compliance theater - Prefer straightforward implementations over complex feature sets

2. Performance First - Direct connections where possible (matching widths = zero overhead) - Minimal conversion logic in critical paths - Optimize for throughput and latency, not feature count

3. Hard Limits for Simplicity

We enforce uniform widths on certain parameters to eliminate complexity:

Parameter	Hard Limit	Rationale
ID Width	8 bits (fixed)	Eliminates ID width conversion logic. 256 unique IDs is sufficient for all realistic use cases.
Address Width	64 bits (fixed)	Eliminates address width conversion. 64-bit addressing covers all memory-mapped systems.
Data Width	Variable (32b-512b)	Width converters ONLY for data. Commonly needed for bandwidth optimization.

Why This Works: - ID width conversion adds complexity with minimal benefit (nobody needs >256 outstanding transactions per master) - Address width conversion is rarely needed (peripheral addresses fit in 32-bit, but using 64-bit everywhere is simpler) - Data width conversion is the ONLY width conversion that provides real value (bandwidth matching)

4. What We DON'T Support (By Design)

Features intentionally excluded for simplicity: - ✗ Variable ID width per port - ✗ Variable address width per port - ✗ AXI4-Lite protocol variant (use standard AXI4 with len=0) - ✗ ACE protocol extensions (cache coherency) - ✗ AXI5 features - ✗ Complete AXI4 sideband signal support (QoS, Region, User signals declared but not routed)

5. What We DO Support

Core AXI4 features that matter: - √ Full 5-channel AXI4 protocol (AW, W, B, AR, R) - √ Burst transactions (INCR, WRAP, FIXED) - √ Out-of-order completion via transaction IDs - √ Multiple outstanding transactions per master - √ Channel-specific masters (write-only, read-only, read-write) - √ Data width conversion (32b ↔ 64b ↔ 128b ↔ 256b ↔ 512b) - √ Configurable M×N topology (1-32 masters, 1-256 slaves) - √ Fair round-robin arbitration per slave

22.2.2 Architecture Philosophy

Target: Intelligent width-aware routing, not fixed-width crossbar

OLD Approach (What We're Moving Away From):

```
Master (64b) → Upsize(64→256) → Fixed 256b Crossbar → Downsize(256→64)
→ Slave (64b)
```

- Two conversions for same-width connections (wasteful!)
- Artificial bandwidth bottleneck
- Unnecessary logic and latency

NEW Approach (Target Architecture):

```
Master (64b) → Direct Connection → Slave (64b)    [0 conversions!]
Master (512b) → Conv(512→64) → Slave (64b)         [1 conversion]
```

- Per-master paths to each unique slave width it connects to
- Direct paths where widths match (zero overhead)
- Converters only where actually needed
- Router selects appropriate path based on address decode

Result: Minimal conversion logic, maximum performance, pragmatic simplicity.

22.3 ⚠️ CRITICAL: RTL Regeneration Requirements

ALL generated RTL files MUST be deleted and regenerated together whenever ANY generator code changes.

Why This Matters: - Generated RTL files may have interdependencies (bridges, wrappers, integrators) - Generator code changes can create version mismatches between files - Partial regeneration creates subtle incompatibilities that cause test failures - Even "small innocuous" generator changes can have cascading effects

The Rule:

```
# ✗ WRONG - Partial regeneration
./bridge_generator.py --masters 5 --slaves 3 --output ../rtl/
# Only regenerates bridge_axi4_flat_5x3.sv
# Other files (wrappers, integrators) now mismatched!

# ✓ CORRECT - Full regeneration
rm ../rtl/bridge_*.sv                  # Delete ALL generated
bridges
rm ../rtl/bridge_wrapper_*.sv          # Delete ALL generated
wrappers
./regenerate_all_bridges.sh            # Regenerate everything
together
```

Generator Files That Trigger Full Regeneration: - `bridge_generator.py` - Main bridge generator - `bridge_csv_generator.py` - CSV-based generator - `bridge_address_arbiter.py` - Address decode logic - `bridge_channel_router.py` - Channel routing logic - `bridge_response_router.py` - Response routing logic - `bridge_amba_integrator.py` - AMBA component integration - `bridge_wrapper_generator.py` - Wrapper generation - **Any** Python file in `projects/components/bridge/bin/`

Symptoms of Version Mismatch: - Tests that previously passed now fail - Simulation errors about missing signals - Mismatched port widths or counts - Address decode routing to wrong slaves

Think of Generated RTL Like Compiled Code: When you update a compiler, you don't selectively recompile - you rebuild everything. When you update a generator, you don't selectively regenerate - you regenerate everything.

22.4 1. Product Overview

22.4.1 1.1 Purpose

Bridge provides automated generation of AXI4 crossbar infrastructure with: - **Python code generation** - Parameterized RTL generation (similar to APB/Delta) - **Performance modeling** - Analytical + simulation validation - **Flat topology** - Full M×N interconnect matrix - **ID-based routing** - Out-of-order transaction support - **Burst optimization** - Pipelined burst transfers

22.4.2 1.2 Target Audience

Primary Users: - RTL designers building SoC interconnect - System architects evaluating interconnect topologies - Verification engineers needing crossbar testbenches - Students learning AXI4 protocol and interconnects

Educational Focus: - Demonstrates AXI4 protocol complexity - Shows arbitration strategies for memory-mapped busses - Teaches ID-based transaction tracking - Illustrates burst optimization techniques

22.4.3 1.3 Success Criteria

Functional: - [x] Generates working AXI4 crossbar RTL (bridge_generator.py) - [x] Generates CSV-configured bridges (bridge_csv_generator.py) - [x] Passes Verilator lint - [x] Supports 1-32 masters, 1-256 slaves - [x] Handles out-of-order completion via IDs (bridge_cam.sv) - [x] Supports burst lengths 1-256 beats - [x] Channel-specific masters (wr/rd/rw) for resource optimization (Phase 2) - [ ] APB converter integration (Phase 3 pending)

Performance: - [x] Latency ≤ 3 cycles for single-beat transactions - [x] Throughput: All M×N paths can transfer concurrently - [x] Performance models implemented (bridge_model.py - V1 Flat) - [ ] Fmax ≥ 300 MHz on UltraScale+ FPGAs (pending synthesis validation)

Quality: - [x] Complete specifications (PRD) before code - [x] Performance models validate requirements (bridge_model.py) - [x] All generated RTL Verilator verified - [x] Integration examples provided (CSV examples) - [x] Comprehensive documentation (BRIDGE_CURRENT_STATE.md, BRIDGE_ARCHITECTURE_DIAGRAMS.md)

22.4.4 1.4 Implementation Status and Phases

Unified Generator (bridge_generator.py):

The bridge generator now supports both TOML/CSV configuration and legacy array-indexed modes:

Configuration Modes: 1. **TOML/CSV Mode (Preferred)** - TOML port configuration (`bridge_name.toml`) - CSV connectivity matrix (`bridge_name_connectivity.csv`) - Custom port prefixes (`rapids_m_axi_`, `cpu_m_axi_`, etc.) - Interface wrapper integration (timing isolation) - Mixed protocols (AXI4 + APB + AXI4-Lite slaves) - Channel-specific masters (wr/rd/rw) - Status: √ Phase 2 complete, Phase 3 pending

2. **Legacy CSV Mode (Backwards Compatible)**
   - Separate `ports.csv` and `connectivity.csv` files
   - Migration path to TOML format
   - See `test_configs/README.md` for conversion guide

Phase Status:

Phase 1: CSV Configuration √ COMPLETE - CSV parser (ports.csv, connectivity.csv) - Port generation with custom prefixes - Converter identification logic - Basic crossbar instantiation

Phase 2: Channel-Specific Masters √ COMPLETE (2025-10-26) - Added `channels` field to PortSpec (rw/wr/rd) - Conditional port generation based on channels - **Resource Optimization:** - wr (write-only): AW, W, B channels only → 39% port reduction - rd (read-only): AR, R channels only → 61% port reduction - rw (full): All 5 channels - Width converter awareness (only generate needed converters) - Example: 4-master bridge saves 35% signals with optimized channels

Phase 3: APB Converter Integration ⏳ PENDING - AXI2APB converter module (create or integrate existing) - APB signal packing/unpacking - APB converter instantiation in generated bridges - Width converter + APB converter chaining - End-to-end testing with APB slaves - Status: Placeholders in generated code with detailed TODO comments

Additional Resources: - `bridge_model.py` - Performance modeling (V1 Flat implemented) - `bridge_cam.sv` - Transaction ID tracking for OOO support - See `BRIDGE_CURRENT_STATE.md` for detailed review - See `docs/BRIDGE_ARCHITECTURE_DIAGRAMS.md` for visual architecture

---

22.5 2. Architecture Overview

22.5.1 2.1 AXI4 vs AXIS vs APB

Feature	APB	AXI-Stream (Delta)	AXI4 (Bridge)
Protocol Type	Simple register bus	Streaming data	Memory-mapped burst
Channels	1 (address + data)	1 (data stream)	5 (AW, W, B, AR, R)
Request Generation	Address range decode	TDEST decode	Address range decode
Arbitration	Per-slave, per-cycle	Per-slave, packet-locked	Per-slave, per-address-phase
Out-of-Order	No (sequential)	No (streaming order)	Yes (ID-based)
Burst Support	No	Packet (via TLAST)	Yes (AWLEN/ARLEN)
Complexity	Low	Medium	**High**
Latency	1-2 cycles	2 cycles	2-3 cycles
Use Case	Control registers	Data streaming	Memory-mapped I/O

Bridge Complexity Sources: 1. **5 independent channels** requiring separate arbitration 2. **ID-based routing** for out-of-order completion 3. **Burst handling** with interleaving constraints 4.

