RTL Design Sherpa

Bridge Micro-Architecture Specification 1.0

January 3, 2026

Table of Contents

List of Figures

List of Tables

List of Waveforms

No waveforms in this document.

1 Bridge Mas Index

Generated: 2026-01-04

2 Document Information

2.1 Bridge Micro-Architecture Specification

Property	Value
Document Title	Bridge Micro-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 Micro-Architecture Specification (MAS) provides detailed implementation information for the Bridge component, covering:

- Block-level architecture and internal structure
- FSM state diagrams and transition tables
- ID management and CAM architecture
- Signal-level interface details
- Width and protocol converter implementation
- Timing diagrams and debug information

For high-level architecture, system integration, and performance overview, refer to the companion **Bridge Hardware Architecture Specification (HAS)**.

2.4 Intended Audience

- RTL designers implementing or modifying Bridge
- Verification engineers developing testbenches
- System integrators debugging Bridge behavior
- Technical staff requiring implementation details

2.5 Related Documents

- **Bridge HAS** - Hardware Architecture Specification (high-level)
- **PRD.md** - Product requirements and overview
- **CLAUDE.md** - AI development guide

3 Overview

3.1 Bridge Micro-Architecture

This document describes the internal micro-architecture of Bridge, a multi-protocol AXI4 crossbar generator. Bridge creates parameterized SystemVerilog RTL from CSV/TOML configuration files.

3.2 Key Capabilities

3.2.1 Multi-Protocol Support

Bridge supports three AMBA protocols:

Table 1.1: Supported Protocols

Protocol	Use Case	Conversion
AXI4 Full	High-bandwidth memory	Direct or width convert
AXI4-Lite	Register access	Protocol downgrade
APB	Low-speed peripherals	Full conversion

3.2.2 Channel-Specific Masters

Bridge generates optimized port interfaces:

Table 1.2: Channel-Specific Master Types

Type	Channels	Signals	Use Case
Full (rw)	AW, W, B, AR, R	100%	CPU, config master
Write-only (wr)	AW, W, B	~60%	DMA write engine
Read-only (rd)	AR, R	~40%	DMA read engine

3.2.3 Automatic Converters

Bridge inserts converters automatically:

- **Width converters** - Handle data width mismatches
- **Protocol converters** - AXI4 to APB conversion
- **Per-path optimization** - Only where needed

3.3 Architecture Summary

3.3.1 Block Organization

```
Bridge NxM
├── Master Adapters (M instances)
│   ├── Skid buffers
│   └── ID extension
├── Crossbar Core
│   ├── Address decode
│   ├── Per-slave arbiters
│   └── Channel muxes
├── Slave Routers (N instances)
│   ├── Protocol converters
│   └── Width converters
└── Response Routing
    ├── ID extraction
    └── Master demux
```

3.3.2 Signal Flow

```
Master → Adapter → Decode → Arbiter → Mux → Converter → Slave
                                                  ↓
Master ← Demux ← ID Extract ← Response ← Converter ← Slave
```

3.4 Document Organization

Table 1.3: Document Organization

Chapter	Content
Ch 2	Block Descriptions
Ch 3	FSM Design
Ch 4	ID Management
Ch 5	Converters
Ch 6	Generated RTL
Ch 7	Verification

3.5 Related Documentation

- **Bridge HAS** - High-level architecture and integration
- **PRD.md** - Product requirements
- **Generator source** - `bin/bridge_csv_generator.py`

4 2.1 Master Adapter

The Master Adapter is a per-master module that provides timing isolation, protocol normalization, and request preparation for the crossbar interconnect. Each master port in the bridge has its own dedicated adapter instance.

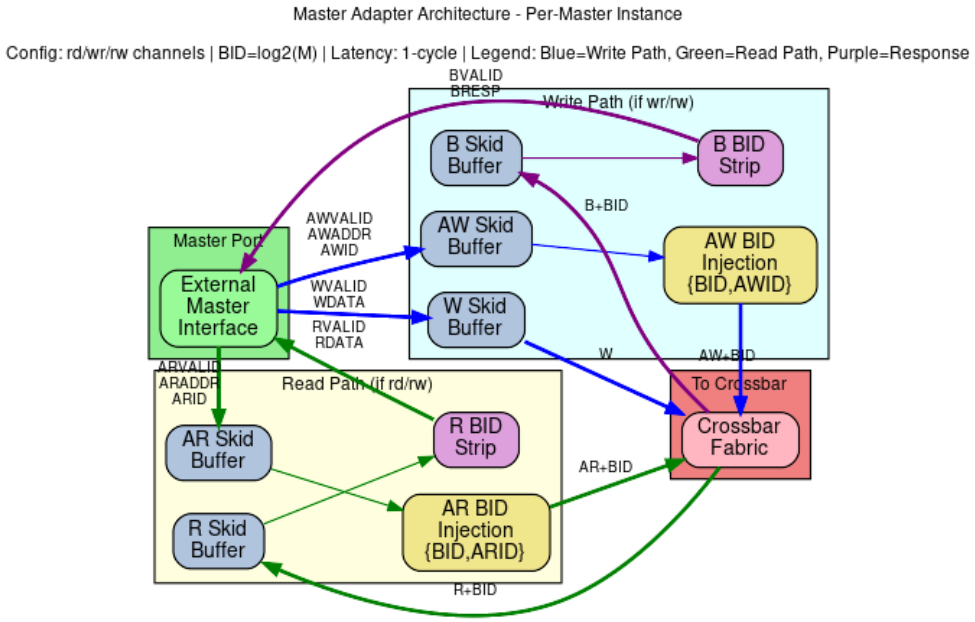
4.1 2.1.1 Purpose and Function

The Master Adapter serves several critical functions:

1. **Timing Isolation**: Inserts pipeline registers (skid buffers) to break combinatorial paths from master to crossbar
2. **Channel Specialization**: Separates read-only, write-only, and read-write masters for optimal resource utilization
3. **Bridge ID Injection**: Adds internal tracking IDs to transactions for response routing
4. **Protocol Normalization**: Ensures all transactions meet crossbar assumptions and constraints
5. **Backpressure Management**: Handles ready/valid handshaking with single-cycle latency

4.2 2.1.2 Block Diagram

4.2.1 Figure 2.1: Master Adapter Architecture



Master Adapter Architecture

The figure shows Master Adapter architecture with per-master skid buffers, bridge ID injection/stripping, and channel specialization for rd/wr/rw configurations.

4.3 2.1.3 Channel Specialization

The Bridge generator creates three types of master adapters based on the channel usage specified in the configuration:

4.3.1 Read-Only Masters

Channels: AR (Address Read), R (Read Data)

```
Configuration: "rd" or "read"
RTL Generated: adapter_master_rd_<id>.sv

Optimizations:
- No write address channel logic
- No write data channel logic
- No write response channel logic
- Reduced arbitration participation (read path only)
```

4.3.2 Write-Only Masters

Channels: AW (Address Write), W (Write Data), B (Write Response)

```
Configuration: "wr" or "write"
RTL Generated: adapter_master_wr_<id>.sv

Optimizations:
- No read address channel logic
- No read data channel logic
- Reduced arbitration participation (write path only)
```

4.3.3 Read-Write Masters

Channels: AR, R, AW, W, B (Full AXI4 interface)

```
Configuration: "rw" or "readwrite"
RTL Generated: adapter_master_rw_<id>.sv

Full Functionality:
- All five AXI4 channels implemented
- Participates in both read and write arbitration
- Maximum flexibility but larger resource footprint
```

4.4 2.1.4 Skid Buffer Architecture

Each AXI4 channel passes through a skid buffer for timing isolation. The skid buffer implements:

4.4.1 Registered Forward Path

```
// Simplified skid buffer concept
always_ff @(posedge clk) begin
    if (!rst_n) begin
        valid_reg <= 1'b0;
    end else if (!valid_reg || ready_out) begin
        valid_reg <= valid_in;
        data_reg  <= data_in;
    end
end
```

4.4.2 Single-Cycle Backpressure Response

- When downstream is not ready, accepts one additional beat into holding register
- Signals ready_in = 0 in same cycle to upstream
- No combinatorial paths between upstream and downstream

4.4.3 Pipeline Depth

- Default: 1 stage (skid buffer)
- Configurable: Up to 8 stages for high-frequency designs
- Trade-off: Latency vs. timing closure

Performance Impact: - Adds 1 cycle latency per channel - Enables higher clock frequencies - Prevents critical paths through crossbar

4.5 2.1.5 Bridge ID Management

4.5.1 ID Width Calculation

```
BID_WIDTH = clog2(num_masters)

Examples:
- 2 masters  → BID_WIDTH = 1
- 4 masters  → BID_WIDTH = 2
- 8 masters  → BID_WIDTH = 3
- 16 masters → BID_WIDTH = 4
```

4.5.2 ID Injection (Request Path)

For each master adapter, a unique constant Bridge ID is appended to the AXI transaction ID:

```
AR Channel:
  Internal ARID = {MASTER_BID[BID_WIDTH-1:0], External
ARID[ARID_WIDTH-1:0]}
  Internal ARID Width = BID_WIDTH + ARID_WIDTH

AW Channel:
  Internal AWID = {MASTER_BID[BID_WIDTH-1:0], External
AWID[AWID_WIDTH-1:0]}
  Internal AWID Width = BID_WIDTH + AWID_WIDTH
```

Example: 4-master system, external ARID_WIDTH = 4

```
Master 0: BID = 2'b00, External ID = 4'h3 → Internal ID = 6'b00_0011
Master 1: BID = 2'b01, External ID = 4'h3 → Internal ID = 6'b01_0011
Master 2: BID = 2'b10, External ID = 4'h5 → Internal ID = 6'b10_0101
Master 3: BID = 2'b11, External ID = 4'hA → Internal ID = 6'b11_1010
```

4.5.3 ID Stripping (Response Path)

The Bridge ID is removed from responses before returning to the master:

```
R Channel:
  External RID = Internal RID[ARID_WIDTH-1:0]
  Bridge routes based on Internal RID[BID_WIDTH+ARID_WIDTH-
```

```
1:ARID_WIDTH]

B Channel:
  External BID = Internal BID[AWID_WIDTH-1:0]
  Bridge routes based on Internal BID[BID_WIDTH+AWID_WIDTH-
1:AWID_WIDTH]
```

This ensures: - Master sees original transaction IDs - Bridge internally tracks which master originated each transaction - Responses route back to correct master even with ID reuse across masters

4.6 2.1.6 Protocol Normalization

The Master Adapter enforces several protocol requirements:

4.6.1 Valid Address Ranges

- Checks addresses against slave address maps
- Flags out-of-range accesses for error handling
- Prevents deadlocks from illegal addresses

4.6.2 Burst Constraints

- Validates ARLEN/AWLEN vs. 4KB boundary rules
- Ensures burst types are supported (FIXED, INCR, WRAP)
- Checks SIZE vs. DATA_WIDTH compatibility

4.6.3 Signal Defaults

- Provides default values for unused signals
- Ensures ARCACHE/AWCACHE have legal values
- Sets ARPROT/AWPROT based on configuration

4.7 2.1.7 Interface Specifications

4.7.1 External Master Interface (per master)

```
Input Channels (from master):
- ARVALID, ARREADY, ARADDR, ARBURST, ARCACHE, ARID, ARLEN,
  ARLOCK, ARPROT, ARQOS, ARSIZE

- AWVALID, AWREADY, AWADDR, AWBURST, AWCACHE, AWID, AWLEN,
  AWLOCK, AWPROT, AWQOS, AWSIZE

- WVALID, WREADY, WDATA, WSTRB, WLAST
```

```
Output Channels (to master):
- RVALID, RREADY, RDATA, RID, RLAST, RRESP

- BVALID, BREADY, BID, BRESP
```

4.7.2 Internal Crossbar Interface (per master)

```
Output Channels (to crossbar):
- AR* signals + Internal ARID (wider)
- AW* signals + Internal AWID (wider)
- W* signals (unchanged)

Input Channels (from crossbar):
- R* signals + Internal RID (wider)
- B* signals + Internal BID (wider)
```

4.8 2.1.8 Resource Utilization

4.8.1 Per-Master Adapter Resources (Typical)

Read-Write Master (64-bit data, 32-bit addr, 4-bit ID):

```
Logic Elements:  ~200-400 (depending on ID width and features)
Registers:       ~150-250 (pipeline stages + control)
Block RAM:       0 (CAM is in crossbar core)

Breakdown:
- Skid buffers (5 channels × ~30 regs each): ~150 regs
- ID manipulation logic: ~50 LEs
- Control FSMs: ~100 LEs
- Routing logic: ~50 LEs
```

Read-Only Master: ~60% of read-write resources
Write-Only Master: ~65% of read-write resources

4.8.2 Scaling Considerations

Resource usage scales with: - Number of masters (linear scaling, one adapter per master) - ID width (logarithmic: +BID_WIDTH bits per transaction) - Pipeline depth (linear: deeper pipelines = more registers) - Data width (linear: wider data = wider skid buffers)

4.9 2.1.9 Timing Characteristics

4.9.1 Latency

Request Path (Master → Crossbar): - Minimum: 1 cycle (skid buffer) - With 4-stage pipeline: 4 cycles - With ID injection: +0 cycles (combinatorial within stage)

Response Path (Crossbar → Master): - Minimum: 1 cycle (skid buffer) - With 4-stage pipeline: 4 cycles - With ID stripping: +0 cycles (combinatorial within stage)

Total Round-Trip Overhead: 2-8 cycles (depending on pipeline depth)

4.9.2 Throughput

- **Maximum**: 1 transaction per cycle per channel (fully pipelined)
- **Backpressure**: Propagates with 1-cycle latency
- **Burst Performance**: No degradation; bursts flow continuously when ready

4.9.3 Critical Paths

Typical critical paths (if skid buffers not used): - Master ARVALID → Crossbar arbitration logic - Crossbar grant → Master ARREADY - Slave RDATA → Master RDATA

Solution: Skid buffers break all these paths at the cost of 1-cycle latency.

4.10 2.1.10 Configuration Parameters

4.10.1 Per-Master Parameters (from TOML/CSV)

```
[[masters]]
name = "cpu"
channels = "rw"              # "rd", "wr", or "rw"
arid_width = 4               # External ID width
awid_width = 4               # Can differ from ARID
addr_width = 32              # Address bus width
data_width = 64              # Data bus width
pipeline_depth = 1           # Skid buffer stages (1-8)
```

4.10.2 Global Parameters (affect all adapters)

```
[bridge]
internal_data_width = 64     # Crossbar data width
enable_width_conversion = true
bid_width = 2                # Calculated: clog2(num_masters)
```

4.11 2.1.11 Debug and Observability

4.11.1 Recommended Debug Signals

For ILA or waveform capture:

```
- Master adapter input valid/ready (all channels)
- Master adapter output valid/ready (all channels)
- Bridge ID assignments per transaction
- Pipeline stage occupancy
- Backpressure events (ready = 0 while valid = 1)
```

4.11.2 Common Issues and Debug

Symptom: Master hangs waiting for READY
Check: - Is adapter receiving grants from arbiters? - Is slave responding to requests? - Is response path clear?

Symptom: Incorrect data returned to master
Check: - Bridge ID extraction logic - ID width mismatches - Response routing in crossbar

Symptom: Timing violations
Check: - Increase pipeline_depth parameter - Verify clock frequency vs. design complexity - Check for combinatorial loops

4.12 2.1.12 Future Enhancements

4.12.1 Planned Features

- **Dynamic Pipeline Depth**: Adjust depth based on operating frequency
- **FIFO Mode**: Deeper buffering (16-256 entries) for burst-intensive masters
- **QoS Support**: Priority-based arbitration hints
- **Performance Counters**: Transaction counts, stall cycles, utilization metrics

4.12.2 Under Consideration

- **Clock Domain Crossing**: Per-master clock domains with async FIFOs
- **Width Conversion at Adapter**: Move width logic to adapters for distributed conversion
- **Outstanding Transaction Tracking**: Windowing for improved OOO performance

---

Related Sections: - Section 2.3: Crossbar Core (interconnect architecture) - Section 2.4: Arbitration (how adapters compete for slaves) - Section 2.5: ID Management (CAM structures for response routing) - Section 3.2: Master Port Interface (signal-level specifications)

5 2.2 Slave Router

The Slave Router is responsible for address decoding and routing master requests to the appropriate slave. Each master has its own dedicated slave router instance that examines addresses and directs transactions to one of the configured slaves or generates error responses for out-of-range addresses.

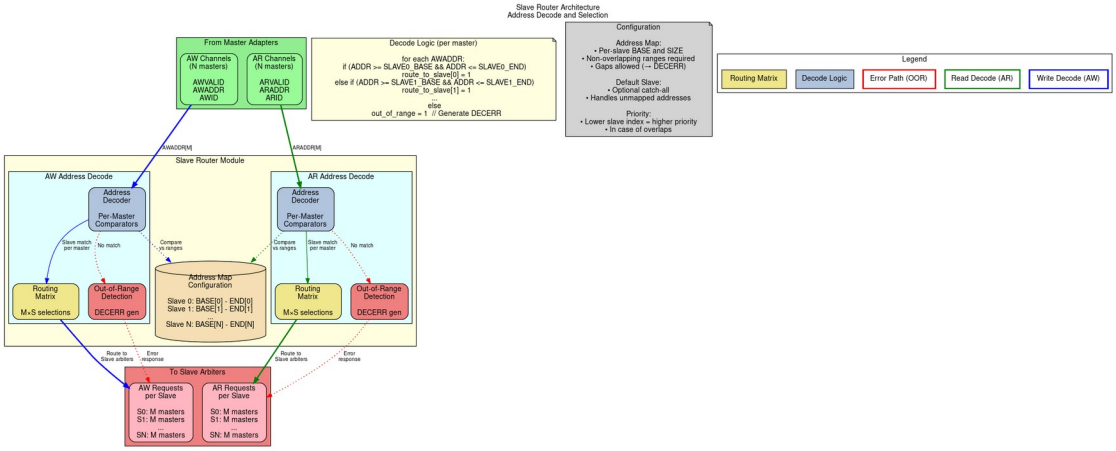
5.1 2.2.1 Purpose and Function

The Slave Router performs the following critical functions:

1. **Address Decoding**: Matches request addresses against configured slave address ranges
2. **Request Routing**: Directs AR/AW/W channels to the selected slave
3. **Out-of-Range Detection**: Identifies addresses that don't map to any slave
4. **Error Response Generation**: Creates DECERR responses for invalid addresses
5. **Default Slave Support**: Routes unmapped addresses to optional default slave

5.2 2.2.2 Block Diagram

5.2.1 Figure 2.2: Slave Router Architecture



Slave Router Architecture

Slave router architecture showing address decoding, routing matrix, and out-of-range detection for AW and AR channels.

5.3 2.2.3 Address Decoding Algorithm

5.3.1 Configuration-Based Address Maps

Each slave is configured with:

```
[[slaves]]
name = "memory"
base_address = 0x4000_0000
size = 0x1000_0000        # 256 MB
default = false           # Not a default slave
```

The router generates address ranges:

```
Start Address = base_address
End Address   = base_address + size - 1
```

5.3.2 Decoding Priority

When multiple slaves have overlapping address ranges, the router uses **first-match priority**:

```
Priority Order = Order in configuration file (top to bottom)

Example:
  Slave 0: 0x0000_0000 - 0x0FFF_FFFF  (checked first)
  Slave 1: 0x1000_0000 - 0x1FFF_FFFF  (checked second)
  Slave 2: 0x2000_0000 - 0x2FFF_FFFF  (checked third)

Address 0x1000_5000 → Matches Slave 1
```

5.3.3 Range Checking Logic

For each address, the router performs:

```
// Simplified address decode logic
logic [NUM_SLAVES-1:0] slave_match;

for (int i = 0; i < NUM_SLAVES; i++) begin
    slave_match[i] = (addr >= SLAVE_BASE[i]) &&
                     (addr <= SLAVE_END[i]);
end

// Priority encode: Select first matching slave
logic [$clog2(NUM_SLAVES)-1:0] slave_select;
always_comb begin
    slave_select = 0;
    for (int i = 0; i < NUM_SLAVES; i++) begin
        if (slave_match[i]) begin
            slave_select = i;
```

```
            break;  // First match wins
        end
    end
end

// Out-of-range detection
logic oor = ~(|slave_match);  // No slaves matched
```

5.3.4 Power-of-Two Optimization

For slaves with power-of-two sizes starting at aligned addresses, simplified decode:

```
// Optimized decode for: base=0x8000_0000, size=0x1000_0000 (256MB)
// Mask off lower bits: Only check upper bits
logic match = (addr[31:28] == 4'h8);  // Much simpler than range check
```

The generator automatically detects and applies this optimization.

5.4 2.2.4 Request Routing

5.4.1 AR Channel Routing

Read address routing flow: 1. **Decode**: Determine target slave from ARADDR 2. **Check OOR**: If no slave matches, flag for error response 3. **Route**: Send AR transaction to selected slave's arbiter 4. **Track**: Remember routing decision for R channel responses

```
ARADDR → Decoder → Slave Selection → AR to Slave Arbiter
                        ↓
                  If OOR → Error Response Generator
```

5.4.2 AW/W Channel Routing

Write transactions require coordinated routing:

1. **AW Phase**:
   - Decode AWADDR to determine target slave
   - Route AW transaction to selected slave's arbiter
   - **Store routing decision** for subsequent W beats
2. **W Phase**:
   - **Follow AW routing** (W channel has no address)
   - Route all W beats to same slave as AW
   - Continue until WLAST = 1

```
AWADDR → Decoder → Slave Selection → AW to Slave Arbiter
                                        ↓
                                   Store Routing
                                        ↓
```

```
W[0..N] ──────────────────────────────────→ W to Same Slave
(until WLAST)
```

5.4.3 Write Data Tracking FSM

```
State Machine for W Channel Routing:

IDLE:
  - Wait for AWVALID && AWREADY
  - On handshake: Store target slave, goto WRITING

WRITING:
  - Route W beats to stored slave
  - On WVALID && WREADY && WLAST: goto IDLE

Error Handling:
  - If AW was OOR: Discard W beats, generate BRESP error
```

5.5 2.2.5 Out-of-Range Handling

5.5.1 Detection

An address is out-of-range (OOR) if:

```
∀ slaves[i]: (address < base[i]) OR (address > end[i])
```

Unless a default slave is configured.

5.5.2 Error Response Generation

When OOR is detected:

Read Transactions (AR → R):

```
1. Accept ARVALID (assert ARREADY)
2. Do NOT forward to any slave
3. Generate R response internally:
   - RID = Original ARID (with BID preserved)
   - RDATA = 0xBADDCAFE_DEADBEEF (debug pattern) or all zeros
   - RRESP = 2'b11 (DECERR)
   - RLAST = 1 (for each beat if burst)
4. Latency: Typically 2-3 cycles after AR acceptance
```

Write Transactions (AW/W → B):

```
1. Accept AWVALID (assert AWREADY)
2. Accept all W beats until WLAST (sink the data)
3. Do NOT forward to any slave
4. Generate B response internally:
   - BID = Original AWID (with BID preserved)
```

```
   - BRESP = 2'b11 (DECERR)
5. Latency: Typically 2-3 cycles after WLAST
```

5.5.3 Debug Data Patterns

For OOR read responses, configurable data patterns aid debugging:

```
Option 1: All zeros (default)
  RDATA = 64'h0000_0000_0000_0000

Option 2: Debug signature
  RDATA = 64'hBADD_CAFE_DEAD_BEEF

Option 3: Address echo (LSBs)
  RDATA = {32'hBADD_ADDR, ARADDR[31:0]}

Option 4: Master/Slave ID indicator
  RDATA = {8'hEE, Master_ID[7:0], Slave_ID[7:0], ARADDR[47:0]}
```

5.6 2.2.6 Default Slave Support

5.6.1 Configuration

```toml
[[slaves]]
name = "error_responder"
base_address = 0x0        # Ignored for default slave
size = 0x0                # Ignored for default slave
default = true            # Catch-all for unmapped addresses
```

5.6.2 Behavior

When a default slave is configured: - Addresses that don't match any specific slave → Routed to default slave - Default slave typically returns DECERR but with configurable response - Useful for prototyping (accept all addresses initially) - Can implement memory-mapped debug register for address capture

Note: Only ONE default slave allowed per bridge.

5.7 2.2.7 Address Aliasing

5.7.1 Multiple Slaves, Same Address

If configuration has overlapping ranges:

```toml
[[slaves]]
name = "fast_cache"
base_address = 0x8000_0000
size = 0x1000_0000
```

```
[[slaves]]
name = "slow_memory"
base_address = 0x8000_0000  # Same base!
size = 0x4000_0000
```

Result: First-match priority applies. All accesses go to `fast_cache`. The `slow_memory` range 0x9000_0000-0xBFFF_FFFF is **unreachable** from this master.

Warning: Generator can optionally flag this as error in DRC mode.

5.7.2 Intentional Aliasing Use-Cases

Legitimate uses of overlapping ranges: 1. **Cache hierarchy**: Small fast cache shadows larger slow memory 2. **Memory remapping**: Different views of same physical memory 3. **Peripheral mirroring**: Register block appears at multiple addresses

5.8 2.2.8 Configuration Parameters

5.8.1 Per-Router Parameters

```
# Router behavior is defined by slave configurations

[[slaves]]
name = "ddr_memory"
base_address = 0x8000_0000
size = 0x4000_0000           # 1 GB
default = false
oor_data_pattern = 0xDEAD    # Pattern for OOR reads
```

5.8.2 Global Parameters

```
[bridge]
enable_default_slave = false      # Allow default slave
strict_address_decode = true      # Flag overlapping ranges as errors
oor_response_latency = 2          # Cycles for OOR error response (2-
4)
```

5.9 2.2.9 Resource Utilization

5.9.1 Per-Router Resources (Typical)

4-slave configuration (32-bit address):

```
Logic Elements:  ~150-200 LEs
Registers:       ~50 regs
Block RAM:       0
```

```
Breakdown:
- Address comparators (4 slaves × ~30 LEs): ~120 LEs
- Priority encoder: ~20 LEs
- W channel FSM: ~30 regs, ~20 LEs
- OOR error generator: ~20 regs, ~10 LEs
```

8-slave configuration:

```
Logic Elements:  ~250-350 LEs (scales roughly with slave count)
Registers:       ~60 regs
```

5.9.2 Scaling Considerations

Resource usage scales with: - **Number of slaves**: Linear (each slave adds comparator logic) - **Address width**: Minimal impact (wider comparators, but same structure) - **Optimizations**: Power-of-two sizes reduce logic significantly

5.10 2.2.10 Timing Characteristics

5.10.1 Decode Latency

Combinatorial Decode (Default): - Address → Slave selection: 0 cycles (combinatorial) - Critical path: ARADDR → Slave arbiter request - May limit maximum frequency in large systems

Registered Decode (Optional): - Address → Slave selection: 1 cycle (registered) - Adds latency but breaks critical path - Recommended for >8 slaves or >300 MHz operation

5.10.2 Throughput

- **Maximum**: 1 transaction per cycle per master
- **No blocking**: Router does not stall; arbiters handle conflicts
- **Pipelining**: AR and AW decode in parallel (independent paths)

5.10.3 Critical Paths

Typical critical paths: - ARADDR → Address comparators → Priority encoder → Arbiter request - w/4 slaves: ~10-15 logic levels (FPGA-dependent) - w/8 slaves: ~12-18 logic levels

Mitigation: - Enable registered decode mode (+1 cycle latency) - Use power-of-two slave sizes (simplified compare) - Synthesizer optimization directives

5.11 2.2.11 Debug and Observability

5.11.1 Recommended Debug Signals

```
- Address decode outputs (which slave matched)
- OOR flags (per channel)
- Default slave hit counter
```

```
- W channel FSM state
- Routing decision storage (for W tracking)
```

5.11.2 Common Issues and Debug

Symptom: Reads return all zeros
Check: - Is address out-of-range? - Check slave base/size configuration - Verify address decode logic in waveform

Symptom: Write data goes to wrong slave
Check: - W channel tracking FSM state - Did AW routing complete before W started? - Check for AWVALID/AWREADY handshake

Symptom: Unexpected DECERR responses
Check: - Address decode configuration - Overlapping slave ranges (wrong priority) - Off-by-one in size calculations

5.12 2.2.12 Verification Considerations

5.12.1 Address Decode Tests

Critical test scenarios: 1. **Boundary conditions**: base_address, base_address + size - 1 2. **Just out-of-range**: base_address - 1, base_address + size 3. **Each slave**: Verify routing to correct slave 4. **Overlapping ranges**: Verify priority encoding 5. **Default slave**: Unmapped addresses route correctly

5.12.2 Write Tracking Tests

W channel FSM testing: 1. **Simple write**: Single AW, single W (WLAST=1) 2. **Burst write**: AW with AWLEN=7, eight W beats 3. **Back-to-back writes**: New AW before previous W completes 4. **Interleaved masters**: Multiple masters writing simultaneously (if supported)

5.12.3 OOR Error Tests

```
Test: Read from unmapped address
- Send AR to 0xFFFF_FFFF (assuming unmapped)
- Verify R response with RRESP = DECERR
- Verify RID matches ARID
- Check response latency

Test: Write to unmapped address
- Send AW to invalid address
- Send W data
- Verify B response with BRESP = DECERR
- Verify BID matches AWID
```

5.13 2.2.13 Performance Optimization

5.13.1 Techniques

1. Registered Decode (High Frequency)

```
Trade-off: +1 cycle latency for timing closure
Best for: >8 slaves, >300 MHz targets
```

2. Simplified Address Decode (Power-of-2)

```
Optimization: Mask-based compare instead of range check
Best for: Slaves with aligned, power-of-2 sizes
Savings: ~50% reduction in comparator logic
```

3. Parallel Decode (Low Logic)

```
Implementation: Separate decoders per address bit-field
Best for: Many non-overlapping slaves
Savings: Reduces logic depth for priority encoding
```

4. CAM-Based Decode (Many Slaves)

```
Trade-off: Uses block RAM for address lookup table
Best for: >16 slaves, non-contiguous ranges
Note: Requires RAM resources
```

5.14 2.2.14 Future Enhancements

5.14.1 Planned Features

- **Dynamic Address Remapping**: Runtime-configurable slave ranges
- **Transaction Filtering**: Block certain address ranges per-master
- **Priority Hints**: QoS-based prioritization (beyond first-match)
- **Address Translation**: Offset/mask transformations before slave routing

5.14.2 Under Consideration

- **Multi-region Slaves**: Slave spans multiple non-contiguous ranges
- **Secure Address Spaces**: Per-master access control lists
- **Debug Address Capture**: Log invalid addresses to register

---

Related Sections: - Section 2.1: Master Adapter (upstream from router) - Section 2.3: Crossbar Core (downstream arbitration) - Section 2.4: Arbitration (how routed requests compete) - Section 3.3: Slave Port Interface (target of routed requests)

6 2.3 Crossbar Core

The Crossbar Core is the central interconnect fabric that enables any-to-any connectivity between masters and slaves. It contains the arbitration logic, transaction routing, and response path management that form the heart of the bridge.