Write response tracking (match AW with B channel) 5. **Address decode** + **ID muxing** for response routing

22.5.2 2.2 Block Diagram

```
┌──────────────────────────────────────────────────────────────────────
─┐
│                          BRIDGE AXI4 CROSSBAR
│
└──────────────────────────────────────────────────────────────────────
─┘

Masters (M)                                                     Slaves
(S)
┌─────────┐
┌─────────┐
│ Master 0│ ─AW──┐                                      ┌AW──│ Slave
0 │
│  (CPU)  │ ─W───┤                                      ├W───│ (DDR)
│         │ ─B───┤                                      ├B───│
│         │ ─AR──┤                                      ├AR──│
│         │ ─R───┤                                      ├R───│
└─────────┘
└─────────┘       │                                      │
                  │                                      │
┌─────────┐       │   ┌───────────────────────────┐     │
┌─────────┐
│ Master 1│ ──────┼───│  Request Generation       │─────┼─────│ Slave 1
│  (GPU)  │       │   │  (Address Range Decode)   │     │     │ (SRAM)
└─────────┘       │   └───────────────────────────┘     │
└─────────┘
                  │               │                      │
┌─────────┐       │   ┌───────────────────────────┐     │
┌─────────┐
│ Master 2│ ──────┼───│  Arbitration Matrix       │─────┼─────│ Slave 2
│  (DMA)  │       │   │  (5 separate arbiters     │     │     │ (PCIE)
└─────────┘       │   │   per slave: AW,W,B,AR,R)│     │
└─────────┘       │   └───────────────────────────┘     │
                  │               │                      │
┌─────────┐
┌─────────┐
```

```
│ Master 3│ ─────┴─────│  Data/Addr Multiplexing  │──────┴──────│ Slave 3
│                      │                          │             │
│ (Accel) │            │  (ID-based routing)      │             │
(Periph)│
└─────────┘            └──────────────────────────┘
└─────────┘
                                    │
                       ┌──────────────────────────┐
                       │  Transaction Tracking    │
                       │  (ID tables for OoO)     │
                       └──────────────────────────┘
```

22.5.3 2.3 Key Components

1. Request Generation - Address range decode for each slave - Generates M×S request matrix per channel (AW, AR) - Similar to APB but more complex (2 address channels)

2. Per-Slave Arbitration - **5 separate arbiters per slave:** - AW channel arbiter (write address) - W channel arbiter (write data - locked to AW grant) - B channel arbiter (write response - routed by ID) - AR channel arbiter (read address) - R channel arbiter (read data - routed by ID) - Round-robin with burst locking - Separate read/write paths (no head-of-line blocking)

3. Data Multiplexing - Mux master signals to selected slave - ID-based response routing (B, R channels) - Burst tracking (hold grant until xlast)

4. Transaction Tracking - ID tables per slave for out-of-order support - Track {Master ID, Transaction ID} → Master mapping - Required for routing B/R channels back to correct master

5. Optional Performance Counters - Transaction counts per master/slave - Arbitration conflict counts - Latency histograms

---

22.6 3. Functional Requirements

22.6.1 3.1 AXI4 Protocol Compliance

FR-1: Full AXI4 Protocol Support - Support all 5 AXI4 channels: AW, W, B, AR, R - Comply with AMBA AXI4 specification (ARM IHI 0022) - Support burst lengths: 1-256 beats (AWLEN/ARLEN = 0-255) - Support burst types: INCR, WRAP, FIXED - Support burst sizes: 1-128 bytes (AWSIZE/ARSIZE = 0-7)

FR-2: Out-of-Order Transaction Support - Route responses via ID matching - Maintain transaction ID integrity (AWID → BID, ARID → RID) - Support configurable ID width (1-16 bits) - Track up to 2^ID_WIDTH outstanding transactions per slave

FR-3: Atomic Operations - Support exclusive access (AWLOCK/ARLOCK) - Track exclusive monitor per slave - Generate BRESP/RRESP errors for failed exclusives

22.6.2 3.2 Address Decoding

FR-4: Configurable Address Map - Support M masters × S slaves - Each slave has base address and size - Non-overlapping address ranges (verified at generation) - Default slave for unmapped addresses (optional)

FR-5: Address Range Configuration

```
# Example address map
address_map = {
    0: {'base': 0x00000000, 'size': 0x40000000, 'name': 'DDR'},      # 1GB
    1: {'base': 0x40000000, 'size': 0x00100000, 'name': 'SRAM'},     # 1MB
    2: {'base': 0x50000000, 'size': 0x01000000, 'name': 'PCIE'},     # 16MB
    3: {'base': 0x60000000, 'size': 0x00010000, 'name': 'Peripherals'} # 64KB
}
```

22.6.3 3.3 Arbitration Strategy

FR-6: Round-Robin Arbitration - Fair bandwidth allocation (no starvation) - Separate arbiters for AW and AR channels - Burst locking: Grant held until xlast (WLAST/RLAST) - Configurable arbitration policy (round-robin default)

FR-7: Read/Write Independence - Separate read and write paths - No head-of-line blocking between read/write - Concurrent read and write to same slave (if slave supports)

22.6.4 3.4 Burst Handling

FR-8: Burst Optimization - Pipelined burst transfers (overlap address and data) - No artificial burst splitting - Full AXI4 burst protocol support

FR-9: Interleaving Constraints - W channel locked to AW grant master - R channel routed by transaction ID - Support ID-based interleaving (slave-dependent)

22.7 4. Non-Functional Requirements

22.7.1 4.1 Performance

NFR-1: Latency - **Single-beat read:** ≤ 3 cycles (address decode + arbitration + mux) - **Single-beat write:** ≤ 3 cycles (address decode + arbitration + mux) - **Burst transfer:** No additional latency per beat (pipelined)

NFR-2: Throughput - **Concurrent transfers:** All M×S paths can transfer simultaneously - **Burst efficiency:** Line-rate data transfer after address phase - **No artificial stalls:** Crossbar adds no wait states beyond arbitration

NFR-3: Fmax - **Target:** 300-400 MHz on Xilinx UltraScale+ - **Registered outputs:** All slave outputs registered for timing closure - **Pipelineable:** Optional pipeline stages for >400 MHz

22.7.2 4.2 Resource Usage (Estimated)

M = 4 masters, S = 4 slaves, DATA_WIDTH = 512, ADDR_WIDTH = 64, ID_WIDTH = 4:

Resource	Flat Crossbar	Notes
LUTs	~2,500	Address decode + arbiters + mux
FFs	~3,000	Registered outputs + ID tables
BRAM	0	Distributed RAM for small ID tables
DSP	0	No arithmetic operations

Scaling: ~150 LUTs per M×S connection

22.7.3 4.3 Quality Requirements

NFR-4: Code Generation Quality - Lint-clean SystemVerilog (Verilator) - Synthesizable (Vivado, Yosys, Design Compiler) - No vendor-specific primitives (technology-agnostic) - Clear structure (commented, readable)

NFR-5: Verification - CocoTB testbench framework - Transaction-level verification - Out-of-order test scenarios - Burst interleaving tests - >95% functional coverage

---

22.8 5. Interface Specifications

22.8.1 5.1 AXI4 Master Interfaces (M × 5 channels)

Write Address Channel (AW):

```
input  logic [ADDR_WIDTH-1:0]   s_axi_awaddr  [NUM_MASTERS];
input  logic [ID_WIDTH-1:0]     s_axi_awid    [NUM_MASTERS];
input  logic [7:0]              s_axi_awlen   [NUM_MASTERS];  // Burst
length-1
input  logic [2:0]              s_axi_awsize  [NUM_MASTERS];  // Burst
```

```
size
input  logic [1:0]                s_axi_awburst [NUM_MASTERS];  //
INCR/WRAP/FIXED
input  logic                      s_axi_awlock  [NUM_MASTERS];  //
Exclusive access
input  logic [3:0]                s_axi_awcache [NUM_MASTERS];  // Cache
attributes
input  logic [2:0]                s_axi_awprot  [NUM_MASTERS];  //
Protection type
input  logic                      s_axi_awvalid [NUM_MASTERS];
output logic                      s_axi_awready [NUM_MASTERS];
```

Write Data Channel (W):

```
input  logic [DATA_WIDTH-1:0]   s_axi_wdata  [NUM_MASTERS];
input  logic [DATA_WIDTH/8-1:0] s_axi_wstrb  [NUM_MASTERS];  // Byte
strobes
input  logic                    s_axi_wlast  [NUM_MASTERS];  // Last
beat
input  logic                    s_axi_wvalid [NUM_MASTERS];
output logic                    s_axi_wready [NUM_MASTERS];
```

Write Response Channel (B):

```
output logic [ID_WIDTH-1:0]     s_axi_bid    [NUM_MASTERS];
output logic [1:0]              s_axi_bresp  [NUM_MASTERS];  //
OKAY/EXOKAY/SLVERR/DECERR
output logic                    s_axi_bvalid [NUM_MASTERS];
input  logic                    s_axi_bready [NUM_MASTERS];
```

Read Address Channel (AR):

```
input  logic [ADDR_WIDTH-1:0]   s_axi_araddr  [NUM_MASTERS];
input  logic [ID_WIDTH-1:0]     s_axi_arid    [NUM_MASTERS];
input  logic [7:0]              s_axi_arlen   [NUM_MASTERS];
input  logic [2:0]              s_axi_arsize  [NUM_MASTERS];
input  logic [1:0]              s_axi_arburst [NUM_MASTERS];
input  logic                    s_axi_arlock  [NUM_MASTERS];
input  logic [3:0]              s_axi_arcache [NUM_MASTERS];
input  logic [2:0]              s_axi_arprot  [NUM_MASTERS];
input  logic                    s_axi_arvalid [NUM_MASTERS];
output logic                    s_axi_arready [NUM_MASTERS];
```

Read Data Channel (R):

```
output logic [DATA_WIDTH-1:0]   s_axi_rdata  [NUM_MASTERS];
output logic [ID_WIDTH-1:0]     s_axi_rid    [NUM_MASTERS];
output logic [1:0]              s_axi_rresp  [NUM_MASTERS];
output logic                    s_axi_rlast  [NUM_MASTERS];
output logic                        s_axi_rvalid [NUM_MASTERS];
input  logic                        s_axi_rready [NUM_MASTERS];
```

22.8.2 5.2 AXI4 Slave Interfaces (S × 5 channels)

Mirror of master interfaces, with M → S direction reversed.

22.8.3 5.3 Configuration Parameters

```
parameter int NUM_MASTERS   = 4;          // Number of master
interfaces
parameter int NUM_SLAVES    = 4;          // Number of slave
interfaces
parameter int DATA_WIDTH    = 512;        // Data bus width (bits)
parameter int ADDR_WIDTH    = 64;         // Address bus width (bits)
parameter int ID_WIDTH      = 4;          // Transaction ID width
(bits)
parameter int MAX_BURST_LEN = 256;        // Maximum burst length
(beats)
parameter bit PIPELINE_OUTPUTS = 1;       // Register slave outputs
parameter bit ENABLE_COUNTERS  = 1;       // Performance counters
```

22.9 6. Performance Modeling

22.9.1 6.1 Analytical Model

Latency Components (Flat Crossbar):

```
Single-Beat Read Latency:
  1. Address Decode:      0 cycles (combinatorial)
  2. Arbitration:         1 cycle  (AR arbiter decision)
  3. Address Mux:         0 cycles (combinatorial)
  4. Slave Access:        (slave-dependent, not crossbar)
  5. Response Mux:        1 cycle  (R channel ID lookup + mux)
  6. Output Register:     1 cycle  (optional, for timing closure)

  Total (no pipeline):    2 cycles
  Total (pipelined):      3 cycles
```

Burst Transfer Throughput:

```
After address phase completes, data transfer is line-rate:
  - W channel: 1 beat/cycle (locked to AW grant)
  - R channel: 1 beat/cycle (ID-routed from slave)


Example: 256-beat burst
  Address phase:  2-3 cycles (one-time)
  Data phase:     256 cycles (line-rate)
  Total:          258-259 cycles
  Efficiency:     98.8%
```

Concurrent Throughput:

```
Maximum concurrent transfers (4×4 crossbar):
  - Read:  4 concurrent (one per master-slave pair)
  - Write: 4 concurrent (one per master-slave pair)
  - Total: 8 concurrent read+write (separate paths)

Throughput @ 100 MHz, 512-bit data:
  Per path:   100 MHz × 512 bits = 51.2 Gbps
  Total:      8 paths × 51.2 Gbps = 409.6 Gbps theoretical
```

22.9.2 6.2 Resource Scaling

LUT Usage Formula (empirical):

```
LUTs ≈ 500 (base) + 150 × M × S + 20 × ID_TABLE_DEPTH
```

Example (4×4 crossbar, ID_WIDTH=4, ID_TABLE_DEPTH=16 per slave):

```
LUTs ≈ 500 + 150 × 16 + 20 × 16 × 4
     ≈ 500 + 2,400 + 1,280
     ≈ 4,180 LUTs
```

22.9.3 6.3 Comparison with Other Crossbars

Crossbar Type	Latency	Throughput	Complexity	Use Case
APB	1-2 cycles	Low (serialized)	Low	Control registers
AXI-Stream (Delta)	2 cycles	High (streaming)	Medium	Data streaming
AXI4 (Bridge)	2-3 cycles	High (burst)	**High**	Memory-mapped I/O
AXI4 + Slices	4-6 cycles	High (burst)	Very High	>400 MHz designs

Bridge Sweet Spot: High-performance memory-mapped interconnects where out-of-order and burst efficiency are critical.

22.10 7. Generator Architecture

22.10.1 7.1 Python Generator Structure

```python
class BridgeGenerator:
    """AXI4 crossbar RTL generator"""

    def __init__(self, config):
        self.num_masters = config.num_masters
        self.num_slaves = config.num_slaves
        self.data_width = config.data_width
        self.addr_width = config.addr_width
        self.id_width = config.id_width
        self.address_map = config.address_map

    def generate_address_decode(self) -> str:
        """Generate address range decode logic"""
        # For each master × slave, check if address in range
        # More complex than AXIS (uses address), simpler than full
decode

    def generate_aw_arbiter(self, slave_idx) -> str:
        """Generate write address channel arbiter for one slave"""
        # Round-robin arbiter
        # Grants locked until corresponding B response completes

    def generate_ar_arbiter(self, slave_idx) -> str:
        """Generate read address channel arbiter for one slave"""
        # Round-robin arbiter
        # Grants locked until corresponding R response completes
(RLAST)

    def generate_w_channel_mux(self, slave_idx) -> str:
        """Generate write data channel multiplexer"""
        # W channel follows AW grant (locked until WLAST)

    def generate_b_channel_demux(self, slave_idx) -> str:
        """Generate write response channel demultiplexer"""
        # Route by ID: lookup master from {slave_idx, BID}

    def generate_r_channel_demux(self, slave_idx) -> str:
        """Generate read data channel demultiplexer"""
        # Route by ID: lookup master from {slave_idx, RID}

    def generate_id_table(self, slave_idx) -> str:
        """Generate transaction ID tracking table"""
        # Maps {slave, transaction_id} → master_id
        # Indexed on AW/AR grant, looked up on B/R response

    def generate_crossbar(self) -> str:
        """Generate complete crossbar module"""
        # Instantiate all arbiters, muxes, demuxes, ID tables
```

22.10.2 7.2 Address Map Configuration

Python Configuration:

```
bridge_config = {
    'num_masters': 4,
    'num_slaves': 4,
    'data_width': 512,
    'addr_width': 64,
    'id_width': 4,
    'address_map': {
        0: {'base': 0x00000000, 'size': 0x40000000, 'name': 'DDR'},
        1: {'base': 0x40000000, 'size': 0x00100000, 'name': 'SRAM'},
        2: {'base': 0x50000000, 'size': 0x01000000, 'name': 'PCIE'},
        3: {'base': 0x60000000, 'size': 0x00010000, 'name':
'Peripherals'}
    },
    'pipeline_outputs': True,
    'enable_counters': True
}
```

Generated Address Decode:

```
// Address decode for each master
always_comb begin
    for (int s = 0; s < NUM_SLAVES; s++)
        aw_request_matrix[s] = '0;

    for (int m = 0; m < NUM_MASTERS; m++) begin
        if (s_axi_awvalid[m]) begin
            // Slave 0: DDR (0x00000000 - 0x3FFFFFFF)
            if (s_axi_awaddr[m] >= 64'h00000000 &&
                s_axi_awaddr[m] < 64'h40000000)
                aw_request_matrix[0][m] = 1'b1;

            // Slave 1: SRAM (0x40000000 - 0x400FFFFF)
            if (s_axi_awaddr[m] >= 64'h40000000 &&
                s_axi_awaddr[m] < 64'h40100000)
                aw_request_matrix[1][m] = 1'b1;

            // ... slaves 2-3 ...
            end
        end
    end
```

22.11 8. Comparison with APB and Delta

22.11.1 8.1 Code Reuse from APB Generator

Similar Components (~70% reuse): - Address range decode logic (same pattern, different signals) - Round-robin arbiter (same algorithm) - Data multiplexing pattern (same approach) - Backpressure handling (similar to PREADY)

New Components for Bridge: - **5× the arbiters** (AW, W, B, AR, R instead of single channel) - **ID-based routing** (B and R channel demuxing) - **Transaction tracking** (ID tables for out-of-order) - **Burst handling** (grant locking until xlast)

Migration Effort from APB: - ~120 minutes (vs ~75 min for AXIS, due to higher complexity) - Most time: ID table logic and response demuxing

22.11.2 8.2 Code Reuse from Delta Generator

Similar Components (~60% reuse): - Python generation framework - Command-line interface - Performance modeling structure - Arbitration pattern (round-robin with locking)

Key Differences: - **5 channels** vs 1 channel (Delta only has TDATA/TVALID/TREADY/TLAST) - **ID-based routing** vs TDEST-based routing - **Address decode** vs TDEST decode (Bridge more complex) - **Transaction tracking** vs packet atomicity (different mechanisms)

22.11.3 8.3 Complexity Comparison

Metric	APB Crossbar	Delta (AXIS)	Bridge (AXI4)
Channels to arbitrate	1	1	5 ★
Request generation	Address ranges	TDEST decode	Address ranges
Response routing	Grant-based	Grant-based	ID-based ★
Burst support	No	Packet (TLAST)	Yes (AWLEN/ARLEN) ★
Out-of-	No	No	Yes (ID tables) ★

Metric	APB Crossbar	Delta (AXIS)	Bridge (AXI4)
order			
Transaction tracking	No	No	Yes ★
Lines of Python	~500	~697	**~900** (est.)
Lines of generated SV	~200 (4×4)	~250 (4×4)	**~400** (4×4) (est.)