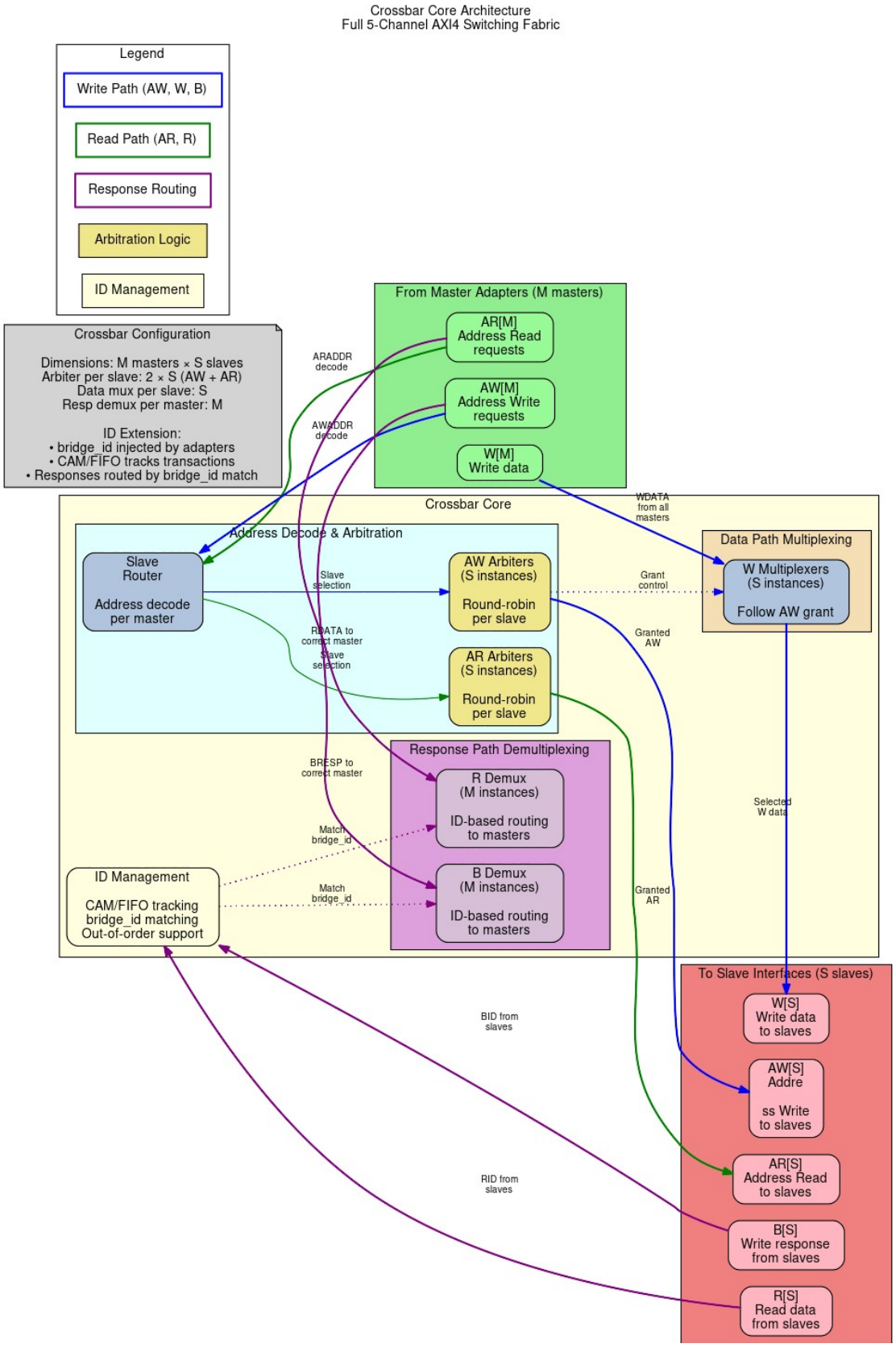
6.1 2.3.1 Purpose and Function

The Crossbar Core performs the following critical functions:

1. **Full Connectivity**: Provides complete N×M master-to-slave interconnect matrix
2. **Independent Arbitration**: Per-slave arbiters allow parallel access to different slaves
3. **Response Routing**: Directs slave responses back to originating masters using Bridge IDs
4. **Transaction Ordering**: Maintains AXI ordering requirements within address dependencies
5. **Backpressure Management**: Handles flow control across multiple concurrent transactions

6.2 2.3.2 Block Diagram

6.2.1 Figure 2.3: Crossbar Core Architecture


Crossbar Core Architecture
Full 5-Channel AXI4 Switching Fabric
Legend
Write Path (AW, W, B)
Read Path (AR, R)
Response Routing
Arbitration Logic
ID Management
Crossbar Configuration
Dimensions: M masters × S slaves
Arbiter per slave: 2 × S (AW + AR)
Data mux per slave: S
Resp demux per master: M
ID Extension:
• bridge_id injected by adapters
• CAM/FIFO tracks transactions
• Responses routed by bridge_id match
From Master Adapters (M masters)
AR[M]
Address Read
requests
AW[M]
Address Write
requests
W[M]
Write data
ARADDR
decode
AWADDR
decode
WDATA
from all
masters
Crossbar Core
Address Decode & Arbitration
Slave
Router
Address decode
per master
Slave
selection
AW Arbiters
(S instances)
Round-robin
per slave
RDATA to
correct master
Slave
selection
AR Arbiters
(S instances)
Round-robin
per slave
Grant
control
Granted
AW
Data Path Multiplexing
W Multiplexers
(S instances)
Follow AW grant
Selected
W data
Response Path Demultiplexing
BRESP to
correct master
R Demux
(M instances)
ID-based routing
to masters
Match
bridge_id
B Demux
(M instances)
ID-based routing
to masters
Match
bridge_id
Granted
AR
ID Management
CAM/FIFO tracking
bridge_id matching
Out-of-order support
To Slave Interfaces (S slaves)
W[S]
Write data
to slaves
AW[S]
Addre
ss Write
to slaves
AR[S]
Address Read
to slaves
B[S]
Write response
from slaves
R[S]
Read data
from slaves
BID from
slaves
RID from
slaves

Crossbar Core Architecture

Crossbar core architecture showing complete M×S switching fabric with address decode, per-slave arbitration, data path multiplexing, and ID-based response routing.

6.3 2.3.3 Connectivity Matrix

6.3.1 Full N×M Crossbar

The bridge implements a **non-blocking crossbar** where: - N masters can each access different slaves simultaneously - Conflicts occur only when multiple masters target the same slave - Independent read and write paths increase parallelism

Example: 4 masters, 3 slaves

```
        S0   S1   S2
    M0  ●    ●    ●
    M1  ●    ●    ●
    M2  ●    ●    ●
    M3  ●    ●    ●
```

```
● = Connection available
```

6.3.2 Request Path Multiplexing

For each slave, requests from all masters are multiplexed:

```
Slave 0 Request Inputs:
  - M0 → S0 (AR, AW, W channels)
  - M1 → S0 (AR, AW, W channels)
  - M2 → S0 (AR, AW, W channels)
  - M3 → S0 (AR, AW, W channels)

Arbiter selects one master per channel per cycle
  → Grants propagate through MUX
  → Selected master's transaction forwarded to slave
```

6.3.3 Response Path Demultiplexing

Responses from slaves are demultiplexed back to masters:

```
Slave 0 Response Outputs (R, B channels):
  - Extract Bridge ID from RID/BID
  - Route to corresponding master (M0, M1, M2, or M3)
  - Strip Bridge ID before delivering to master adapter

Uses CAM (Content Addressable Memory) for fast ID lookup
```

6.4 2.3.4 Request Path Architecture

6.4.1 Per-Slave Request Arbitration

Each slave has **independent arbiters** for: 1. **AR Channel**: Read address arbitration (all masters competing) 2. **AW/W Channels**: Write address/data arbitration (all masters competing)

This separation allows: - Simultaneous read and write to same slave (if slave supports it) - Independent grant decisions for AR vs. AW channels - Better throughput for mixed read/write workloads

6.4.2 Request Multiplexers

After arbitration, multiplexers select granted master's signals:

```
// Simplified AR channel MUX for Slave 0
always_comb begin
    case (ar_grant_s0)
        2'b00: begin  // Master 0 granted
            s0_arvalid = m0_arvalid;
            s0_araddr  = m0_araddr;
            s0_arid    = m0_arid;    // Includes Bridge ID
            // ... other AR signals
        end
        2'b01: begin  // Master 1 granted
            s0_arvalid = m1_arvalid;
            s0_araddr  = m1_araddr;
            s0_arid    = m1_arid;
            // ... other AR signals
        end
        // ... cases for M2, M3
    endcase
end
```

6.4.3 Backpressure Propagation

Ready signals flow back from slave through arbiter to granted master:

```
Slave S0 ARREADY → Arbiter → Granted Master ARREADY
                           ↘ Other Masters ARREADY = 0
```

This ensures: - Only granted master sees READY from slave - Non-granted masters see READY = 0 (backpressure) - No combinatorial loops in ready path (registered arbitration)

6.5 2.3.5 Response Path Architecture

6.5.1 Bridge ID Extraction

Response routing uses the Bridge ID embedded in transaction IDs:

```
R Channel:
  Slave RID = {BID[BID_WIDTH-1:0], Original_ID[ID_WIDTH-1:0]}
  Extract: Master_Index = BID
  Route R response to Master[Master_Index]

B Channel:
  Slave BID = {BID[BID_WIDTH-1:0], Original_ID[ID_WIDTH-1:0]}
  Extract: Master_Index = BID
  Route B response to Master[Master_Index]
```

6.5.2 CAM-Based Routing (Optional)

For large master counts or OOO responses, a CAM tracks outstanding transactions:

```
CAM Entry Structure:
  - Internal ID (with BID)
  - Master index
  - Transaction type (read/write)
  - Timestamp (for timeout detection)

Lookup:
  Input: RID or BID from slave
  Output: Master index for routing
  Latency: 1 cycle (registered CAM)
```

Benefit: Handles complex scenarios like ID reordering, burst interleaving

6.5.3 Response Demultiplexers

Based on extracted Bridge ID, responses are routed:

```
// Simplified R channel DEMUX from Slave 0
logic [BID_WIDTH-1:0] master_id;
assign master_id = s0_rid[TOTAL_ID_WIDTH-1:ID_WIDTH];  // Extract BID

always_comb begin
    // Default: No masters receive response
    m0_rvalid = 1'b0;
    m1_rvalid = 1'b0;
    m2_rvalid = 1'b0;
    m3_rvalid = 1'b0;

    // Route to indicated master
```

```
    case (master_id)
        2'b00: begin
            m0_rvalid = s0_rvalid;
            m0_rdata  = s0_rdata;
            m0_rid    = s0_rid[ID_WIDTH-1:0];  // Strip BID
            // ... other R signals
        end
        2'b01: begin
            m1_rvalid = s0_rvalid;
            // ... route to M1
        end
        // ... M2, M3
    endcase
end
```

6.5.4 Multi-Slave Response Merging

When multiple slaves can respond simultaneously, arbitration ensures: - Only one slave's response delivered per master per cycle - Fair arbitration if multiple slaves have responses for same master - No response loss (responses queued until master ready)

6.6 2.3.6 AXI Ordering Requirements

The crossbar maintains AXI ordering rules:

6.6.1 Read-After-Write (RAW) Ordering

Rule: Read from address must see data from earlier write to same address

Crossbar Behavior: - Ordering enforced at slave level (slave handles RAW within itself) - Crossbar does NOT reorder transactions to same slave from same master - Different masters to same slave: No ordering guaranteed (slave must handle)

6.6.2 Write-After-Write (WAW) Ordering

Rule: Writes to overlapping addresses must complete in issue order

Crossbar Behavior: - Same master to same slave: Order preserved by arbiter (FIFO grant queue) - Different masters to same slave: Slave responsible for write ordering

6.6.3 Out-of-Order (OOO) Completion

Allowed: - Read responses can return out-of-order (different RIDs) - Reads to different slaves can complete in any order - Writes to different slaves can complete in any order

Crossbar Support: - Bridge ID tracking enables OOO response routing - Master sees responses in slave-determined order - Multi-master OOO requires careful slave design

6.7 2.3.7 Resource Utilization

6.7.1 Crossbar Core Resources

4 masters × 3 slaves configuration (64-bit data, 32-bit addr):

```
Logic Elements:  ~2000-3500 LEs
Registers:       ~800-1200 regs
Block RAM:       0-4 KB (if CAM used)

Breakdown per slave:
- Arbiter (4 masters, RR):        ~200 LEs, ~50 regs
- Request MUX (AR/AW/W):          ~400 LEs, ~100 regs
- Response DEMUX (R/B):           ~300 LEs, ~80 regs
- Control FSMs:                   ~100 LEs, ~50 regs

Total for 3 slaves: 3 × 1000 LEs = ~3000 LEs
Plus routing overhead: +500 LEs
```

6.7.2 Scaling with Masters and Slaves

Linear scaling: - Adding 1 master: +~500 LEs per slave (new arbiter input) - Adding 1 slave: +~1000 LEs (complete new slave port)

Example: 8 masters × 6 slaves

```
Estimated:  ~12,000 LEs, ~3000 regs
Block RAM:  ~8 KB (for CAM if enabled)
```

6.7.3 Optimization Techniques

1. **Read-Only/Write-Only Masters**: Reduces arbiter complexity by 40-50%
2. **Power-of-Two Master Count**: Simplifies BID width and routing logic
3. **Pipeline Stages**: Trading latency for frequency (deeper pipelines)

6.8 2.3.8 Timing Characteristics

6.8.1 Latency

Request Path (Master → Slave): - Arbiter decision: 1 cycle (registered) - MUX selection: 0 cycles (combinatorial) or +1 cycle (registered) - **Total**: 1-2 cycles through crossbar

Response Path (Slave → Master): - BID extraction: 0 cycles (combinatorial) - DEMUX routing: 0 cycles (combinatorial) or +1 cycle (registered) - **Total**: 0-1 cycles through crossbar

End-to-End (Master adapter → Slave adapter): - Adapters: 2 cycles (skid buffers) - Router: 0-1 cycles (decode) - Crossbar: 1-2 cycles (arbitration + MUX) - **Total**: 3-5 cycles minimum latency

6.8.2 Throughput

Best Case (no conflicts): - Each master to different slave: N transactions/cycle (N = master count) - Maximum aggregate bandwidth: N × data_width bits/cycle

Arbitration Limits: - Multiple masters to one slave: 1 transaction/cycle to that slave - Other slaves remain available for parallel access

Burst Performance: - Once granted, bursts flow at 1 beat/cycle - Grant held until burst completes (RLAST or WLAST)

6.9 2.3.9 Critical Paths

6.9.1 Common Critical Paths

1. **Arbiter Request → Grant**:
   - All masters' VALID signals → Arbiter logic → Grant decision
   - Depth: ~5-8 logic levels for 4 masters
2. **Grant → Ready Backpressure**:
   - Slave READY → Arbiter → Selected master READY
   - Depth: ~4-6 logic levels
3. **Response Demux**:
   - Slave RDATA/RID → BID extraction → Master select → Master RDATA
   - Depth: ~6-10 logic levels for 64-bit data

6.9.2 Mitigation Strategies

1. **Registered MUX/DEMUX**: +1 cycle latency, breaks paths
2. **Pipelined Arbitration**: Multi-cycle arbiter for >8 masters
3. **Hierarchical Crossbar**: For >16 masters, use tree structure

6.10 2.3.10 Configuration Parameters

6.10.1 Crossbar Parameters (from TOML)

```toml
[bridge]
num_masters = 4
num_slaves = 3
internal_data_width = 64
arbiter_type = "round_robin"       # "round_robin", "fixed_priority",
"weighted"
registered_mux = false             # true = +1 cycle, better timing
registered_demux = false           # true = +1 cycle, better timing
```

```
enable_cam = false                  # true = CAM-based routing
cam_depth = 16                      # Outstanding transactions tracked
```

6.11 2.3.11 Debug and Observability

6.11.1 Recommended Debug Signals

```
Per Slave:
- Arbiter grants (which master granted)
- Arbiter requests (which masters requesting)
- Request MUX outputs (VALID, READY, ADDR, ID)
- Response DEMUX inputs (VALID, READY, DATA, ID)

Global:
- Active transactions count
- Stall counters (arbiter conflicts)
- BID extraction errors
- CAM hit/miss (if enabled)
```

6.11.2 Performance Counters

Useful metrics for profiling:

```
- Transactions per slave (read, write separate)
- Arbiter conflict rate (requests denied due to grant  contention)
- Average grant latency
- Utilization per master (% cycles active)
- Utilization per slave (% cycles busy)
```

6.12 2.3.12 Common Issues and Debug

Symptom: Master hangs with VALID=1, READY=0
Check: - Is another master holding grant to this slave? - Is arbiter logic functioning (check grant signals)? - Is slave responding with READY?

Symptom: Response goes to wrong master
Check: - Bridge ID values (verify correct BID per master) - CAM contents (if used) - BID extraction logic (check bit positions)

Symptom: Throughput lower than expected
Check: - Arbiter conflicts (multiple masters to same slave?) - Pipeline depth (excessive latency reducing effective bandwidth?) - Burst efficiency (are bursts being granted properly?)

6.13 2.3.13 Verification Considerations

6.13.1 Functional Tests

1. **Single Master to Each Slave**: Verify basic connectivity

2. **All Masters to One Slave**: Stress arbiter fairness
3. **All Masters to All Slaves**: Maximum parallelism test
4. **OOO Responses**: Issue transactions with different latencies
5. **Burst Interleaving**: Multiple masters with bursts to same slave

6.13.2 Corner Cases

```
- Back-to-back grants (no idle cycles)
- Grant held for maximum burst length (256 beats)
- Rapid master switching (each gets 1 transaction then switches)
- Response while request in progress (pipelining)
- All masters requesting same slave simultaneously
```

6.13.3 Protocol Compliance

- AXI4 protocol checker at each master/slave interface
- Verify no READY → VALID dependencies (AXI violation)
- Check ID preservation through crossbar (modulo Bridge ID)
- Verify LAST signal handling

6.14 2.3.14 Future Enhancements

6.14.1 Planned Features

- **Weighted Round-Robin**: QoS support with configurable priorities
- **Slave-Side Arbitration Policies**: Per-slave arbiter configuration
- **Grant Prediction**: Speculative grant for lower latency
- **Congestion Control**: Throttling to prevent hotspots

6.14.2 Under Consideration

- **Partial Crossbar**: Configurable master-to-slave connectivity (not full mesh)
- **Multi-Tier Hierarchy**: For 32+ masters/slaves
- **Virtual Channels**: Separate channels for different traffic classes
- **Register Slicing**: Automatic pipeline insertion for timing

---

Related Sections: - Section 2.1: Master Adapter (request sources) - Section 2.2: Slave Router (address decode before arbitration) - Section 2.4: Arbitration (detailed arbiter algorithms) - Section 2.5: ID Management (CAM structures, Bridge ID tracking) - Section 3.1: Top-Level Integration (crossbar instantiation)

7 2.4 Arbitration

Arbitration is the mechanism by which the bridge decides which master gains access to a slave when multiple masters simultaneously request the same slave. Each slave has dedicated arbiters for its read and write channels, enabling fair and efficient resource allocation.

7.1 2.4.1 Purpose and Function

The arbiter performs the following critical functions:

1. **Conflict Resolution**: Selects one master when multiple masters request same slave
2. **Fairness**: Ensures all masters eventually get access (starvation-free)
3. **Throughput Optimization**: Minimizes idle cycles and maximizes utilization
4. **Grant Management**: Maintains grants for burst transactions
5. **Priority Enforcement**: Supports priority-based access policies (optional)

7.2 2.4.2 Arbitration Architecture

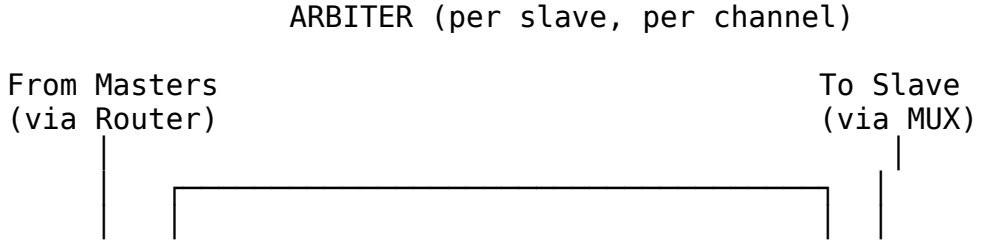
7.2.1 Per-Slave, Per-Channel Arbiters

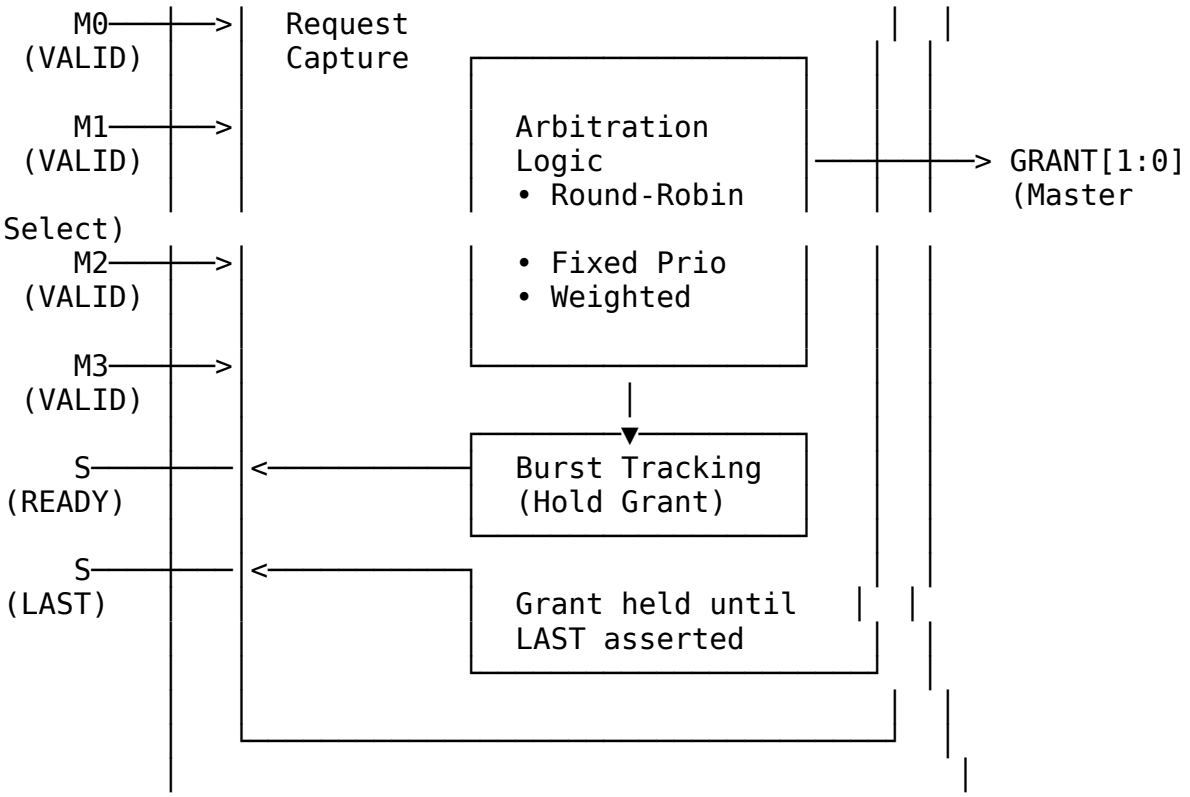
Each slave has **two independent arbiters**:

```
Slave 0:
  - AR Arbiter (Read Address Channel)
    * Inputs: M0_ARVALID, M1_ARVALID, M2_ARVALID, M3_ARVALID
    * Output: Grant[1:0] → Selects one master

  - AW Arbiter (Write Address Channel)
    * Inputs: M0_AWVALID, M1_AWVALID, M2_AWVALID, M3_AWVALID
    * Output: Grant[1:0] → Selects one master
    * Coordinates with W channel data flow
```

Independence: AR and AW arbiters operate independently, allowing simultaneous read and write to same slave (if slave supports full-duplex operation).

7.2.2 Block Diagram





7.3 2.4.3 Round-Robin Arbitration

7.3.1 Algorithm

The **Round-Robin (RR)** arbiter cycles through masters in order, giving each master one opportunity to access the slave:

```
Priority Order (rotates after each grant):
  Cycle 0: M0 > M1 > M2 > M3
  Cycle 1: M1 > M2 > M3 > M0  (M0 was granted, now lowest priority)
  Cycle 2: M2 > M3 > M0 > M1  (M1 was granted)
  Cycle 3: M3 > M0 > M1 > M2  (M2 was granted)
  Cycle 4: M0 > M1 > M2 > M3  (M3 was granted, cycle repeats)
```

7.3.2 Implementation

```
// Simplified Round-Robin arbiter (4 masters)
logic [1:0] last_grant;      // Which master was granted last
logic [3:0] request;         // Current requests (VALID signals)
logic [1:0] grant;           // Grant decision

always_ff @(posedge clk) begin
    if (!rst_n) begin
        last_grant <= 2'b00;
```

```
    end else if (|request && slave_ready) begin
        // Update last_grant when transaction completes
        if (grant_accepted) begin
            last_grant <= grant;
        end
    end
end

always_comb begin
    // Default: No grant
    grant = 2'b00;

    // Priority encode starting after last grant
    case (last_grant)
        2'b00: begin  // Last grant to M0, try M1, M2, M3, M0
            if (request[1])      grant = 2'b01;
            else if (request[2]) grant = 2'b10;
            else if (request[3]) grant = 2'b11;
            else if (request[0]) grant = 2'b00;
        end
        2'b01: begin  // Last grant to M1, try M2, M3, M0, M1
            if (request[2])      grant = 2'b10;
            else if (request[3]) grant = 2'b11;
            else if (request[0]) grant = 2'b00;
            else if (request[1]) grant = 2'b01;
        end
        // ... similar for M2, M3
    endcase
end
```

7.3.3 Characteristics

Fairness: Excellent - Each master gets equal access over time
Latency: Bounded - Maximum wait = (N-1) × transaction_time
Throughput: Optimal - No idle cycles when requests pending
Starvation: None - Every master eventually gets grant

Best Use Cases: - Peers with similar priority - General-purpose interconnects - Default choice for most bridges

7.4 2.4.4 Fixed-Priority Arbitration

7.4.1 Algorithm

The **Fixed-Priority** arbiter always grants to the highest-priority requesting master:

```
Priority Order (never changes):
  M0 > M1 > M2 > M3  (M0 always highest priority)
```

```
Grant Decision:
  if (M0_VALID) → Grant = M0
  else if (M1_VALID) → Grant = M1
  else if (M2_VALID) → Grant = M2
  else if (M3_VALID) → Grant = M3
```

7.4.2 Implementation

```
// Fixed-priority arbiter (4 masters)
logic [3:0] request;
logic [1:0] grant;

always_comb begin
    // Priority encoder: M0 highest, M3 lowest
    if (request[0])      grant = 2'b00;  // M0 highest priority
    else if (request[1]) grant = 2'b01;
    else if (request[2]) grant = 2'b10;
    else if (request[3]) grant = 2'b11;  // M3 lowest priority
    else                 grant = 2'b00;  // Default (no requests)
end
```

7.4.3 Characteristics

Fairness: Poor - Low-priority masters can starve
Latency: Variable - High-priority: minimal, Low-priority: unbounded
Throughput: Optimal - No idle cycles when requests pending
Starvation: Possible for low-priority masters

Best Use Cases: - Real-time systems with priority requirements - CPU (high) vs. DMA (low) scenarios - Time-critical masters need guaranteed access

7.4.4 Starvation Prevention

Optional enhancement: **Priority aging**

```
- Track cycles since last grant per master
- Temporarily boost priority of starved masters
- Reset age counter after grant

Example:
  M0 age = 100 cycles → Boost M0 above normal priority
  M1 age = 5 cycles   → Normal priority
```

7.5 2.4.5 Weighted Arbitration

7.5.1 Algorithm

The **Weighted** arbiter assigns different access rates to masters based on weights:

```
Configuration:
  M0 weight = 4  (50% of bandwidth)
  M1 weight = 2  (25% of bandwidth)
  M2 weight = 1  (12.5% of bandwidth)
  M3 weight = 1  (12.5% of bandwidth)

Grant Sequence (repeating pattern):
  M0, M0, M0, M0, M1, M1, M2, M3  (8 cycles total)
  M0 gets 4/8, M1 gets 2/8, M2 gets 1/8, M3 gets 1/8
```

7.5.2 Implementation

```
// Weighted arbiter using credit counter
logic [7:0] credits [0:3];  // Credit counters per master
logic [1:0] grant;

always_ff @(posedge clk) begin
    if (!rst_n) begin
        credits[0] <= 8'd4;  // M0 weight
        credits[1] <= 8'd2;  // M1 weight
        credits[2] <= 8'd1;  // M2 weight
        credits[3] <= 8'd1;  // M3 weight
    end else if (grant_accepted) begin
        // Decrement granted master's credits
        credits[grant] <= credits[grant] - 1;

        // Reload when all credits exhausted
        if (all_credits_zero) begin
            credits[0] <= 8'd4;
            credits[1] <= 8'd2;
            credits[2] <= 8'd1;
            credits[3] <= 8'd1;
        end
    end
end

always_comb begin
    // Grant to master with highest credits and active request
    grant = select_highest_credit_with_request(credits, request);
end
```

7.5.3 Characteristics

Fairness: Configurable - Proportional to weights
Latency: Bounded - Depends on weight ratio
Throughput: Near-optimal - Small overhead for credit tracking
Starvation: None - All masters get share per weight

Best Use Cases: - QoS requirements (guaranteed bandwidth) - Mixed workload (video + CPU + DMA) - Service-level agreements

7.6 2.4.6 Burst Handling

7.6.1 Grant Locking

Once granted, a master **holds the grant** until its burst completes:

```
AR Channel Burst:
  1. Master asserts ARVALID with ARLEN = 7 (8-beat burst)
  2. Arbiter grants to this master
  3. Grant held until slave returns RLAST
  4. Other masters blocked during this time

AW/W Channel Burst:
  1. Master asserts AWVALID with AWLEN = 15 (16-beat burst)
  2. Arbiter grants to this master
  3. Grant held until master asserts WLAST
  4. Other masters blocked during W data transfer
```

7.6.2 Implementation

```
// Burst grant locking
logic grant_locked;
logic [1:0] locked_master;

always_ff @(posedge clk) begin
    if (!rst_n) begin
        grant_locked <= 1'b0;
    end else begin
        if (grant_accepted && !last_beat) begin
            // Lock grant for burst
            grant_locked <= 1'b1;
            locked_master <= grant;
        end else if (last_beat) begin
            // Release grant when burst completes
            grant_locked <= 1'b0;
        end
    end
end
```

```
assign grant = grant_locked ? locked_master : arbiter_decision;
```

7.6.3 Fairness Considerations

Long bursts can block other masters: - Maximum AXI burst: 256 beats - At 1 cycle/beat: Up to 256 cycles blocked - Impacts latency for other masters

Mitigation: - Limit burst lengths in master configuration - Use burst interleaving (complex, not implemented in Phase 1) - Monitor arbiter stall counters

7.7 2.4.7 Resource Utilization

7.7.1 Per-Arbiter Resources

Round-Robin (4 masters):

```
Logic Elements:  ~150 LEs
Registers:       ~40 regs

Breakdown:
- Priority encoder:        ~80 LEs
- Last grant tracking:     ~10 regs
- Request capture:         ~20 regs
- Grant FSM:               ~40 LEs, ~10 regs
```

Fixed-Priority (4 masters):

```
Logic Elements:  ~80 LEs
Registers:       ~20 regs

Simpler than RR (no rotation logic)
```

Weighted (4 masters):

```
Logic Elements:  ~200 LEs
Registers:       ~60 regs

Additional credit counters and comparison logic
```

7.7.2 Scaling with Master Count

Resource usage scales approximately $\mathbf{O(N^2)}$ where N = master count:

```
2 masters:  ~50 LEs
4 masters:  ~150 LEs
8 masters:  ~400 LEs
16 masters: ~1200 LEs
```

The N^2 scaling comes from: - Priority encoder: Compares all pairs of masters - Grant MUX: N-to-1 multiplexing increases with N

7.8 2.4.8 Timing Characteristics

7.8.1 Arbitration Latency

Single-Cycle Arbitration (default):

```
Request → Grant Decision: 1 cycle
Grant Decision → MUX Output: 0 cycles (combinatorial)
Total: 1 cycle
```

Multi-Cycle Arbitration (high frequency):

```
Request → Grant Decision: 2-3 cycles (pipelined)
Improves timing at cost of latency
```

7.8.2 Critical Paths

Typical critical paths in arbiter: 1. **Request collection**: All master VALID signals → Arbiter input 2. **Priority encode**: Compare all requests → Grant decision 3. **Grant propagate**: Grant → MUX select → Slave interface

Path Depth: - 4 masters: ~6-8 logic levels - 8 masters: ~8-12 logic levels - 16 masters: ~12-16 logic levels

7.9 2.4.9 Configuration Parameters

7.9.1 Arbiter Configuration (TOML)

```toml
[bridge]
arbiter_type = "round_robin"    # "round_robin", "fixed_priority",
"weighted"

[bridge.arbitration]
pipeline_stages = 1             # 1-3 (more = better timing, higher
latency)
enable_priority_aging = false   # Prevent starvation in fixed-
priority
aging_threshold = 1000          # Cycles before priority boost

# Weighted arbitration weights (only if arbiter_type = "weighted")
[[bridge.arbitration.weights]]
master = "cpu"
weight = 4                      # Relative weight (1-255)

[[bridge.arbitration.weights]]
```

```
master = "dma"
weight = 2

[[bridge.arbitration.weights]]
master = "periph"
weight = 1
```

7.10 2.4.10 Debug and Observability

7.10.1 Recommended Debug Signals

```
Per Arbiter:
- Request vector (all masters requesting)
- Grant decision (which master granted)
- Grant locked status (burst in progress)
- Last grant (for RR rotation tracking)

Performance Counters:
- Grants per master (utilization)
- Denied requests per master (contention)
- Average grant latency per master
- Arbiter conflict cycles (multiple requests, one grant)
```

7.10.2 Common Issues and Debug

Symptom: Master never gets grant (starvation)
Check: - Arbiter type (fixed-priority can starve low-priority masters) - Other masters continuously requesting? - Enable priority aging if using fixed-priority

Symptom: Unfair access (one master dominates)
Check: - Arbiter type (should be round-robin for fairness) - Burst lengths (long bursts block others) - Request patterns (one master requesting more frequently)

Symptom: Low throughput despite requests
Check: - Arbiter conflicts (too many masters to one slave) - Pipeline depth (excessive arbitration latency) - Burst efficiency (short bursts waste cycles)

7.11 2.4.11 Verification Considerations

7.11.1 Test Scenarios

1. **Fair Access Test** (Round-Robin):

```
- All masters continuously request same slave
- Verify each master gets equal grants over 1000 cycles
- Expected: Each of 4 masters gets ~250 grants
```