★ = Additional complexity in Bridge

22.12 9. Use Cases

22.12.1 9.1 Multi-Core Processor Interconnect

Scenario: 4 CPU cores + GPU accessing DDR + SRAM + Peripherals

```
Configuration:
  Masters:  5 (4 CPUs, 1 GPU)
  Slaves:   3 (DDR, SRAM, Peripherals)
  Data:     512-bit (cache line width)
  Address:  64-bit (large memory space)
  ID:       4-bit (up to 16 outstanding per master)
```

Benefits: - Concurrent access to all slaves - Out-of-order completion for high-performance CPUs - Burst transfers for cache line fills - Separate read/write paths (no head-of-line blocking)

22.12.2 9.2 DMA + Accelerator System

Scenario: DMA engine + 4 accelerators accessing shared memory

```
Configuration:
  Masters:  5 (1 DMA, 4 accelerators)
  Slaves:   2 (DDR, Control registers)
  Data:     512-bit (high-bandwidth DMA)
  Address:  32-bit (limited address space)
  ID:       2-bit (simple ID space)
```

Benefits: - High-bandwidth memory access for DMA - Fair arbitration prevents accelerator starvation - Control register access doesn't block data transfers

22.12.3 9.3 FPGA System Integration

Scenario: MicroBlaze CPU + custom accelerators + memory controllers

```
Configuration:
  Masters:  8 (1 CPU, 7 accelerators)
  Slaves:   4 (DDR, BRAM, AXI GPIO, AXI DMA)
  Data:     128-bit (AXI4 standard width)
  Address:  32-bit (standard FPGA address space)
  ID:       4-bit (moderate outstanding transactions)
```

Benefits: - Standard AXI4 interfaces (Xilinx IP compatibility) - Scalable to many masters/slaves - Performance counters for profiling

---

22.13 10. Testing Strategy

22.13.1 10.1 FUB (Functional Unit Block) Tests

Address Decode Tests: - Verify all address ranges correctly decoded - Test boundary conditions (base, base+size-1) - Test unmapped addresses (error response)

Arbiter Tests: - Round-robin fairness (all masters get turns) - Burst locking (grant held until xlast) - Starvation prevention

ID Table Tests: - Correct ID → master mapping - Out-of-order transaction handling - Table full condition

Mux/Demux Tests: - Data integrity through crossbar - Response routing to correct master - Concurrent transfers don't interfere

22.13.2 10.2 Integration Tests

Single-Master, Single-Slave: - Basic read/write transactions - Burst transfers (various lengths) - Out-of-order completions

Multi-Master, Single-Slave: - Arbitration correctness - Fairness verification - Burst interleaving (if supported)

Multi-Master, Multi-Slave: - Concurrent transfers - No crosstalk between paths - Full M×S matrix coverage

Stress Tests: - All masters active simultaneously - Maximum burst lengths - Full ID space utilization - Back-to-back bursts

22.13.3 10.3 Performance Validation

Latency Measurement: - Single-beat read: measure actual vs analytical - Single-beat write: measure actual vs analytical - Compare with specification (≤3 cycles)

Throughput Measurement: - Burst transfer efficiency (should be ~98%) - Concurrent path throughput (should be line-rate × M×S) - Compare with theoretical maximum

Resource Validation: - Synthesize for Xilinx UltraScale+ - Compare LUT/FF usage with estimates - Verify Fmax ≥ 300 MHz

---

22.14 11. Documentation Plan

22.14.1 11.1 Specifications (Before Code)

- ☒ **PRD.md** - This document (complete requirements)
- ☐ **README.md** - User guide with quick start
- ☐ **BRIDGE_VS_APB_GENERATOR.md** - Migration guide from APB
- ☐ **BRIDGE_VS_DELTA_GENERATOR.md** - Comparison with Delta
- ☐ **AXI4_PROTOCOL_GUIDE.md** - AXI4 primer for students

22.14.2 11.2 Performance Analysis (Before Implementation)

- ☐ **bin/bridge_performance_model.py** - Analytical model
  - Latency calculations (single-beat, burst)
  - Throughput estimates (concurrent paths)
  - Resource estimates (LUT/FF scaling)
- ☐ **bin/bridge_simulator.py** - Discrete event simulation (optional)
  - Cycle-accurate modeling
  - Traffic pattern support
  - Validation against RTL

22.14.3 11.3 Code Generation

- ☐ **bin/bridge_generator.py** - Main RTL generator
  - Command-line interface
  - Address map configuration
  - Generated RTL output

22.14.4 11.4 Verification

- ☐ **dv/tests/** - CocoTB testbenches
  - FUB tests (individual blocks)
  - Integration tests (multi-block)
  - Stress tests (corner cases)

22.15 12. Success Metrics

22.15.1 12.1 Functional Completeness

- ☐ Generates working AXI4 crossbar RTL
- ☐ Passes all CocoTB tests
- ☐ Verilator lint clean
- ☐ Xilinx Vivado synthesis clean

22.15.2 12.2 Performance Targets

- ☐ Latency ≤ 3 cycles (measured in simulation)
- ☐ Throughput = line-rate × M×S paths (measured)
- ☐ Fmax ≥ 300 MHz (post-synthesis)

22.15.3 12.3 Educational Value

- ☐ Complete specifications demonstrating rigor
- ☐ Performance models validate requirements
- ☐ Clear code structure (readable generated RTL)
- ☐ Integration examples provided

22.15.4 12.4 Reusability

- ☐ ~70% code reuse from APB generator (measured by LOC)
- ☐ Similar patterns to Delta generator
- ☐ Can be extended (weighted arbitration, QoS, etc.)

22.16 12. Attribution and Contribution Guidelines

22.16.1 12.1 Git Commit Attribution

When creating git commits for Bridge documentation or implementation:

Use:

```
Documentation and implementation support by Claude.
```

Do NOT use:

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

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

22.17 12.2 PDF Generation Location

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

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

Quick Command: Use the provided shell script:

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

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

📖 **See:** `bin/md_to_docx.py` for complete implementation details

22.18 13. Future Enhancements

22.18.1 13.1 Short-Term (Post-Initial Release)

- ☐ **Optional pipeline stages** - For Fmax >400 MHz
- ☐ **Weighted arbitration** - QoS support
- ☐ **Default slave** - Unmapped address handling
- ☐ **Exclusive monitor** - Full atomic operation support

22.18.2 13.2 Long-Term

- ☐ **Tree topology** - Hierarchical crossbar (like Delta)
- ☐ **AXI4-Lite variant** - Simplified for control registers
- ☐ **ACE support** - Coherent cache interconnect
- ☐ **GUI configurator** - Visual address map setup

22.19 14. Risk Assessment

Risk	Probability	Impact	Mitigation
ID table complexity	Medium	High	Start with small ID_WIDTH (2-4), test thoroughly
Out-of-order corner cases	High	High	Extensive CocoTB tests with random delays
Fmax below target	Low	Medium	Optional pipeline stages for timing closure
Resource usage exceeds	Low	Low	Empirical formulas guide expectations
Burst interleaving bugs	Medium	High	Separate test suite for burst scenarios

22.20 15. Project Timeline (Estimated)

Week 1: Specifications and Models - [x] Day 1-2: PRD.md (complete) - [ ] Day 3-4: Performance modeling (analytical) - [ ] Day 5-7: README.md, migration guides

Week 2-3: Core Implementation - [ ] Day 1-3: Address decode + arbiter generation - [ ] Day 4-5: Data mux/demux generation - [ ] Day 6-8: ID table generation - [ ] Day 9-10: Integration and testing

Week 4: Verification and Examples - [ ] Day 1-5: CocoTB testbenches - [ ] Day 6-7: Integration examples

22.21 16. References

AXI4 Specifications: - ARM IHI 0022 - AMBA AXI and ACE Protocol Specification - Xilinx UG1037 - Vivado AXI Reference Guide

Related Projects: - **APB Crossbar** - Simple register bus crossbar (existing) - **Delta (AXIS Crossbar)** - Streaming data crossbar (projects/components/delta/) - **RAPIDS** - DMA engine with AXI4 masters (rtl/miop/)

Tools: - Verilator - RTL linting and simulation - CocoTB - Python-based verification - Xilinx Vivado - FPGA synthesis

---

22.22 17. Glossary

- **AXI4:** Advanced eXtensible Interface version 4 (AMBA standard)
- **Burst:** Multi-beat transaction (AWLEN/ARLEN > 0)
- **Crossbar:** Full M×N interconnect matrix
- **ID:** Transaction identifier for out-of-order support
- **Out-of-order:** Responses can return in different order than requests
- **xlast:** WLAST (write) or RLAST (read) - last beat indicator

---

Version: 1.0 **Status:** √ Specification Complete - Ready for Implementation **Next Steps:** Create performance models, then implement generator

Project Bridge - Connecting masters and slaves across the divide 🌉

23 Claude Code Guide: Bridge Subsystem

Version: 2.1 **Last Updated:** 2025-11-03 **Purpose:** AI-specific guidance for working with Bridge subsystem

---

23.1 🚨 CRITICAL: Read Architecture Document First

Before making ANY changes to bridge generator or understanding signal flow:

📖 **READ:** `projects/components/bridge/docs/BRIDGE_ARCHITECTURE.md`

This document contains the **definitive bridge architecture** including: - Correct signal flow (wrappers → decoder → converters → crossbar → slaves) - Component purposes and placement - Common misconceptions that previous agents made - Why there's NO fixed crossbar width

If you skip this document, you WILL make incorrect assumptions.