2. **Priority Test** (Fixed-Priority):

```
- High-priority master requests intermittently
- Low-priority masters request continuously
- Verify high-priority always wins when requesting
```

3. **Burst Locking Test**:

```
- Master issues 16-beat burst
- Other masters request during burst
- Verify grant held until LAST
- Verify other masters blocked during burst
```

4. **Weighted Bandwidth Test**:

```
- Configure M0=4, M1=2, M2=1, M3=1
- All masters request continuously for 10000 cycles
- Verify grants proportional: M0≈50%, M1≈25%, M2≈12.5%, M3≈12.5%
```

7.11.2 Corner Cases

```
- Single master requesting (trivial grant)
- All masters requesting simultaneously (worst-case arbitration)
- Grant released and new grant same cycle (back-to-back)
- Master drops request after arbiter latency (stale grant)
- Maximum length burst (256 beats blocking)
```

7.12 2.4.12 Performance Impact

7.12.1 Arbiter Choice Impact

Round-Robin: - **Pros**: Fair, starvation-free, predictable latency - **Cons**: Cannot prioritize time-critical masters - **Use**: General-purpose, peer masters

Fixed-Priority: - **Pros**: Guaranteed low latency for high-priority masters - **Cons**: Low-priority can starve, unpredictable for low-priority - **Use**: Real-time, priority-sensitive systems

Weighted: - **Pros**: Configurable bandwidth allocation, QoS support - **Cons**: More complex, small overhead - **Use**: Mixed workload, SLA requirements

7.12.2 Arbiter Efficiency

Assuming all masters request same slave continuously:

```
Best Case: 100% efficiency (one master)
  - No arbitration conflicts
  - Zero idle cycles

Typical Case: 25-50% per-master efficiency (4 masters, round-robin)
  - Each master gets ~25% of time grants
  - Load balancing to other slaves improves
```

```
Worst Case: Heavy contention
  - 4 masters to 1 slave: 25% efficiency per master
  - Solution: Add slaves or use more slaves in parallel
```

7.13 2.4.13 Future Enhancements

7.13.1 Planned Features

- **Dynamic Weight Adjustment**: Runtime-configurable weights via registers
- **QoS Classes**: Multiple priority levels with configurable policies
- **Deadline-Based Arbitration**: Grant based on transaction deadlines
- **Predictive Arbitration**: Speculative grants to reduce latency

7.13.2 Under Consideration

- **Multi-Level Arbitration**: Hierarchical for >16 masters
- **Token Bucket**: Rate limiting per master
- **History-Based**: Learn access patterns and optimize grants
- **ECC Protection**: For arbitration state (safety-critical systems)

Related Sections: - Section 2.3: Crossbar Core (arbiter integration) - Section 2.1: Master Adapter (request sources) - Section 2.5: ID Management (transaction tracking during grants) - Chapter 6: Performance (arbiter impact on throughput)

8 2.5 ID Management

ID Management is the system by which the bridge tracks outstanding transactions and routes responses back to the originating master. This is accomplished through Bridge ID injection, Content Addressable Memory (CAM) structures, and ID translation logic.

8.1 2.5.1 Purpose and Function

The ID management system performs the following critical functions:

1. **Transaction Tracking**: Maintains association between requests and originating masters
2. **Response Routing**: Directs responses to correct master using embedded IDs
3. **Out-of-Order Support**: Handles responses returning in different order than issued
4. **ID Space Isolation**: Prevents ID conflicts between multiple masters

5. **Burst Management**: Tracks multi-beat bursts through the bridge

8.2 2.5.2 Bridge ID Concept

8.2.1 Problem Statement

Without ID management, the bridge cannot determine which master originated a transaction:

```
Scenario: Two masters with overlapping IDs
  Master 0: Issues read with ARID = 4'h5
  Master 1: Issues read with ARID = 4'h5 (same ID!)

When slave responds with RID = 4'h5, which master gets the response?
  → Ambiguous! Need additional information.
```

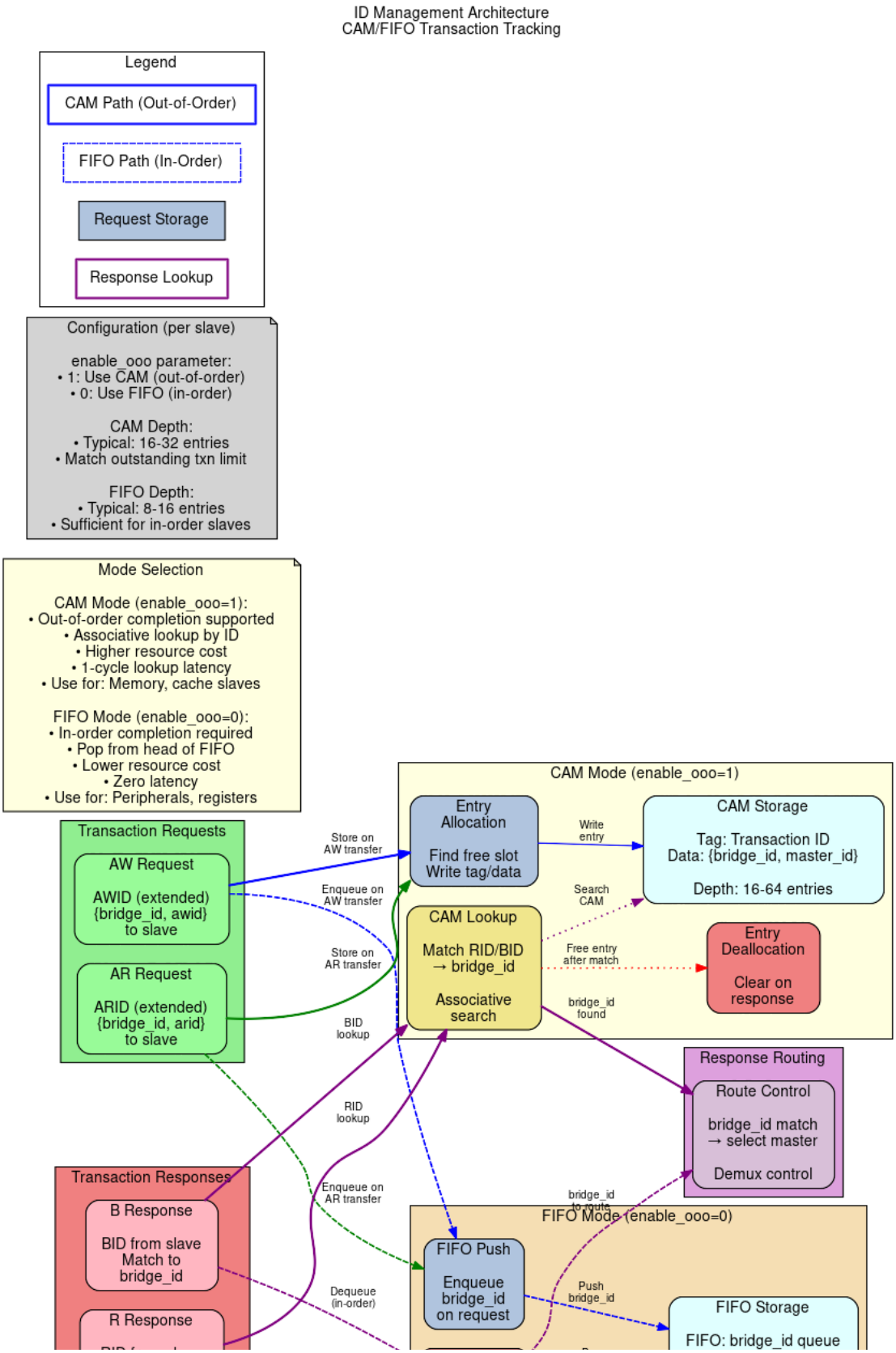
8.2.2 Solution: Bridge ID Injection

The bridge adds a **Bridge ID (BID)** to each transaction ID:

```
Internal ID = {Bridge_ID, Original_ID}

Master 0, ARID = 4'h5:
  Internal ARID = {2'b00, 4'h5} = 6'b00_0101

Master 1, ARID = 4'h5:
  Internal ARID = {2'b01, 4'h5} = 6'b01_0101

Now responses with RID = 6'b00_0101 → Route to Master 0
                    RID = 6'b01_0101 → Route to Master 1
```

8.3 2.5.3 Block Diagram

8.3.1 Figure 2.5: ID Management Architecture


ID Management Architecture
CAM/FIFO Transaction Tracking
Legend
CAM Path (Out-of-Order)
FIFO Path (In-Order)
Request Storage
Response Lookup
Configuration (per slave)
enable_ooo parameter:
• 1: Use CAM (out-of-order)
• 0: Use FIFO (in-order)
CAM Depth:
• Typical: 16-32 entries
• Match outstanding txn limit
FIFO Depth:
• Typical: 8-16 entries
• Sufficient for in-order slaves
Mode Selection
CAM Mode (enable_ooo=1):
• Out-of-order completion supported
• Associative lookup by ID
• Higher resource cost
• 1-cycle lookup latency
• Use for: Memory, cache slaves
FIFO Mode (enable_ooo=0):
• In-order completion required
• Pop from head of FIFO
• Lower resource cost
• Zero latency
• Use for: Peripherals, registers
CAM Mode (enable_ooo=1)
Entry Allocation
Find free slot
Write tag/data
Write entry
CAM Storage
Tag: Transaction ID
Data: {bridge_id, master_id}
Depth: 16-64 entries
Search CAM
CAM Lookup
Match RID/BID → bridge_id
Associative search
Free entry after match
Entry Deallocation
Clear on response
bridge_id found
Transaction Requests
AW Request
AWID (extended)
{bridge_id, awid}
to slave
AR Request
ARID (extended)
{bridge_id, arid}
to slave
Store on AW transfer
Enqueue on AW transfer
Store on AR transfer
BID lookup
RID lookup
Response Routing
Route Control
bridge_id match → select master
Demux control
Transaction Responses
B Response
BID from slave
Match to bridge_id
R Response
Enqueue on AR transfer
Dequeue (in-order)
bridge_id to route
FIFO Mode (enable_ooo=0)
FIFO Push
Enqueue bridge_id on request
Push bridge_id
FIFO Storage
FIFO: bridge_id queue

ID Management Architecture

ID management architecture showing CAM and FIFO modes for transaction tracking and response routing.

8.4 2.5.4 Bridge ID Width Calculation

8.4.1 Formula

```
BID_WIDTH = clog2(NUM_MASTERS)

Where clog2(x) = ceiling(log2(x))
```

8.4.2 Examples

```
2 masters:  clog2(2)  = 1 bit   (BID: 0, 1)
3 masters:  clog2(3)  = 2 bits  (BID: 00, 01, 10)
4 masters:  clog2(4)  = 2 bits  (BID: 00, 01, 10, 11)
5 masters:  clog2(5)  = 3 bits  (BID: 000-100)
8 masters:  clog2(8)  = 3 bits  (BID: 000-111)
16 masters: clog2(16) = 4 bits  (BID: 0000-1111)
```

8.4.3 ID Width Growth

```
Configuration: 4 masters, external ARID_WIDTH = 4

Internal ARID_WIDTH = BID_WIDTH + External ARID_WIDTH
                    = 2 + 4
                    = 6 bits

Slave sees 6-bit IDs
Master sees 4-bit IDs
Bridge translates between them
```

8.5 2.5.5 ID Injection (Request Path)

8.5.1 Read Address Channel (AR)

```
// ID injection at master adapter
logic [BID_WIDTH-1:0] master_bid;         // Constant per master
logic [ARID_WIDTH-1:0] external_arid;     // From master
logic [TOTAL_ARID_WIDTH-1:0] internal_arid;  // To crossbar

assign master_bid = MASTER_INDEX;  // M0=0, M1=1, M2=2, M3=3
assign internal_arid = {master_bid, external_arid};

// Example: Master 2, ARID = 4'h7
// internal_arid = {2'b10, 4'h7} = 6'b10_0111
```

8.5.2 Write Address Channel (AW)

```
// ID injection for write transactions
logic [BID_WIDTH-1:0] master_bid;
logic [AWID_WIDTH-1:0] external_awid;
logic [TOTAL_AWID_WIDTH-1:0] internal_awid;

assign internal_awid = {master_bid, external_awid};

// Note: AWID and ARID widths can differ per master
```

8.5.3 Injection Timing

Combinatorial (default): - ID concatenation done in same cycle as VALID assertion - Zero latency overhead - May contribute to critical path in high-frequency designs

Registered (optional): - Add pipeline register after ID injection - +1 cycle latency - Breaks critical path

8.6 2.5.6 ID Extraction (Response Path)

8.6.1 Read Data Channel (R)

```
// ID extraction at crossbar response router
logic [TOTAL_RID_WIDTH-1:0] internal_rid;   // From slave
logic [BID_WIDTH-1:0] extracted_bid;        // Upper bits
logic [RID_WIDTH-1:0] external_rid;         // Lower bits

// Extract Bridge ID
assign extracted_bid = internal_rid[TOTAL_RID_WIDTH-1:RID_WIDTH];

// Strip Bridge ID for master
assign external_rid = internal_rid[RID_WIDTH-1:0];

// Route based on BID
always_comb begin
    case (extracted_bid)
        2'b00: master0_rvalid = slave_rvalid;
        2'b01: master1_rvalid = slave_rvalid;
        2'b10: master2_rvalid = slave_rvalid;
        2'b11: master3_rvalid = slave_rvalid;
    endcase
end
```

8.6.2 Write Response Channel (B)

```
// Similar extraction for write responses
logic [TOTAL_BID_WIDTH-1:0] internal_bid;
```

```
logic [BID_WIDTH-1:0] extracted_bid;
logic [BID_WIDTH-1:0] external_bid;

assign extracted_bid = internal_bid[TOTAL_BID_WIDTH-1:BID_WIDTH];
assign external_bid = internal_bid[BID_WIDTH-1:0];

// Route to originating master
```

8.7 2.5.7 Content Addressable Memory (CAM)

8.7.1 When to Use CAM

Simple Configurations (No CAM needed): - Single master (BID unnecessary) - Direct ID mapping suffices - Lower resource usage

Complex Configurations (CAM beneficial): - Many masters (>8) - Out-of-order responses common - Multiple outstanding transactions per master - ID reordering within slave

8.7.2 CAM Structure

Entry Format:

Internal ID (6-12 bits	Master Index (2-4 bits)	Transaction Type (R/W)	Timestamp (counter)	Valid (1b)
10b	3b	1b	16b	1b

```
Total per entry: ~30 bits
CAM depth: 16-64 entries typical
```

8.7.3 CAM Operations

Allocation (Request Path):

```
1. Master issues AR/AW transaction
2. Find free CAM entry (Valid = 0)
3. Write entry:
   - Internal ID = {BID, External_ID}
   - Master Index = BID
   - Transaction Type = Read or Write
   - Timestamp = Current cycle count
   - Valid = 1
4. Forward transaction to slave
```

Lookup (Response Path):

```
1. Slave returns R/B response with Internal ID
2. CAM lookup: Search for matching Internal ID
```

```
3. Extract Master Index from matching entry
4. Route response to Master[Master Index]
5. If LAST: Deallocate entry (Valid = 0)
```

Associative Search: - All entries checked in parallel - 1-cycle lookup (registered CAM) - Match found → Returns master index - No match → Error condition (protocol violation)

8.8 2.5.8 CAM Implementation

8.8.1 Parallel CAM (Fast, Resource-Intensive)

```
// Parallel CAM structure (16 entries)
typedef struct packed {
    logic valid;
    logic [TOTAL_ID_WIDTH-1:0] internal_id;
    logic [BID_WIDTH-1:0] master_idx;
    logic txn_type;  // 0=read, 1=write
    logic [15:0] timestamp;
} cam_entry_t;

cam_entry_t cam [0:15];

// Parallel search
logic [15:0] match;
logic [3:0] match_idx;

for (genvar i = 0; i < 16; i++) begin
    assign match[i] = cam[i].valid &&
                      (cam[i].internal_id == response_id);
end

// Priority encode first match
assign match_idx = find_first_set(match);
assign routed_master = cam[match_idx].master_idx;
```

8.8.2 Sequential CAM (Slow, Resource-Efficient)

```
// Sequential CAM search (multi-cycle)
logic [3:0] search_idx;
logic searching;

always_ff @(posedge clk) begin
    if (search_start) begin
        search_idx <= 0;
        searching <= 1'b1;
    end else if (searching) begin
        if (cam[search_idx].valid &&
            cam[search_idx].internal_id == response_id) begin
```

```
            // Match found
            routed_master <= cam[search_idx].master_idx;
            searching <= 1'b0;
        end else if (search_idx == 15) begin
            // No match found (error)
            searching <= 1'b0;
            error <= 1'b1;
        end else begin
            search_idx <= search_idx + 1;
        end
    end
end
```

Trade-off: - Parallel: 1-cycle, ~2000 LEs for 16 entries - Sequential: 16 cycles, ~200 LEs for 16 entries

8.9 2.5.9 Outstanding Transaction Limits

8.9.1 Configuration

```
[bridge]
enable_cam = true
cam_depth = 16              # Max outstanding transactions

[[masters]]
name = "cpu"
max_outstanding_reads = 8   # Per-master limit
max_outstanding_writes = 8
```

8.9.2 Enforcement

When CAM full:

```
1. Master issues new request
2. Check CAM for free entry
3. If full:
   - ARREADY/AWREADY = 0 (backpressure)
   - Wait for response to free entry
4. When entry freed (RLAST/BVALID):
   - Accept new request
```

8.9.3 Sizing Guidelines

```
CAM Depth = Σ(max_outstanding per master) + Safety margin

Example: 4 masters, 4 outstanding each
  Minimum CAM depth = 4 × 4 = 16 entries
  Recommended depth = 20 entries (25% margin)
```

8.10 2.5.10 Resource Utilization

8.10.1 ID Injection/Extraction Only

Per Master (no CAM):

```
Logic Elements:  ~20-50 LEs
Registers:       ~10 regs

Simple bit concatenation and extraction
Minimal overhead
```

8.10.2 With CAM

CAM Resources (16 entries, 6-bit IDs):

```
Parallel CAM:
  Logic Elements:  ~2000 LEs
  Registers:       ~500 regs
  Block RAM:       0 (distributed)

BRAM-Based CAM:
  Logic Elements:  ~500 LEs
  Registers:       ~100 regs
  Block RAM:       1-2 KB

Sequential CAM:
  Logic Elements:  ~200 LEs
  Registers:       ~100 regs
  Block RAM:       0
```

8.10.3 Scaling

```
CAM Depth      Parallel      BRAM          Sequential
16 entries     ~2000 LEs     ~500 LEs      ~200 LEs
32 entries     ~4000 LEs     ~800 LEs      ~250 LEs
64 entries     ~8000 LEs     ~1200 LEs     ~300 LEs
```

8.11 2.5.11 Timing Characteristics

8.11.1 ID Injection Latency

Combinatorial (default): - 0 cycles overhead - Part of adapter pipeline stage

Registered: - +1 cycle latency - Breaks timing path

8.11.2 ID Extraction/Routing Latency

Direct Decode (no CAM): - 0 cycles (combinatorial BID extraction)

Parallel CAM: - 1 cycle (registered search)

Sequential CAM: - 1-N cycles (where N = CAM depth) - Average: N/2 cycles

8.11.3 End-to-End Impact

```
Configuration: No CAM, combinatorial ID management
  Request:  Master → +0 cycles → Crossbar
  Response: Slave → +0 cycles → Master
  Total: 0 cycles overhead

Configuration: Parallel CAM, registered
  Request:  Master → +1 cycle (allocation) → Crossbar
  Response: Slave → +1 cycle (lookup) → Master
  Total: 2 cycles overhead
```

8.12 2.5.12 Configuration Parameters

8.12.1 ID Management Configuration (TOML)

```toml
[bridge]
num_masters = 4
enable_cam = false              # Use CAM for ID tracking
cam_type = "parallel"           # "parallel", "bram", "sequential"
cam_depth = 16                  # Outstanding transaction capacity

[bridge.id_management]
registered_injection = false    # Register ID injection (+1 cycle)
registered_extraction = false   # Register ID extraction (+1 cycle)
enable_timeout = true           # Detect hung transactions
timeout_cycles = 10000          # Cycles before timeout error

[[masters]]
name = "cpu"
arid_width = 4                  # External ID width
awid_width = 4
max_outstanding_reads = 8       # CAM allocation limit
max_outstanding_writes = 8
```

8.13 2.5.13 Debug and Observability

8.13.1 Recommended Debug Signals

```
ID Injection:
- Master BID assignments (constant per master)
- External IDs (from masters)
- Internal IDs (after injection)

ID Extraction:
- Internal IDs (from slaves)
- Extracted BIDs
- External IDs (to masters)

CAM (if enabled):
- CAM occupancy (number of valid entries)
- Allocation/deallocation events
- Search hits/misses
- Timeout events
```

8.13.2 Performance Counters

```
- Total transactions tracked
- CAM hit rate
- CAM miss rate (should be 0)
- Average CAM occupancy
- Peak CAM occupancy
- Timeout errors
- ID width overhead (bits added per transaction)
```

8.14 2.5.14 Common Issues and Debug

Symptom: Response goes to wrong master
Check: - BID assignment (verify each master has unique BID) - ID width calculation (BID_WIDTH correct?) - Bit slicing in extraction logic - CAM contents (if used)

Symptom: CAM full, transactions stalling
Check: - CAM depth vs. outstanding transaction count - Are responses being received? (slave responsive?) - Timeout threshold (too short?) - Leaked entries (not deallocated after LAST)

Symptom: CAM miss errors
Check: - Slave returning incorrect IDs - ID corruption in transit - Race condition (deallocation before response complete)

8.15 2.5.15 Verification Considerations

8.15.1 Test Scenarios

1. **Basic ID Injection/Extraction**:

```
- Single master, single transaction
- Verify BID added correctly
- Verify BID stripped before response delivery
- Check external ID unchanged
```

2. **Multi-Master ID Isolation**:

```
- Multiple masters with same external IDs
- Verify responses route to correct master
- Check BID uniqueness prevents collisions
```

3. **CAM Capacity**:

```
- Issue max_outstanding transactions
- Verify backpressure when CAM full
- Issue responses to free entries
- Verify new transactions accepted
```

4. **Out-of-Order Responses**:

```
- Issue transactions: ID=1, ID=2, ID=3
- Return responses: ID=3, ID=1, ID=2
- Verify CAM correctly routes each
```

5. **CAM Timeout**:

```
- Issue transaction
- Slave doesn't respond within timeout_cycles
- Verify timeout error signaled
- Verify CAM entry deallocated (or flagged)
```

8.16 2.5.16 Future Enhancements

8.16.1 Planned Features

- **Dynamic CAM Depth**: Runtime adjustment based on utilization
- **Per-Master CAM Partition**: Guaranteed entries per master
- **ID Compression**: Reduce internal ID width for slaves with limited ID support
- **Transaction Ordering**: CAM tracks issue order for reordering enforcement

8.16.2 Under Consideration

- **Multi-Level CAM**: Hierarchical for large transaction counts
- **TCAM Support**: Partial ID matching (wildcards)
- **Error Injection**: Debug mode to test error handling
- **CAM Mirrors**: Redundant CAMs for fail-safe operation

Related Sections: - Section 2.1: Master Adapter (BID injection location) - Section 2.3: Crossbar Core (BID extraction, response routing) - Section 2.8: Response Routing (detailed routing logic) - Section 3.2: Master Port Interface (ID width specifications)

9 2.6 Width Conversion

Width Conversion is the mechanism by which the bridge adapts between masters and slaves with different data bus widths. The bridge uses a 64-bit internal data path, with conversion logic at the master and slave interfaces to support narrower or wider external widths.

9.1 2.6.1 Purpose and Function

Width conversion performs the following critical functions:

1. **Data Path Adaptation**: Converts between different data widths (8, 16, 32, 64, 128, 256 bits)
2. **Burst Splitting**: Divides wide transactions into multiple narrow transactions
3. **Burst Merging**: Combines multiple narrow beats into fewer wide beats
4. **Strobe Mapping**: Translates write strobes across width boundaries
5. **Address Alignment**: Adjusts addresses for width differences

9.2 2.6.2 Internal Data Width

9.2.1 64-Bit Standard

The bridge uses **64-bit (8-byte) internal data width** for the crossbar:

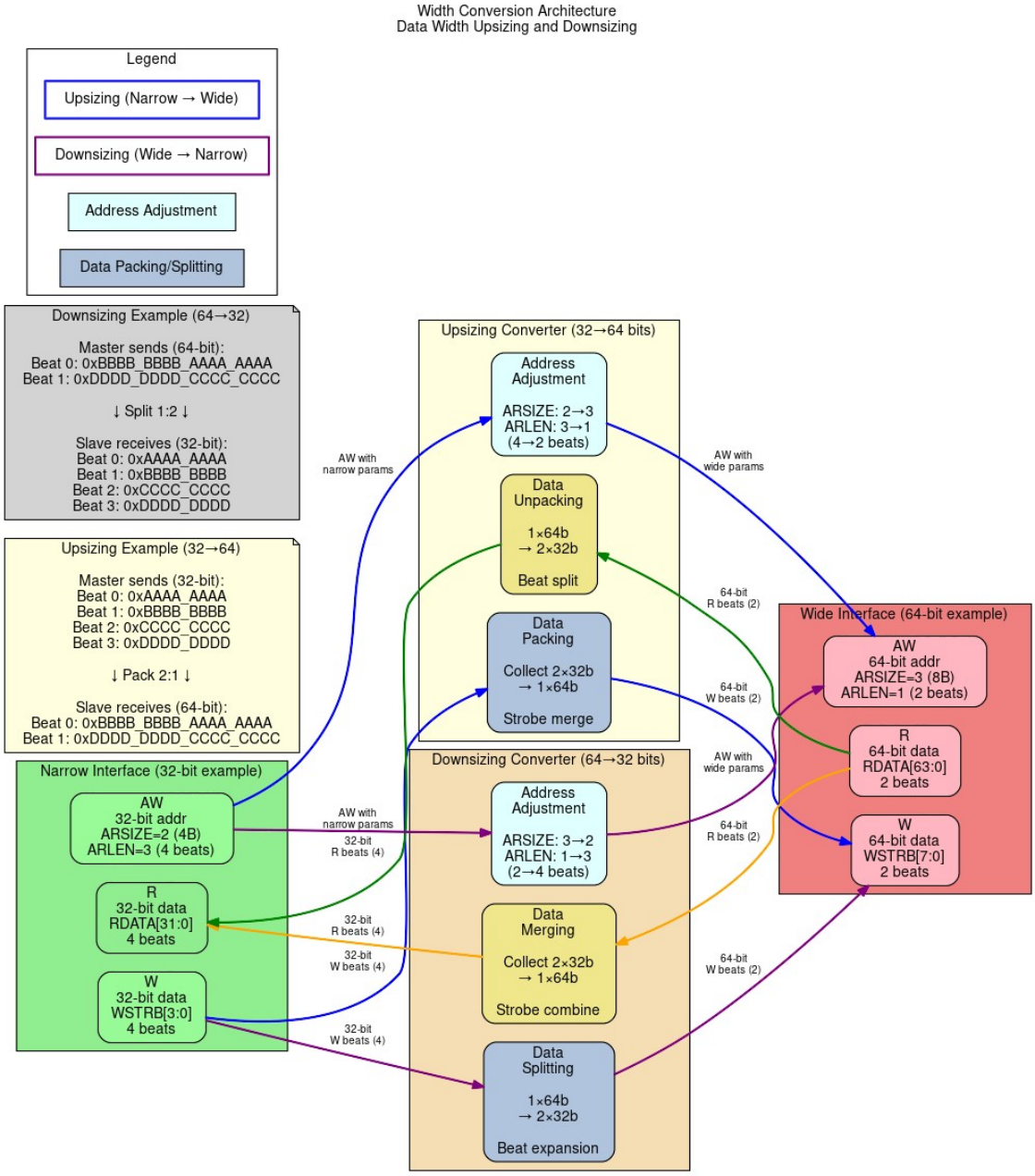
```
Rationale:
- Balance between resource usage and performance
- Common width for modern embedded processors
- Efficient for both 32-bit and 64-bit masters
- Reasonably sized multiplexers in crossbar
```

9.2.2 Width Conversion Locations

```
Master (32-bit) → [Upsizer] → Crossbar (64-bit) → [Downsizer] → Slave
(32-bit)
Master (64-bit) → [No conversion] → Crossbar (64-bit) → [No
conversion] → Slave (64-bit)
Master (128-bit) → [Downsizer] → Crossbar (64-bit) → [Upsizer] → Slave
(128-bit)
```

9.3 2.6.3 Block Diagram

9.3.1 Figure 2.6: Width Conversion Architecture



Width Conversion Architecture

Width conversion architecture showing data upsizing and downsizing with beat count adjustment and strobe handling.

9.4 2.6.4 Upsizing (Narrow to Wide)

9.4.1 Overview

Upsizing converts narrow data to wide data by buffering multiple narrow beats into a single wide beat.

```
Example: 32-bit master → 64-bit crossbar
  Master beat 0: 32'hAAAA_BBBB (bytes 3:0)
  Master beat 1: 32'hCCCC_DDDD (bytes 7:4)
  → Crossbar beat: 64'hCCCC_DDDD_AAAA_BBBB
```

9.4.2 Write Upsizing

Burst of 4 (32-bit) → Burst of 2 (64-bit):

```
Master AW: AWADDR = 0x1000, AWLEN = 3, AWSIZE = 2 (4 bytes)
Master W beats:
  W[0]: WDATA = 0xAAAA_BBBB, WSTRB = 4'b1111, WLAST = 0
  W[1]: WDATA = 0xCCCC_DDDD, WSTRB = 4'b1111, WLAST = 0
  W[2]: WDATA = 0xEEEE_FFFF, WSTRB = 4'b1111, WLAST = 0
  W[3]: WDATA = 0x1111_2222, WSTRB = 4'b1111, WLAST = 1

Crossbar AW: AWADDR = 0x1000, AWLEN = 1, AWSIZE = 3 (8 bytes)
Crossbar W beats:
  W[0]: WDATA = 0xCCCC_DDDD_AAAA_BBBB, WSTRB = 8'b1111_1111, WLAST = 0
  W[1]: WDATA = 0x1111_2222_EEEE_FFFF, WSTRB = 8'b1111_1111, WLAST = 1
```

9.4.3 Read Upsizing

Burst of 8 (32-bit) → Burst of 4 (64-bit):

```
Master AR: ARADDR = 0x2000, ARLEN = 7, ARSIZE = 2 (4 bytes)

Crossbar AR: ARADDR = 0x2000, ARLEN = 3, ARSIZE = 3 (8 bytes)

Crossbar R beats:
  R[0]: RDATA = 0x1111_2222_3333_4444, RLAST = 0
  R[1]: RDATA = 0x5555_6666_7777_8888, RLAST = 0
  R[2]: RDATA = 0x9999_AAAA_BBBB_CCCC, RLAST = 0
  R[3]: RDATA = 0xDDDD_EEEE_FFFF_0000, RLAST = 1

Master R beats (split from crossbar):
  R[0]: RDATA = 0x3333_4444, RLAST = 0  (from R[0] low)
  R[1]: RDATA = 0x1111_2222, RLAST = 0  (from R[0] high)
  R[2]: RDATA = 0x7777_8888, RLAST = 0  (from R[1] low)
  R[3]: RDATA = 0x5555_6666, RLAST = 0  (from R[1] high)
  R[4]: RDATA = 0xBBBB_CCCC, RLAST = 0  (from R[2] low)
```

```
  R[5]: RDATA = 0x9999_AAAA, RLAST = 0  (from R[2] high)
  R[6]: RDATA = 0xFFFF_0000, RLAST = 0  (from R[3] low)
  R[7]: RDATA = 0xDDDD_EEEE, RLAST = 1  (from R[3] high)
```

9.4.4 Strobe Mapping (Upsizing)

```
32-bit WSTRB → 64-bit WSTRB:

Beat 0 (lower 32 bits):
  32-bit WSTRB = 4'b1010 → 64-bit WSTRB = 8'b0000_1010

Beat 1 (upper 32 bits):
  32-bit WSTRB = 4'b1111 → 64-bit WSTRB = 8'b1111_0000

Combined:
  64-bit WSTRB = 8'b1111_1010
```

9.5 2.6.5 Downsizing (Wide to Narrow)

9.5.1 Overview

Downsizing converts wide data to narrow data by splitting a single wide beat into multiple narrow beats.

```
Example: 128-bit master → 64-bit crossbar
  Master beat: 128'h1111_2222_3333_4444_5555_6666_7777_8888
  → Crossbar beat 0: 64'h5555_6666_7777_8888 (lower)
  → Crossbar beat 1: 64'h1111_2222_3333_4444 (upper)
```