23.2 Quick Context

What: Bridge - Two complementary AXI4 Crossbar Generators (framework-based and CSV-based) **Status:** 🟢 Phase 2 Complete - CSV generator with channel-specific masters (wr/rd/rw) **Your Role:** Help users configure CSV files, generate bridges, understand architecture, and create tests

📖 **Complete Documentation (Read in This Order):** 1. `projects/components/bridge/docs/BRIDGE_ARCHITECTURE.md` ← **START HERE** (architecture reference) 2. `projects/components/bridge/PRD.md` ← Product requirements 3. `projects/components/bridge/BRIDGE_CURRENT_STATE.md` ← Current implementation review 4. `projects/components/bridge/docs/BRIDGE_ARCHITECTURE_DIAGRAMS.md` ← Visual architecture diagrams 5. `projects/components/bridge/CSV_BRIDGE_STATUS.md` ← CSV generator status (Phase 1 & 2) 6. `projects/components/bridge/docs/bridge_spec/bridge_index.md` ← Detailed specification

---

23.3 Target Architecture: Intelligent Width-Aware Routing

Core Principle: Direct connections where possible, converters only where needed, no fixed crossbar width.

23.3.1 Efficient Multi-Width Design

```
Master_A (64b)
  ├─ Direct → Slave_0 (64b)          [0 conversions, minimal latency]
  ├─ Conv(64→128) → Slave_1 (128b)   [1 conversion]
  └─ Conv(64→512) → Slave_2 (512b)   [1 conversion]

Master_B (512b)
  ├─ Conv(512→64) → Slave_0 (64b)    [1 conversion]
  ├─ Conv(512→128) → Slave_1 (128b)  [1 conversion]
  └─ Direct → Slave_2 (512b)         [0 conversions, full bandwidth]
```

Router Logic: Address decoder determines target slave, selects correctly-sized path for that master-slave pair.

23.3.2 Why This Architecture

✗ Naive Fixed-Width Approach (Don't Do This):

```
Master (64b) → Upsize(64→256) → Fixed 256b Crossbar → Downsize(256→64)
→ Slave (64b)
```

- TWO conversions for same-width connections (wasteful!)
- Reduced bandwidth on narrow paths

- Unnecessary logic and area
- Higher latency

√ **Intelligent Routing (Target):**

```
Master (64b) → Router → Direct Connection → Slave (64b)
```

- ZERO conversions for matching widths
- Full native bandwidth
- Minimal logic
- Lowest latency

23.3.3 Per-Master Output Paths

Each master has N output paths (one per unique slave width it connects to):

```
// Master_A connects to slaves at 64b, 128b, 512b
// Generate 3 output paths:

logic [63:0]  master_a_64b_wdata;   // For 64b slaves (direct)
logic [127:0] master_a_128b_wdata;  // For 128b slaves (via converter)
logic [511:0] master_a_512b_wdata;  // For 512b slaves (via converter)

// Router selects based on address decode:
always_comb begin
    case (decoded_slave_id)
        SLAVE_0: select master_a_64b_wdata;   // Slave_0 is 64b
        SLAVE_1: select master_a_128b_wdata;  // Slave_1 is 128b
        SLAVE_2: select master_a_512b_wdata;  // Slave_2 is 512b
    endcase
end
```

23.3.4 Benefits

1. **Resource Efficient** - Only instantiate converters actually needed
2. **Maximum Performance** - Direct paths have zero conversion overhead
3. **Optimal Bandwidth** - No artificial width bottlenecks
4. **Lower Latency** - Minimal logic in critical path for matching widths
5. **Scalable** - Works for any combination of master/slave widths

23.3.5 Implementation Status

Current State: Fixed-width crossbar with master-side upsizing (Phase 1 architecture) **Target State:** Intelligent per-master multi-width routing (your vision) **Migration:** Requires generator architecture rework (see TASKS.md)

23.4 🚨 CRITICAL RULE #0: RTL Regeneration Requirements

⚠️ **READ THIS FIRST - FAILURE TO FOLLOW CAUSES TEST FAILURES** ⚠️

23.4.1 The Golden Rule

ALL generated RTL files MUST be deleted and regenerated together whenever ANY generator code changes.

23.4.2 Why This Is Non-Negotiable

Generated RTL files have interdependencies: - `bridge_axi4_flat_*.sv` may be instantiated by `bridge_ooo_with_arbiter.sv` - `bridge_wrapper_*.sv` may wrap `bridge_axi4_flat_*.sv` - Generator changes affect signal names, port widths, interface structure - **Partial regeneration creates version mismatches that cause silent failures**

23.4.3 Real Example (This Session)

```
# ✗ WHAT WENT WRONG:
1. Updated bridge_address_arbiter.py (address decode logic)
2. Regenerated ONLY bridge_axi4_flat_5x3.sv
3. Did NOT regenerate bridge_ooo_with_arbiter.sv (wrapper)
4. Result: All tests that were passing now FAIL
5. Cause: Version mismatch between wrapper and core bridge

# ✓ WHAT SHOULD HAVE BEEN DONE:
1. Updated bridge_address_arbiter.py
2. Delete ALL generated files:
   rm rtl/bridge_axi4_flat_*.sv
   rm rtl/bridge_ooo_*.sv
   rm rtl/bridge_wrapper_*.sv
3. Regenerate ALL bridges from scratch
4. Run ALL tests to verify
```

23.4.4 Generator Files That Trigger Full Regeneration

Any change to these files requires regenerating ALL bridges:

- √ `bridge_generator.py` - Core bridge generator
- √ `bridge_csv_generator.py` - CSV-based generator
- √ `bridge_address_arbiter.py` - Address decode logic
- √ `bridge_channel_router.py` - Channel routing
- √ `bridge_response_router.py` - Response routing
- √ `bridge_amba_integrator.py` - AMBA integration

- √ `bridge_wrapper_generator.py` - Wrapper generation
- √ **ANY** Python file in `projects/components/bridge/bin/`

23.4.5 The Regeneration Workflow

```
# Step 1: Make generator code changes
vim bridge_address_arbiter.py

# Step 2: Delete ALL generated RTL (be aggressive!)
cd projects/components/bridge/rtl
rm bridge_axi4_flat_*.sv
rm bridge_ooo*.sv
rm bridge_wrapper_*.sv
# Verify deletion
ls *.sv  # Should only show manually-written files like bridge_cam.sv

# Step 3: Regenerate everything
cd ../bin
./regenerate_all_bridges.sh  # If script exists
# OR manually regenerate each topology needed

# Step 4: Run ALL tests
cd ../dv/tests
pytest -v  # ALL tests, not just the one you think changed

# Step 5: Verify git diff makes sense
git diff ../rtl/  # Review all changes
```

23.4.6 Symptoms of Version Mismatch

If you see these symptoms, you probably did partial regeneration:

- ✗ Tests that previously passed now fail
- ✗ “Signal not found” errors in simulation
- ✗ Port width mismatches in instantiation
- ✗ Address decode routing to wrong slaves
- ✗ Missing debug signals (dbg_*)
- ✗ Unexpected compile errors in working code

23.4.7 Think Like a Compiler Developer

Generated RTL = **Compiled Object Files**

When you update a compiler, you don’t selectively recompile - you `make clean && make all`.

When you update a generator, you don't selectively regenerate - you **delete all and regenerate all**.

23.4.8 Exception: Hand-Written RTL

These files are **never** regenerated: - `bridge_cam.sv` - CAM module (hand-written) - Any file in `rtl/` that is NOT generated

Check file headers - generated files say "Generated by: bridge_generator.py"

23.5 ⚠️ MANDATORY: Project Organization Pattern

THIS SUBSYSTEM MUST FOLLOW THE RAPIDS/AMBA ORGANIZATIONAL PATTERN - NO EXCEPTIONS

23.5.1 Required Directory Structure

```
projects/components/bridge/
├── bin/                            # Bridge-specific tools
(generators, scripts)
│   └── bridge_generator.py        # AXI4 crossbar generator
├── docs/                           # Design documentation
├── dv/                             # Design Verification (MANDATORY
structure)
│   ├── tbclasses/                  # Testbench classes (MANDATORY - TB
classes here!)
│   │   ├── __init__.py
│   │   └── bridge_axi4_flat_tb.py  # Reusable TB class
│   ├── components/                 # Bridge-specific BFMs (if needed)
│   │   └── __init__.py
│   ├── scoreboards/                # Bridge-specific scoreboards (if
needed)
│   │   └── __init__.py
│   └── tests/                      # All test files (test runners
only)
│       ├── conftest.py             # Pytest configuration (MANDATORY)
│       ├── fub_tests/              # Functional unit block tests
│       │   └── basic/              # Basic bridge tests
│       ├── integration_tests/      # Multi-bridge integration
│       └── system_tests/           # Full system tests
├── rtl/                            # RTL source files
│   └── generated/                  # Generated bridge crossbars
├── CLAUDE.md                       # This file
├── PRD.md                          # Product requirements
└── IMPLEMENTATION_STATUS.md        # Development status
```

23.5.2 Testbench Class Location (MANDATORY)

✗ **WRONG:** Testbench class in test file

```
# projects/components/bridge/dv/tests/test_bridge_axi4_2x2.py
class BridgeAXI4FlatTB:  # ✗ WRONG LOCATION!
    """Embedded TB - NOT REUSABLE"""
```

√ **CORRECT:** Testbench class in PROJECT AREA dv/tbclasses/

```
# projects/components/bridge/dv/tbclasses/bridge_axi4_flat_tb.py
class BridgeAXI4FlatTB(TBBase):  # ✓ CORRECT LOCATION!
    """Reusable TB class - used across all bridge tests"""
```

CRITICAL: TB classes are PROJECT-SPECIFIC and MUST be in the project area (`projects/components/{name}/dv/tbclasses/`), NOT in the framework (`bin/CocoTBFramework/`).