9.5.2 Write Downsizing

Burst of 2 (128-bit) → Burst of 4 (64-bit):

```
Master AW: AWADDR = 0x3000, AWLEN = 1, AWSIZE = 4 (16 bytes)
Master W beats:
  W[0]: WDATA = 0xAAAA...(128 bits), WSTRB = 16'hFFFF, WLAST = 0
  W[1]: WDATA = 0xBBBB...(128 bits), WSTRB = 16'hFFFF, WLAST = 1

Crossbar AW: AWADDR = 0x3000, AWLEN = 3, AWSIZE = 3 (8 bytes)
Crossbar W beats:
  W[0]: WDATA = 0xAAAA_low(64), WSTRB = 8'hFF, WLAST = 0
  W[1]: WDATA = 0xAAAA_high(64), WSTRB = 8'hFF, WLAST = 0
  W[2]: WDATA = 0xBBBB_low(64), WSTRB = 8'hFF, WLAST = 0
  W[3]: WDATA = 0xBBBB_high(64), WSTRB = 8'hFF, WLAST = 1
```

9.5.3 Read Downsizing

Burst of 1 (128-bit) → Burst of 2 (64-bit):

```
Master AR: ARADDR = 0x4000, ARLEN = 0, ARSIZE = 4 (16 bytes)

Crossbar AR: ARADDR = 0x4000, ARLEN = 1, ARSIZE = 3 (8 bytes)

Crossbar R beats:
  R[0]: RDATA = 0x1111_2222_3333_4444, RLAST = 0
  R[1]: RDATA = 0x5555_6666_7777_8888, RLAST = 1

Master R beat (merged):
  R[0]: RDATA = 0x5555_6666_7777_8888_1111_2222_3333_4444, RLAST = 1
```

9.5.4 Strobe Mapping (Downsizing)

```
128-bit WSTRB → 64-bit WSTRB (split):

128-bit WSTRB = 16'hF0F0_0FF0

Beat 0 (bytes 7:0):
  64-bit WSTRB = 8'b0000_1111_1111_0000 = 8'h0F0 → 8'hF0

Beat 1 (bytes 15:8):
  64-bit WSTRB = 8'b1111_0000_1111_0000 = 8'hF0F0 >> 8 → 8'hF0
```

9.6 2.6.6 Address Alignment

9.6.1 Address Adjustment for Width

When changing widths, addresses must align to the new width:

```
32-bit (4-byte) aligned address: 0x1004
Upsized to 64-bit (8-byte): 0x1000 (round down to 8-byte boundary)

Narrow access at 0x1004 within 64-bit word:
  - Byte offset = 0x1004 & 0x7 = 4
  - Data at bytes [7:4] of 64-bit word
  - Lower bytes [3:0] unused (WSTRB = 8'b1111_0000)
```

9.6.2 Unaligned Access Handling

```
Master: 32-bit, unaligned access at 0x1002
Problem: Not aligned to 4-byte boundary

Options:
1. Error response (strict mode)
2. Round down address, use WSTRB for actual bytes
3. Split into multiple aligned accesses
```

```
Bridge default: Option 2 (use WSTRB)
```

9.7 2.6.7 Burst Length Adjustment

9.7.1 Length Calculation

```
Formula:
  New_Length = (Old_Length + 1) × (Old_Width / New_Width) - 1

Example: Upsizing 32→64
  Old: AWLEN = 7 (8 beats), WIDTH = 32
  New: AWLEN = (7+1) × (32/64) - 1 = 8 × 0.5 - 1 = 3 (4 beats)

Example: Downsizing 128→64
  Old: ARLEN = 3 (4 beats), WIDTH = 128
  New: ARLEN = (3+1) × (128/64) - 1 = 4 × 2 - 1 = 7 (8 beats)
```

9.7.2 Odd Burst Lengths

When burst doesn't divide evenly:

```
Example: 3 beats of 32-bit → 64-bit
  3 beats × 4 bytes = 12 bytes total
  12 bytes / 8 bytes per beat = 1.5 beats

Solution:
  - Beat 0: Full 64-bit (8 bytes)
  - Beat 1: Partial 64-bit (4 bytes, WSTRB = 8'b0000_1111)
  - AWLEN = 1 (2 beats)
```

9.8 2.6.8 Resource Utilization

9.8.1 Per-Converter Resources

32→64 Upsizer:

```
Logic Elements:  ~300 LEs
Registers:       ~100 regs (buffering + FSM)
Block RAM:       0

Breakdown:
- Data buffer (32 bits): ~32 regs
- Strobe buffer: ~4 regs
- Beat counter: ~8 regs
- Control FSM: ~50 LEs, ~20 regs
- MUX logic: ~150 LEs
```

64→32 Downsizer:

```
Logic Elements:  ~250 LEs
Registers:       ~120 regs
Block RAM:       0

Similar structure but includes split logic
```

128→64 Downsizer:

```
Logic Elements:  ~400 LEs
Registers:       ~180 regs
Block RAM:       0

Larger data paths, more complex MUX
```

9.8.2 Scaling

Resource usage scales primarily with: - **Width ratio**: 2:1 vs. 4:1 vs. 8:1 conversion - **Data width**: 128-bit vs. 256-bit buffers - **Buffering depth**: Single vs. multi-beat buffering

9.9 2.6.9 Timing Characteristics

9.9.1 Latency

Upsizing (combining beats): - Buffering latency: 1-2 cycles (wait for N narrow beats) - First beat output: After receiving required narrow beats - Example: 32→64 requires 2 master beats before 1 crossbar beat

Downsizing (splitting beats): - Minimal latency: 1 cycle (register stage) - First beat output: Immediately (split from wide beat) - Example: 128→64 outputs first 64-bit beat immediately

9.9.2 Throughput

Upsizing Throughput:

```
32→64 upsizing:
  Master: 32 bits/cycle = 4 bytes/cycle
  Crossbar: 64 bits per 2 cycles = 4 bytes/cycle (same)
  No throughput loss
```

Downsizing Throughput:

```
128→64 downsizing:
  Master: 128 bits/cycle = 16 bytes/cycle
  Crossbar: 64 bits/cycle = 8 bytes/cycle
  Throughput halved (crossbar becomes bottleneck)
```

9.10 2.6.10 Configuration Parameters

9.10.1 Width Conversion Configuration (TOML)

```toml
[bridge]
internal_data_width = 64        # Crossbar width
enable_width_conversion = true  # Allow width differences

[[masters]]
name = "cpu_32bit"
data_width = 32                 # Narrower than crossbar (upsizing)
addr_width = 32

[[masters]]
name = "dma_64bit"
data_width = 64                 # Matches crossbar (no conversion)
addr_width = 32

[[masters]]
name = "gpu_128bit"
data_width = 128                # Wider than crossbar (downsizing)
addr_width = 36

[[slaves]]
name = "memory_64bit"
data_width = 64                 # Matches crossbar (no conversion)

[[slaves]]
name = "periph_32bit"
data_width = 32                 # Narrower than crossbar (downsizing)
```

9.11 2.6.11 Debug and Observability

9.11.1 Recommended Debug Signals

```
Upsizer:
- Beat accumulator (partial wide beat being assembled)
- Beat counter (position in burst)
- Buffer valid (accumulated beats ready)
- Strobe accumulation

Downsizer:
- Beat splitter state (which sub-beat currently outputting)
- Remaining beats to output
- Source data register
```

9.11.2 Common Issues and Debug

Symptom: Data corruption after width conversion
Check: - Byte ordering (endianness) - Strobe mapping (correct bytes enabled) - Address alignment - Burst length calculation

Symptom: Stalls after width conversion
Check: - Buffer depths (upsizer needs buffering) - Backpressure propagation - LAST signaling

Symptom: Incorrect burst lengths
Check: - Length calculation formula - Odd burst handling - SIZE field matching width

9.12 2.6.12 Verification Considerations

9.12.1 Test Scenarios

1. **Power-of-2 Width Ratios**:

```
- 32→64 (2:1)
- 64→128 (1:2)
- 32→128 (4:1)
```

2. **Various Burst Lengths**:

```
- Single beat (ALEN=0)
- Even bursts (ALEN=3, 7, 15)
- Odd bursts (ALEN=1, 5, 9)
```

3. **Sub-Word Accesses**:

```
- Byte writes (WSTRB with individual bytes)
- Half-word, word accesses
- Unaligned accesses
```

4. **Mixed Widths**:

```
- 32-bit master → 64-bit crossbar → 32-bit slave
- 128-bit master → 64-bit crossbar → 32-bit slave
- All width combinations
```

9.13 2.6.13 Performance Considerations

9.13.1 Conversion Overhead

Upsizing Overhead: - Buffering latency: +1-2 cycles - Throughput maintained - Best for burst-oriented masters

Downsizing Overhead: - Split latency: +1 cycle - Throughput reduced by width ratio - Can bottleneck wide masters

9.13.2 Optimization Strategies

1. **Match Common Widths**: Design masters/slaves to match internal width (64-bit)
2. **Burst Sizing**: Use appropriate burst lengths for width ratios
3. **Parallel Paths**: Multiple 32-bit masters can aggregate to 64-bit crossbar bandwidth
4. **Selective Conversion**: Only convert where necessary

9.14 2.6.14 Future Enhancements

9.14.1 Planned Features

- **Configurable Internal Width**: Support 32, 64, 128, 256-bit crossbar
- **Multi-Beat Buffering**: Deeper upsizing buffers for smoother flow
- **Byte Rotation**: Handle unaligned access more efficiently
- **Pipelined Conversion**: Multi-stage for high frequency

9.14.2 Under Consideration

- **Asymmetric Widths**: Different widths for read vs. write paths
- **Sparse Data Optimization**: Skip unused bytes in conversions
- **Tagged Data**: Metadata preservation through width conversion
- **Zero-Copy Bypass**: Direct routing for matching widths

Related Sections: - Section 2.1: Master Adapter (upsizing location) - Section 2.3: Crossbar Core (internal data width) - Section 3.2: Master Port Interface (width specifications) - Section 3.3: Slave Port Interface (width specifications)

10 2.7 Protocol Conversion

Protocol Conversion enables the bridge to interface with slaves using different bus protocols. While the bridge internally uses AXI4, it can convert to simpler protocols like APB (Advanced Peripheral Bus) for low-bandwidth peripheral access.

10.1 2.7.1 Purpose and Function

Protocol conversion performs the following critical functions:

1. **Protocol Translation**: Converts AXI4 transactions to target protocol (e.g., APB)

2. **Handshake Mapping**: Translates ready/valid to protocol-specific handshakes
3. **Burst Decomposition**: Breaks AXI bursts into single-beat target transactions
4. **Response Mapping**: Converts protocol-specific responses back to AXI responses
5. **Timing Adaptation**: Handles different timing requirements between protocols

10.2 2.7.2 Supported Protocols

10.2.1 Current Support (Phase 2)

AXI4 (Native): - Full AXI4 protocol - No conversion required - Maximum performance

AXI4-Lite (Master-Side): - Simplified AXI4 subset - Single-beat transactions only (ARLEN=0, AWLEN=0) - Converts to full AXI4 for crossbar - Common for control/status registers

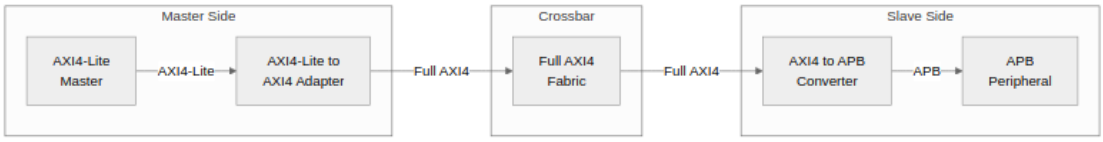
APB (Slave-Side): - APB3 and APB4 support - For low-bandwidth peripherals - Simplified handshaking

10.2.2 Future Support (Phase 2+)

- **AHB**: Advanced High-performance Bus
- **Wishbone**: Open-source bus standard
- **Custom**: User-defined protocols

10.3 2.7.3 Conversion Architecture Overview

10.3.1 Figure 2.7.1: Protocol Conversion Architecture

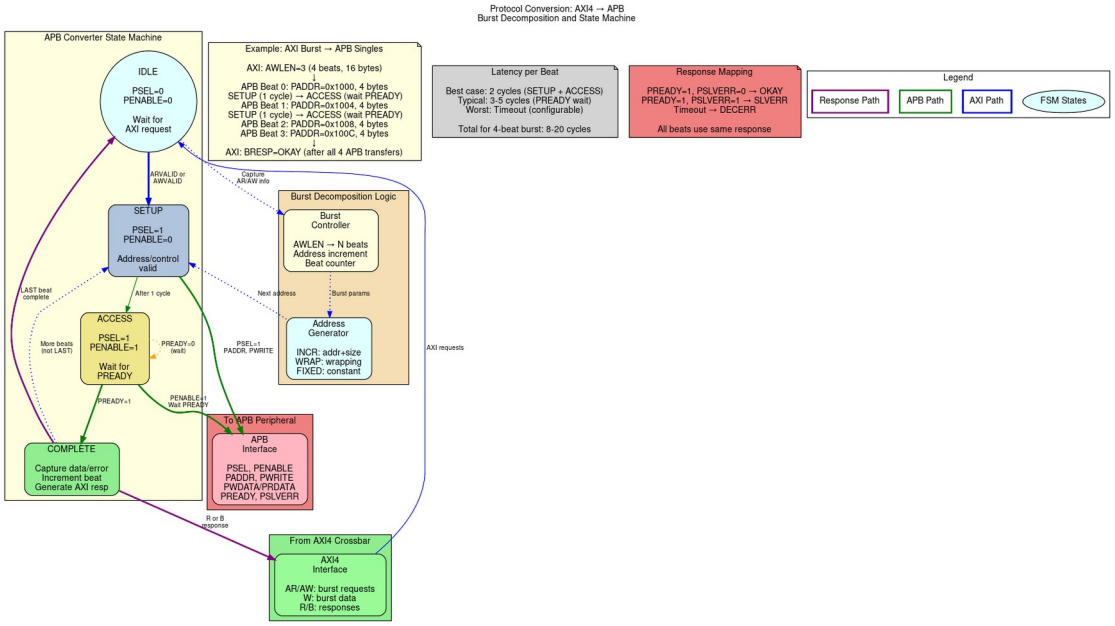


Protocol Conversion Architecture

The diagram shows the complete protocol conversion flow: AXI4-Lite masters are adapted to full AXI4 before entering the crossbar, while APB peripherals receive AXI4 transactions through a dedicated converter.

10.4 2.7.4 Block Diagram

10.4.1 Figure 2.7: Protocol Conversion Architecture



Protocol Conversion Architecture

Protocol conversion showing AXI4 to APB conversion with state machine, burst decomposition, and response mapping.