23.5.3 Test File Pattern (MANDATORY)

Test files MUST follow this structure:

```
#
projects/components/bridge/dv/tests/fub_tests/basic/test_bridge_axi4_2
x2.py

import os
import pytest
import cocotb
from cocotb_test.simulator import run

# ✓ IMPORT testbench class from PROJECT AREA
import sys
sys.path.insert(0, os.path.join(os.path.dirname(__file__),
'../../../..'))
from projects.components.bridge.dv.tbclasses.bridge_axi4_flat_tb
import BridgeAXI4FlatTB
from CocoTBFramework.tbclasses.shared.utilities import get_paths,
create_view_cmd
from CocoTBFramework.tbclasses.shared.tbbase import TBBase

#
======================================================================
=====
# COCOTB TEST FUNCTIONS - Prefix with "cocotb_" to prevent pytest
collection
#
======================================================================
=====
@cocotb.test(timeout_time=100, timeout_unit='us')
async def cocotb_test_basic_routing(dut):
    """Test basic address routing"""
    tb = BridgeAXI4FlatTB(dut, num_masters=2, num_slaves=2)  # ✓
Import TB
    await tb.setup_clocks_and_reset()
    result = await tb.test_basic_routing()
    assert result, "Basic routing test failed"

# More cocotb test functions...

#
======================================================================
=====
# PARAMETER GENERATION - At bottom of file
#
======================================================================
=====

def generate_bridge_test_params():
    """Generate test parameters for bridge tests"""
    return [
        # (num_masters, num_slaves, data_width, addr_width, id_width)
        (2, 2, 32, 32, 4),
        (4, 4, 32, 32, 4),
    ]

bridge_params = generate_bridge_test_params()

#
======================================================================
=====
# PYTEST WRAPPER FUNCTIONS - At bottom of file
#
======================================================================
=====

@pytest.mark.bridge
@pytest.mark.routing
@pytest.mark.parametrize("num_masters, num_slaves, data_width,
addr_width, id_width", bridge_params)
def test_basic_routing(request, num_masters, num_slaves, data_width,
addr_width, id_width):
    """Pytest wrapper for basic routing test"""
    module, repo_root, tests_dir, log_dir, rtl_dict = get_paths({
        'rtl_bridge': '../../rtl'
    })

    # ... setup verilog_sources, parameters, etc ...

    run(
        verilog_sources=verilog_sources,
        toplevel=f"bridge_axi4_flat_{num_masters}x{num_slaves}",
        module=module,
        testcase="cocotb_test_basic_routing",  # ← cocotb function
name
        parameters=rtl_parameters,
        sim_build=sim_build,
        # ...
    )
```

23.6 Critical Rules for This Subsystem

23.6.1 Rule #0: Attribution Format for Git Commits

IMPORTANT: When creating git commit messages for Bridge documentation or code:

Use:

```
Documentation and implementation support by Claude.
```

Do NOT use:

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

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

23.6.2 Rule #0.1: Testbench Architecture - MANDATORY SEPARATION

⚠️ THIS IS A HARD REQUIREMENT - NO EXCEPTIONS ⚠️

NEVER embed testbench classes inside test runner files!

The same testbench logic will be reused across multiple test scenarios. Having testbench code only in test files makes it COMPLETELY WORTHLESS for reuse.

MANDATORY Structure:

```
projects/components/bridge/
├── dv/
│   ├── tbclasses/                  # TB classes HERE (not in
framework!)
│    │    ├── __init__.py
│    │    └── bridge_axi4_flat_tb.py  ← REUSABLE TB CLASS
│    ├── components/               # Bridge-specific BFMs (if
needed)
│    │    └── __init__.py
│    ├── scoreboards/              # Bridge-specific scoreboards
│    │    └── __init__.py
│    └── tests/                    # Test runners
│         ├── conftest.py           ← MANDATORY pytest config
│         ├── fub_tests/
│         │    └── basic/
│         │         └── test_bridge_axi4_2x2.py  ← TEST RUNNER ONLY
(imports TB)
│         ├── integration_tests/
│         │    └── test_bridge_multiport.py     ← TEST RUNNER ONLY
│         └── system_tests/
│              └── test_bridge_system.py         ← TEST RUNNER ONLY
```

Why This Matters:

1. **Reusability**: Same TB class used in:
   - Unit tests (`fub_tests/`)
   - Integration tests (`integration_tests/`)
   - System-level tests (`system_tests/`)
   - User projects (external imports)
2. **Maintainability**: Fix bug once in TB class, all tests benefit
3. **Composition**: TB classes can inherit/compose for complex scenarios

---

23.6.3 Rule #1: All Testbenches Inherit from TBBase

Every testbench class MUST inherit from TBBase:

```python
from CocoTBFramework.tbclasses.shared.tbbase import TBBase

class BridgeAXI4FlatTB(TBBase):
    """Testbench for Bridge crossbar - inherits base functionality"""

    def __init__(self, dut, num_masters=2, num_slaves=2, **kwargs):
        super().__init__(dut)
        # Bridge-specific initialization
```

TBBase Provides: - Clock management (`start_clock`, `wait_clocks`) - Reset utilities (`assert_reset`, `deassert_reset`) - Logging (`self.log`) - Progress tracking (`mark_progress`) - Safety monitoring (timeouts, memory limits)

23.6.4 Rule #2: Mandatory Testbench Methods

Every testbench class MUST implement these three methods:

```
async def setup_clocks_and_reset(self):
    """Complete initialization - starts clocks and performs reset"""
    await self.start_clock('aclk', freq=10, units='ns')

    # Set config signals before reset (if needed)
    # self.dut.cfg_param.value = initial_value

    # Reset sequence
    await self.assert_reset()
    await self.wait_clocks('aclk', 10)
    await self.deassert_reset()
    await self.wait_clocks('aclk', 5)

async def assert_reset(self):
    """Assert reset signal (active-low for AXI4)"""
    self.dut.aresetn.value = 0

async def deassert_reset(self):
    """Deassert reset signal"""
    self.dut.aresetn.value = 1
```

Why Required: - Consistency across all testbenches - Reusability for mid-test resets - Clear test structure and intent

---

23.6.5 Rule #3: Use GAXI Components for Protocol Handling

For Bridge testing, use GAXI Master/Slave components for AXI4 channel handling:

```
from CocoTBFramework.components.gaxi.gaxi_master import GAXIMaster
from CocoTBFramework.components.gaxi.gaxi_slave import GAXISlave
from CocoTBFramework.components.axi4.axi4_field_configs import
AXI4FieldConfigHelper

# ✓ CORRECT: Use GAXI for AXI4 channels
self.aw_master = GAXIMaster(
    dut=dut,
    title="AW_M0",
    prefix="s0_axi4_",
    clock=clock,

field_config=AXI4FieldConfigHelper.create_aw_field_config(id_width,
addr_width, 1),
    pkt_prefix="aw",
    multi_sig=True,
    log=log
)
```

Never manually drive AXI4 valid/ready handshakes - Use GAXI components.

23.6.6 Rule #4: Queue-Based Verification

For simple in-order verification, use direct monitor queue access:

```
# ✓ CORRECT: Direct queue access
aw_pkt = self.aw_slave._recvQ.popleft()
w_pkt = self.w_slave._recvQ.popleft()

# Verify
assert aw_pkt.addr == expected_addr
assert w_pkt.data == expected_data

# ✗ WRONG: Memory model for simple test
memory_model = MemoryModel()  # Unnecessary complexity
```

When to Use Memory Models: - ✗ Simple in-order tests → Use queue access - ✗ Single-master systems → Use queue access - √ Complex out-of-order scenarios → Memory model may help - √ Multi-master with address overlap → Memory model tracks state

---

23.7 TOML/CSV-Based Bridge Generator (Phase 2 Complete)

23.7.1 Overview

The Bridge generator creates parameterized SystemVerilog crossbars from TOML configuration files with CSV connectivity matrices, eliminating manual RTL editing for complex interconnects.

Key Benefits: - **Human-readable configuration** - TOML for ports, CSV for connectivity - **Custom signal prefixes** - Each port has unique prefix (rapids_m_axi_, apb0_, etc.) - **Channel-specific masters** - Write-only (wr), read-only (rd), or full (rw) - **Interface modules** - Timing isolation via axi4_master/slave wrappers with configurable skid depths - **Automatic converters** - Width and protocol conversion inserted automatically - **Resource efficient** - Only generates needed channels and converters

23.7.2 Quick Start

1. Create bridge_mybridge.toml:

```
[bridge]
  name = "bridge_mybridge"
  description = "Custom bridge example"

  # Default skid buffer depths (can be overridden per port)
  defaults.skid_depths = {ar = 2, r = 4, aw = 2, w = 4, b = 2}

  masters = [
    {name = "cpu", prefix = "cpu_m_axi", id_width = 4, addr_width =
32, data_width = 64, user_width = 1,
     interface = {type = "axi4_master"}},
    {name = "dma", prefix = "dma_m_axi", id_width = 4, addr_width =
32, data_width = 512, user_width = 1,
     interface = {type = "axi4_master", skid_depths = {ar = 4, r = 8,
aw = 4, w = 8, b = 4}}}
  ]

  slaves = [
    {name = "ddr", prefix = "ddr_s_axi", id_width = 4, addr_width =
32, data_width = 512, user_width = 1,
     base_addr = 0x00000000, addr_range = 0x80000000, interface =
{type = "axi4_slave"}},
    {name = "sram", prefix = "sram_s_axi", id_width = 4, addr_width =
32, data_width = 256, user_width = 1,
     base_addr = 0x80000000, addr_range = 0x80000000, interface =
{type = "axi4_slave"}}
  ]
```

2. Create bridge_mybridge_connectivity.csv:

```
master\slave,ddr,sram
cpu,1,1
dma,1,1
```

3. Generate bridge:

```
cd projects/components/bridge/bin
python3 bridge_generator.py --ports test_configs/bridge_mybridge.toml
# Auto-finds bridge_mybridge_connectivity.csv

# Or use bulk generation
python3 bridge_generator.py --bulk bridge_batch.csv
```

Result: Complete SystemVerilog module with: - Custom port prefixes per port - Timing isolation via interface wrappers - Only needed AXI4 channels (wr/rd/rw optimized) - Width converters for data mismatches - Internal crossbar instantiation - APB/AXI4-Lite converter integration points

23.7.3 Configuration Format Details

TOML Port Configuration:

The primary format is now **TOML** (preferred over CSV for better structure and interface configuration):

Port Specifications: - `name` - Unique identifier (cpu, dma, ddr, etc.) - `prefix` - Signal prefix (cpu_m_axi_, ddr_s_axi_, etc.) - `id_width` - AXI4 ID width in bits - `addr_width` - Address width in bits - `data_width` - Data width in bits (32, 64, 128, 256, 512) - `user_width` - AXI4 user signal width - `base_addr` - Slave base address (slaves only) - `addr_range` - Slave address range (slaves only) - `interface` - Interface wrapper configuration (optional) - `type` - "axi4_master", "axi4_slave", "axi4_master_mon", "axi4_slave_mon", or omit for direct connection - `skid_depths` - Per-channel buffer depths: {ar, r, aw, w, b} (valid: 2, 4, 6, 8)

CSV Connectivity Matrix:

```
master\slave,slave0,slave1,slave2
master0,1,0,1
master1,0,1,1
```

- `1` = connected, `0` = not connected
- Partial connectivity supported (not all masters to all slaves)
- Auto-detected based on TOML filename: `bridge_name.toml` → `bridge_name_connectivity.csv`

Legacy CSV Format:

For backwards compatibility, the generator still supports CSV port files: - See `test_configs/README.md` for migration guide from CSV to TOML - TOML is now preferred for new bridges (better structure, interface config support)

23.7.4 Channel-Specific Masters (Phase 2 Feature)

Why Channel-Specific? Real hardware often has dedicated read or write masters. Generating all 5 AXI4 channels wastes resources:

Traditional (wasteful):

```
// Write-only master gets unused read channels
input  logic [63:0]  rapids_descr_m_axi_awaddr,  // ✓ USED
input  logic [511:0] rapids_descr_m_axi_wdata,   // ✓ USED
output logic [7:0]   rapids_descr_m_axi_bid,     // ✓ USED
input  logic [63:0]  rapids_descr_m_axi_araddr,  // ✗ UNUSED (50%
waste!)
output logic [511:0] rapids_descr_m_axi_rdata,   // ✗ UNUSED
```

Channel-Specific (optimized):

```
rapids_descr_wr,master,axi4,wr,rapids_descr_m_axi_,512,64,8,N/A,N/A
```

Generated:

```
// Write-only master - only AW, W, B channels
input  logic [63:0]  rapids_descr_m_axi_awaddr,
input  logic [511:0] rapids_descr_m_axi_wdata,
output logic [7:0]   rapids_descr_m_axi_bid,
// ✓ NO READ CHANNELS (araddr, rdata, etc.)
```

Resource Savings: - 40-60% fewer ports for dedicated masters - Only necessary width converters instantiated - Channel-aware direct connection wiring - Faster synthesis, smaller netlists

23.7.5 Example: RAPIDS-Style Configuration

RAPIDS Architecture: - Descriptor write master (wr) - Writes descriptors to memory - Sink write master (wr) - Writes incoming packets to memory - Source read master (rd) - Reads outgoing packets from memory - CPU master (rw) - Full access for configuration

TOML Configuration (bridge_rapids.toml):

```
[bridge]
  name = "bridge_rapids"
  description = "RAPIDS accelerator bridge"
  defaults.skid_depths = {ar = 2, r = 4, aw = 2, w = 4, b = 2}

  masters = [
    {name = "rapids_descr_wr", prefix = "rapids_descr_m_axi", channels
= "wr",
     id_width = 8, addr_width = 64, data_width = 512, user_width = 1,
     interface = {type = "axi4_master"}},
    {name = "rapids_sink_wr", prefix = "rapids_sink_m_axi", channels =
"wr",
     id_width = 8, addr_width = 64, data_width = 512, user_width = 1,
     interface = {type = "axi4_master"}},
    {name = "rapids_src_rd", prefix = "rapids_src_m_axi", channels =
"rd",
     id_width = 8, addr_width = 64, data_width = 512, user_width = 1,
     interface = {type = "axi4_master"}},
    {name = "cpu", prefix = "cpu_m_axi", channels = "rw",
     id_width = 4, addr_width = 32, data_width = 64, user_width = 1,
     interface = {type = "axi4_master"}}
  ]
  slaves = [
    {name = "ddr", prefix = "ddr_s_axi", protocol = "axi4",
     id_width = 8, addr_width = 64, data_width = 512, user_width = 1,
     base_addr = 0x80000000, addr_range = 0x80000000,
     interface = {type = "axi4_slave"}},
    {name = "apb_periph", prefix = "apb0_", protocol = "apb",
     addr_width = 32, data_width = 32,
     base_addr = 0x00000000, addr_range = 0x00010000}
  ]
```

Connectivity CSV (bridge_rapids_connectivity.csv):

```
master\slave,ddr,apb_periph
rapids_descr_wr,1,0
rapids_sink_wr,1,0
rapids_src_rd,1,0
cpu,1,1
```

Generated RTL Features: - rapids_descr_wr: 37 signals (write channels only) vs 61 signals (full) = **39% reduction** - rapids_sink_wr: 37 signals (write channels only) vs 61 signals (full) = **39% reduction** - rapids_src_rd: 24 signals (read channels only) vs 61 signals (full) = **61% reduction** - cpu_master: Width converters for 64b→512b upsize (both wr and rd converters) - ddr_controller: Direct 512b connection (no conversion) - apb_periph0: APB converter placeholder (Phase 3)

23.7.6 Common User Questions

Q: “How do I generate a bridge?”

A: Three steps:

1. Create TOML port configuration file
2. Create CSV connectivity matrix
3. Run bridge_generator.py

```
# Create bridge_mybridge.toml (port configuration)
# Create bridge_mybridge_connectivity.csv (connectivity matrix)

cd projects/components/bridge/bin
python3 bridge_generator.py --ports test_configs/bridge_mybridge.toml
# Auto-finds bridge_mybridge_connectivity.csv

# Or use bulk generation for multiple bridges
python3 bridge_generator.py --bulk bridge_batch.csv
```

Q: “What’s the difference between wr/rd/rw channels?”

A: Number of AXI4 channels generated:

- **rw** (read/write) - All 5 channels: AW, W, B, AR, R

- **wr** (write-only) - 3 channels: AW, W, B (no read channels)
- **rd** (read-only) - 2 channels: AR, R (no write channels)

Benefits: 40-60% fewer ports for dedicated masters, less logic, faster synthesis

Q: "Can I add timing isolation to ports?"

A: Yes, use interface configuration:

```
masters = [
  {name = "cpu", prefix = "cpu_m_axi", ...,
   interface = {type = "axi4_master", skid_depths = {ar = 2, r = 4, aw
= 2, w = 4, b = 2}}}
]
```

Available interface types: - `"axi4_master"` - Timing isolation on master port - `"axi4_slave"` - Timing isolation on slave port - `"axi4_master_mon"` - Timing + monitoring - `"axi4_slave_mon"` - Timing + monitoring - Omit `interface` field for direct connection

Q: "Can I mix AXI4 and APB slaves?"

A: Yes, use protocol field in TOML:

```
slaves = [
  {name = "ddr", prefix = "ddr_s_axi", protocol = "axi4", ...},
  {name = "apb0", prefix = "apb0_", protocol = "apb", ...}
]
```

Generator inserts AXI2APB converters automatically (Phase 3 for full implementation).

Q: "What if data widths don't match?"

A: Generator inserts width converters automatically:

```
masters = [
  {name = "cpu", data_width = 64, ...},  # 64b master
]
slaves = [
  {name = "ddr", data_width = 512, ...}  # 512b slave
]
# Generator creates width converter: 64b → 512b
```

Q: "Do all masters need to connect to all slaves?"

A: No, use partial connectivity in CSV:

```
master\slave,ddr,sram,apb
rapids_descr,1,1,0     # Connects to ddr and sram only
cpu,1,1,1              # Connects to all three
```

23.7.7 Generator Output Structure

Generated File Contains:

1. **Module Header** - Parameterized with NUM_MASTERS, NUM_SLAVES, widths
2. **Port Declarations** - Custom prefix per port, channel-specific signals
3. **Internal Signals** - Crossbar interface arrays (xbar_m_, *xbar_s_*)
4. **Width Converters** - Master-side upsize instances (channel-aware)
5. **Crossbar Instance** - Internal AXI4 full crossbar
6. **Direct Connections** - For matching-width interfaces
7. **APB Converters** - Placeholder TODO comments (Phase 3)

Example Output Size: - Simple 2x2 bridge: ~400 lines - Complex 5x3 with mixed protocols: ~900 lines - Includes comprehensive comments and structure

23.7.8 Testing Generated Bridges

For Pure AXI4 Bridges (Working Now):

```
# Generate bridge
python3 bridge_csv_generator.py --ports ports.csv --connectivity
conn.csv --name my_bridge --output ../rtl/

# Create testbench (use BridgeAXI4FlatTB from dv/tbclasses/)
cd ../dv/tests/fub_tests/basic
pytest test_my_bridge.py -v
```

For Mixed AXI4/APB (Requires Phase 3): APB converter placeholders need implementation before end-to-end testing.

23.8 Bridge Architecture Quick Reference

23.8.1 Generated Bridge Crossbar Structure

```
module bridge_axi4_flat_2x2 #(
    parameter NUM_MASTERS = 2,
    parameter NUM_SLAVES = 2,
    parameter DATA_WIDTH = 32,
    parameter ADDR_WIDTH = 32,
    parameter ID_WIDTH = 4
) (
    input  logic aclk,
    input  logic aresetn,
    // Master-side interfaces (slave ports on bridge)
    // AW, W, B, AR, R channels for each master

    // Slave-side interfaces (master ports on bridge)
    // AW, W, B, AR, R channels for each slave
);
```

23.8.2 Key Features

1. **5-Channel Implementation:** Complete AW, W, B, AR, R routing
2. **Round-Robin Arbitration:** Per-slave arbitration with grant locking
3. **Transaction ID Tracking:** Distributed RAM for B/R response routing
4. **ID-Based Routing:** Enables out-of-order responses
5. **Configurable Parameters:** NxM topology, data/addr/ID widths

23.8.3 Address Map

Default address map (configurable): - Slave 0: 0x00000000 - 0x0FFFFFFF (256MB) - Slave 1: 0x10000000 - 0x1FFFFFFF (256MB) - Slave N: N * 0x10000000

23.9 Test Organization

23.9.1 Test Hierarchy

```
projects/components/bridge/dv/tests/
├── conftest.py                 # Pytest configuration with fixtures
├── fub_tests/                  # Functional unit block tests
│   └── basic/
│       ├── test_bridge_axi4_2x2.py       # Basic 2x2 tests
│       ├── test_bridge_axi4_4x4.py       # Full 4x4 tests
│       └── test_bridge_axi4_routing.py   # Routing tests
├── integration_tests/          # Multi-bridge scenarios
│   └── test_bridge_cascade.py
└── system_tests/               # Full system tests
    └── test_bridge_dma.py
```

23.9.2 Test Levels

Basic (FUB) Tests: - Individual bridge functionality - Address routing - ID tracking - Arbitration

Integration Tests: - Multi-bridge cascades - Complex topologies - Cross-bridge transactions

System Tests: - Full DMA transfers - Realistic traffic patterns - Performance validation

23.10 Common User Questions and Responses

23.10.1 Q: "How does the Bridge work?"

A: Direct answer:

The Bridge AXI4 crossbar connects multiple AXI4 masters to multiple slaves:

1. **Address Decode (AW/AR):** Routes master requests to appropriate slave based on address
2. **Write Path:** AW → arbitration → slave, W follows locked grant
3. **Read Path:** AR → arbitration → slave, R returns via ID table
4. **Arbitration:** Per-slave round-robin with grant locking until burst complete
5. **Response Routing:** B/R responses use ID lookup tables (not grant-based)

Key Features: - Out-of-order response support via ID tracking - Burst-aware arbitration (grant locked until WLAST/RLAST) - Configurable NxM topology - Single-clock domain

📖 **See:** - `projects/components/bridge/PRD.md` - Complete specification - `projects/components/bridge/bin/bridge_generator.py` - Generator implementation

23.10.2 Q: "How do I generate a Bridge?"

A: Use the bridge_generator.py tool:

```
cd projects/components/bridge/bin
python bridge_generator.py --masters 2 --slaves 4 --data-width 32 --
addr-width 32 --id-width 4 --output ../rtl/bridge_axi4_flat_2x4.sv
```

Generated files: - RTL: `projects/components/bridge/rtl/bridge_axi4_flat_<config>.sv` - Contains complete 5-channel crossbar with ID tracking

23.10.3 Q: "How do I test a Bridge?"

A: Use the Bridge testbench class:

```
# Import from project area
import sys
sys.path.insert(0, os.path.join(os.path.dirname(__file__),
'../../../..'))
from projects.components.bridge.dv.tbclasses.bridge_axi4_flat_tb
import BridgeAXI4FlatTB

@cocotb.test()
```

```
async def test_basic(dut):
    tb = BridgeAXI4FlatTB(dut, num_masters=2, num_slaves=2)
    await tb.setup_clocks_and_reset()

    # Send write to slave 0
    await tb.write_transaction(master_idx=0, address=0x00001000,
data=0xDEADBEEF)

    # Verify routing
    assert len(tb.aw_slaves[0]._recvQ) > 0, "Slave 0 should receive
AW"
```

Run tests:

```
cd projects/components/bridge/dv/tests/fub_tests/basic
pytest test_bridge_axi4_2x2.py -v
```

23.11 Anti-Patterns to Avoid

23.11.1 ✗ Anti-Pattern 1: Embedded Testbench Classes

```
✗ WRONG: TB class in test file
class BridgeTB:
    """NOT REUSABLE - WRONG LOCATION"""

✓ CORRECT: Import from project area
import sys
sys.path.insert(0, os.path.join(os.path.dirname(__file__),
'../../../..'))
from projects.components.bridge.dv.tbclasses.bridge_axi4_flat_tb
import BridgeAXI4FlatTB
```

23.11.2 ✗ Anti-Pattern 2: Manual AXI4 Handshaking

```
✗ WRONG: Manual signal driving
self.dut.s0_axi4_awvalid.value = 1
while self.dut.s0_axi4_awready.value == 0:
    await RisingEdge(self.clock)

✓ CORRECT: Use GAXI components
await self.aw_master.send(aw_pkt)
```

23.11.3 ✗ Anti-Pattern 3: Memory Models for Simple Tests

```
✗ WRONG: Unnecessary complexity
memory = MemoryModel()
memory.write(addr, data)
```

```
result = memory.read(addr)

✓ CORRECT: Direct queue verification
aw_pkt = self.aw_slave._recvQ.popleft()
assert aw_pkt.addr == expected_addr
```

23.12 Quick Reference

23.12.1 Finding Existing Components

```
# Bridge TB class (in PROJECT AREA, not framework!)
cat projects/components/bridge/dv/tbclasses/bridge_axi4_flat_tb.py

# Bridge generator
cat projects/components/bridge/bin/bridge_generator.py

# Test examples
ls projects/components/bridge/dv/tests/fub_tests/basic/
```

23.12.2 Common Commands

```
# Generate 2x2 bridge
python projects/components/bridge/bin/bridge_generator.py --masters 2
--slaves 2 --output
projects/components/bridge/rtl/bridge_axi4_flat_2x2.sv

# Run tests
pytest projects/components/bridge/dv/tests/fub_tests/basic/ -v

# Lint generated RTL
verilator --lint-only
projects/components/bridge/rtl/bridge_axi4_flat_2x2.sv
```

23.13 Remember

1. 🏗️ **MANDATORY: Testbench architecture** - TB classes in framework, tests import them
2. 📁 **MANDATORY: Directory structure** - Follow RAPIDS/AMBA pattern exactly
3. 📋 **MANDATORY: conftest.py** - Must exist in dv/tests/
4. 🎯 **Use GAXI components** - Never manually drive AXI4 handshakes

5. 📊 **Queue-based verification** - Simple tests use direct queue access
6. 🏛️ **Three-layer architecture** - TB (framework) + Test (runner) + Scoreboard (verification)
7. ⚙️ **Three mandatory methods** - setup_clocks_and_reset, assert_reset, deassert_reset
8. 🔍 **Search first** - Use existing components before creating new ones
9. 📈 **Test scalability** - Support basic/medium/full test levels
10. 💯 **100% success** - All tests must achieve 100% success rate

---

23.14 PDF Generation Location

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

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

Quick Command: Use the provided shell script:

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

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

📖 **See:** `bin/md_to_docx.py` for complete implementation details

Version: 1.0 **Last Updated:** 2025-10-18 **Maintained By:** RTL Design Sherpa Project