10.5 2.7.5 AXI4-Lite to AXI4 Conversion (Master-Side)

10.5.1 AXI4-Lite Protocol Overview

AXI4-Lite is a simplified subset of AXI4 designed for simple control/status register access:

```
Key Differences from Full AXI4:
- FIXED burst length: ARLEN/AWLEN always = 0 (single beat)
- FIXED burst size: No SIZE field, always full data width
- FIXED burst type: No BURST field, always INCR
- NO exclusive access: No LOCK support
- NO unaligned transfers: Address must be aligned
- Simpler ID: Typically 1-4 bits (fewer outstanding transactions)

Similarities to AXI4:
- Same 5 channels: AR, R, AW, W, B
- Same valid/ready handshaking
- Same response codes: OKAY, SLVERR, DECERR, EXOKAY
- Same data widths: 32 or 64 bits typically
```

10.5.2 Conversion Requirements

To adapt AXI4-Lite masters to the full AXI4 crossbar, the adapter must:

1. **Add Missing Signals**: Provide default values for burst-related signals
2. **Validate Constraints**: Ensure single-beat assumption holds
3. **Pass-Through Simplicity**: Most signals connect directly

10.5.3 Signal Mapping

```
// AXI4-Lite to AXI4 Signal Mapping

// AR Channel (Read Address)
// AXI4-Lite Input          AXI4 Crossbar Output
axi4lite_arvalid      →     axi4_arvalid
axi4lite_arready      ←     axi4_arready
axi4lite_araddr       →     axi4_araddr
axi4lite_arprot       →     axi4_arprot
axi4lite_arid (opt)   →     axi4_arid

// Added by adapter (constants):
                            axi4_arlen    = 8'h00      // Always 1
beat
                            axi4_arsize   = log2(DW/8) // Full width
                            axi4_arburst  = 2'b01      // INCR
                            axi4_arlock   = 1'b0       // No lock
                            axi4_arcache  = 4'b0000    // Device non-
buf
                            axi4_arqos    = 4'h0       // No QoS
                            axi4_arregion = 4'h0       // Region 0

// R Channel (Read Data)
// AXI4 Crossbar Input      AXI4-Lite Output
axi4_rvalid           →     axi4lite_rvalid
axi4_rready           ←     axi4lite_rready
axi4_rdata            →     axi4lite_rdata
axi4_rresp            →     axi4lite_rresp
axi4_rid (opt)        →     axi4lite_rid (opt)

// Discarded by adapter:
axi4_rlast                  // Always 1 for single beat

// AW Channel (Write Address)
axi4lite_awvalid      →     axi4_awvalid
axi4lite_awready      ←     axi4_awready
axi4lite_awaddr       →     axi4_awaddr
axi4lite_awprot       →     axi4_awprot
```

```
axi4lite_awid (opt)   →     axi4_awid

// Added by adapter:
                            axi4_awlen    = 8'h00
                            axi4_awsize   = log2(DW/8)
                            axi4_awburst  = 2'b01
                            axi4_awlock   = 1'b0
                            axi4_awcache  = 4'b0000
                            axi4_awqos    = 4'h0
                            axi4_awregion = 4'h0

// W Channel (Write Data)
axi4lite_wvalid       →     axi4_wvalid
axi4lite_wready       ←     axi4_wready
axi4lite_wdata        →     axi4_wdata
axi4lite_wstrb        →     axi4_wstrb

// Added by adapter:
                            axi4_wlast    = 1'b1       // Always last

// B Channel (Write Response)
axi4_bvalid           →     axi4lite_bvalid
axi4_bready           ←     axi4lite_bready
axi4_bresp            →     axi4lite_bresp
axi4_bid (opt)        →     axi4lite_bid (opt)
```

10.5.4 Implementation

The AXI4-Lite adapter is extremely simple, primarily providing constant values:

```
// AXI4-Lite to AXI4 Adapter (simplified)
module axi4lite_to_axi4_adapter #(
    parameter ADDR_WIDTH = 32,
    parameter DATA_WIDTH = 64,
    parameter ID_WIDTH = 4         // Optional, often 0 for AXI4-Lite
) (
    input logic clk,
    input logic rst_n,

    // AXI4-Lite Master Interface
    input  logic                    lite_arvalid,
    output logic                    lite_arready,
    input  logic [ADDR_WIDTH-1:0]   lite_araddr,
    input  logic [2:0]              lite_arprot,
    input  logic [ID_WIDTH-1:0]     lite_arid,    // Optional

    output logic                    lite_rvalid,
    input  logic                    lite_rready,
```

```
    output logic [DATA_WIDTH-1:0]    lite_rdata,
    output logic [1:0]               lite_rresp,
    output logic [ID_WIDTH-1:0]      lite_rid,     // Optional

    input  logic                     lite_awvalid,
    output logic                     lite_awready,
    input  logic [ADDR_WIDTH-1:0]    lite_awaddr,
    input  logic [2:0]               lite_awprot,
    input  logic [ID_WIDTH-1:0]      lite_awid,    // Optional

    input  logic                     lite_wvalid,
    output logic                     lite_wready,
    input  logic [DATA_WIDTH-1:0]    lite_wdata,
    input  logic [DATA_WIDTH/8-1:0]  lite_wstrb,

    output logic                     lite_bvalid,
    input  logic                     lite_bready,
    output logic [1:0]               lite_bresp,
    output logic [ID_WIDTH-1:0]      lite_bid,     // Optional

    // Full AXI4 Crossbar Interface
    output logic                     axi4_arvalid,
    input  logic                     axi4_arready,
    output logic [ADDR_WIDTH-1:0]    axi4_araddr,
    output logic [7:0]               axi4_arlen,
    output logic [2:0]               axi4_arsize,
    output logic [1:0]               axi4_arburst,
    output logic                     axi4_arlock,
    output logic [3:0]               axi4_arcache,
    output logic [2:0]               axi4_arprot,
    output logic [3:0]               axi4_arqos,
    output logic [3:0]               axi4_arregion,
    output logic [ID_WIDTH-1:0]      axi4_arid,

    input  logic                     axi4_rvalid,
    output logic                     axi4_rready,
    input  logic [DATA_WIDTH-1:0]    axi4_rdata,
    input  logic [1:0]               axi4_rresp,
    input  logic                     axi4_rlast,
    input  logic [ID_WIDTH-1:0]      axi4_rid,

    // ... (AW, W, B channels similar)
);

    // AR Channel: Pass-through with constants
    assign axi4_arvalid  = lite_arvalid;
    assign lite_arready  = axi4_arready;
    assign axi4_araddr   = lite_araddr;
```

```
    assign axi4_arprot   = lite_arprot;
    assign axi4_arid     = lite_arid;

    // Constants for single-beat burst
    assign axi4_arlen    = 8'h00;                    // 1 beat
    assign axi4_arsize   = $clog2(DATA_WIDTH/8);     // Full width
    assign axi4_arburst  = 2'b01;                    // INCR
    assign axi4_arlock   = 1'b0;                     // No lock
    assign axi4_arcache  = 4'b0000;                  // Device non-buf
    assign axi4_arqos    = 4'h0;                     // No QoS
    assign axi4_arregion = 4'h0;                     // Region 0

    // R Channel: Pass-through, ignore rlast
    assign lite_rvalid = axi4_rvalid;
    assign axi4_rready = lite_rready;
    assign lite_rdata  = axi4_rdata;
    assign lite_rresp  = axi4_rresp;
    assign lite_rid    = axi4_rid;
    // axi4_rlast ignored (always 1 for single beat)

    // AW Channel: Similar to AR
    assign axi4_awvalid  = lite_awvalid;
    assign lite_awready  = axi4_awready;
    assign axi4_awaddr   = lite_awaddr;
    assign axi4_awprot   = lite_awprot;
    assign axi4_awid     = lite_awid;
    assign axi4_awlen    = 8'h00;
    assign axi4_awsize   = $clog2(DATA_WIDTH/8);
    assign axi4_awburst  = 2'b01;
    assign axi4_awlock   = 1'b0;
    assign axi4_awcache  = 4'b0000;
    assign axi4_awqos    = 4'h0;
    assign axi4_awregion = 4'h0;

    // W Channel: Pass-through, add wlast
    assign axi4_wvalid = lite_wvalid;
    assign lite_wready = axi4_wready;
    assign axi4_wdata  = lite_wdata;
    assign axi4_wstrb  = lite_wstrb;
    assign axi4_wlast  = 1'b1;                       // Always last

    // B Channel: Pass-through
    assign lite_bvalid = axi4_bvalid;
    assign axi4_bready = lite_bready;
    assign lite_bresp  = axi4_bresp;
    assign lite_bid    = axi4_bid;

endmodule
```

10.5.5 Resource Utilization

AXI4-Lite Adapter Resources:

```
Logic Elements:  ~50-100 LEs (minimal, mostly wiring)
Registers:       ~50 regs (if skid buffers added)
Block RAM:       0

Breakdown:
- Signal pass-through: ~20 LEs (buffering)
- Constant generation: ~10 LEs
- Optional skid buffers: ~50 regs (for timing)

Note: Most implementations are purely combinatorial wire
      assignments with optional pipeline registers.
```

10.5.6 Performance Impact

Latency: - **Zero-latency** (combinatorial) if no pipeline stages - **1-2 cycles** if skid buffers added for timing - No protocol conversion overhead

Throughput: - **1 transaction per cycle** (same as native AXI4) - No degradation for single-beat transactions - Limited by AXI4-Lite's single-beat constraint

10.5.7 Configuration

```toml
[[masters]]
name = "control_processor"
protocol = "axi4lite"          # Specify simplified protocol
channels = "rw"                # Full read-write
arid_width = 0                 # Often no ID in AXI4-Lite
awid_width = 0
addr_width = 32
data_width = 32                # Typically 32 or 64 bits
```

10.5.8 Common Issues and Debug

Issue 1: Burst Detected on AXI4-Lite

```
Symptom: ARLEN/AWLEN != 0 on AXI4-Lite interface
Cause: Master not properly configured as AXI4-Lite
Check: Verify master only issues single-beat transactions
```

Issue 2: Unaligned Addresses

```
Symptom: ARADDR/AWADDR not aligned to data width
Cause: AXI4-Lite requires full-width aligned access
Check: Address[log2(DW/8)-1:0] should be zero
```

Issue 3: rlast/wlast Handling

```
Symptom: Master expects rlast/wlast but doesn't have them
Cause: True AXI4-Lite interface omits these signals
Solution: Adapter provides wlast=1 to crossbar, strips rlast
```

10.6 2.7.6 AXI4 to APB Conversion (Slave-Side)

10.6.1 APB Protocol Overview

APB is a simple, low-power bus protocol:

```
Characteristics:
- Single address phase
- Single data phase
- No burst support (one transfer per operation)
- Minimal logic
- Low power consumption
- Suitable for peripherals: UARTs, timers, GPIOs
```

10.6.2 APB Signals

```
Address Phase:
  PADDR[N-1:0]  - Address bus
  PSEL          - Slave select
  PENABLE       - Enable (2nd cycle of transfer)
  PWRITE        - Write direction (1=write, 0=read)

Data Phase (Write):
  PWDATA[N-1:0] - Write data
  PSTRB[N/8-1:0]- Write strobes (APB4 only)

Data Phase (Read):
  PRDATA[N-1:0] - Read data

Response:
  PREADY        - Slave ready (can extend transfer)
  PSLVERR       - Slave error (APB3+)
```

10.6.3 APB State Machine

APB requires a 2-phase handshake:

```
IDLE:
  - Wait for AXI request (ARVALID or AWVALID)
  - PSEL = 0, PENABLE = 0

SETUP:
  - Assert PSEL = 1
  - Drive PADDR, PWRITE, PWDATA (if write)
  - PENABLE = 0
```

```
  - Duration: 1 cycle

ACCESS:
  - Assert PENABLE = 1
  - Wait for PREADY = 1
  - Capture PRDATA (if read) or PSLVERR
  - Can extend multiple cycles if PREADY = 0

Complete:
  - De-assert PSEL, PENABLE
  - Return to IDLE or SETUP (if more beats)
```

10.6.4 Read Transaction Conversion

AXI4 Read:

```
Cycle 0: ARVALID=1, ARADDR=0x100, ARLEN=3 (4 beats)
Cycle 1: ARREADY=1
Cycles 2-5: R beats returning
```

Converted to APB (4 separate APB reads):

```
Beat 0:
  Cycle 0: PSEL=1, PENABLE=0, PADDR=0x100, PWRITE=0 (SETUP)
  Cycle 1: PSEL=1, PENABLE=1, wait PREADY (ACCESS)
  Cycle 2: PREADY=1, capture PRDATA → First R beat

Beat 1:
  Cycle 3: PSEL=1, PENABLE=0, PADDR=0x104 (SETUP)
  Cycle 4: PSEL=1, PENABLE=1, wait PREADY (ACCESS)
  Cycle 5: PREADY=1, capture PRDATA → Second R beat

... (beats 2 and 3 similar)
```

Latency: 2-3 cycles per beat (SETUP + ACCESS + ready)

10.6.5 Write Transaction Conversion

AXI4 Write:

```
Cycle 0: AWVALID=1, AWADDR=0x200, AWLEN=1 (2 beats)
Cycle 1: AWREADY=1
Cycle 1: WVALID=1, WDATA=0xAAAA_BBBB, WLAST=0
Cycle 2: WREADY=1
Cycle 2: WVALID=1, WDATA=0xCCCC_DDDD, WLAST=1
Cycle 3: WREADY=1
Cycle 4: BVALID=1, BRESP=OKAY
```

Converted to APB (2 separate APB writes):

```
Beat 0:
  Cycle 0: PSEL=1, PENABLE=0, PADDR=0x200, PWRITE=1,
PWDATA=0xAAAA_BBBB
  Cycle 1: PSEL=1, PENABLE=1, wait PREADY
  Cycle 2: PREADY=1 → First write complete

Beat 1:
  Cycle 3: PSEL=1, PENABLE=0, PADDR=0x204, PWRITE=1,
PWDATA=0xCCCC_DDDD
  Cycle 4: PSEL=1, PENABLE=1, wait PREADY
  Cycle 5: PREADY=1 → Second write complete, return B
```

10.6.6 Burst Handling

APB does not support bursts, so:

```
AXI Burst: AWLEN = 15 (16 beats)
→ 16 separate APB transfers
→ Address increments per AWBURST type:
   - INCR: Addr += SIZE each beat
   - WRAP: Wrapping within boundary
   - FIXED: Same address each beat
```

10.6.7 Response Mapping

```
APB → AXI Response Translation:

PREADY=1, PSLVERR=0 → RRESP/BRESP = 2'b00 (OKAY)
PREADY=1, PSLVERR=1 → RRESP/BRESP = 2'b10 (SLVERR)

Timeout (PREADY stuck at 0):
  After N cycles → RRESP/BRESP = 2'b11 (DECERR)
```

10.7 2.7.5 Implementation

10.7.1 AXI4-to-APB Converter FSM

```
// Simplified AXI4-to-APB converter
typedef enum logic [2:0] {
    IDLE,
    AR_SETUP,
    AR_ACCESS,
    AW_SETUP,
    W_ACCESS,
    B_RESPONSE
} state_t;

state_t state, next_state;
```

```
always_ff @(posedge clk) begin
    if (!rst_n) state <= IDLE;
    else state <= next_state;
end

always_comb begin
    next_state = state;

    case (state)
        IDLE: begin
            if (arvalid) next_state = AR_SETUP;
            else if (awvalid) next_state = AW_SETUP;
        end

        AR_SETUP: begin
            next_state = AR_ACCESS;  // 1 cycle SETUP
        end

        AR_ACCESS: begin
            if (pready) begin
                if (more_beats) next_state = AR_SETUP;  // Next beat
                else next_state = IDLE;
            end
        end

        AW_SETUP: begin
            if (wvalid) next_state = W_ACCESS;
        end

        W_ACCESS: begin
            if (pready) begin
                if (!wlast) next_state = AW_SETUP;  // Next beat
                else next_state = B_RESPONSE;
            end
        end

        B_RESPONSE: begin
            if (bready) next_state = IDLE;
        end
    endcase
end

// APB signal generation
assign psel = (state != IDLE);
assign penable = (state == AR_ACCESS || state == W_ACCESS);
assign pwrite = (state == AW_SETUP || state == W_ACCESS);
```

10.7.2 Address Generation

```
// Address increment for burst
logic [ADDR_WIDTH-1:0] current_addr;
logic [7:0] beat_count;

always_ff @(posedge clk) begin
    if (state == IDLE) begin
        current_addr <= arvalid ? araddr : awaddr;
        beat_count <= 0;
    end else if ((state == AR_ACCESS || state == W_ACCESS) && pready)
begin
        beat_count <= beat_count + 1;

        case (burst_type)
            2'b01: current_addr <= current_addr + (1 << size);  //
INCR
            2'b10: current_addr <= wrap_address(current_addr);  //
WRAP
            2'b00: current_addr <= current_addr;                //
FIXED
        endcase
    end
end

assign paddr = current_addr;
```

10.8 2.7.6 Resource Utilization

10.8.1 APB Converter Resources

Per APB Slave Interface:

```
Logic Elements:  ~400 LEs
Registers:       ~150 regs
Block RAM:       0

Breakdown:
- FSM control:           ~100 LEs, ~20 regs
- Address generation:    ~80 LEs, ~40 regs
- Burst counter:         ~50 LEs, ~20 regs
- Response accumulation: ~80 LEs, ~30 regs
- Data path MUX:         ~90 LEs, ~40 regs
```

10.8.2 Scaling

Adding APB slaves: - Linear scaling: +~400 LEs per APB slave - Shared address decoder logic - Independent per-slave FSMs

10.9 2.7.7 Timing Characteristics

10.9.1 Latency

APB Read Latency (per beat):

```
Best case: 2 cycles (SETUP + ACCESS with PREADY=1)
Typical: 3-5 cycles (if slave extends with PREADY=0)
Worst case: Configurable timeout (e.g., 1000 cycles)

For 8-beat AXI burst:
  Total: 8 × 3 = 24 cycles typical
```

APB Write Latency (per beat):

```
Similar to read: 2-5 cycles per beat
```

10.9.2 Throughput

Severely Limited:

```
APB: ~0.3-0.5 transactions/cycle (due to 2-3 cycle protocol)
AXI4: 1 transaction/cycle (burst mode)

APB suitable only for low-bandwidth peripherals
```

10.10 2.7.8 Configuration Parameters

10.10.1 Protocol Conversion Configuration (TOML)

```toml
[[slaves]]
name = "uart_peripheral"
protocol = "apb"              # "axi4", "apb", "ahb" (future)
base_address = 0xF000_0000
size = 0x1000
data_width = 32
apb_timeout = 1000            # Cycles before timeout error

[[slaves]]
name = "ddr_memory"
protocol = "axi4"             # Native, no conversion
base_address = 0x8000_0000
size = 0x4000_0000
data_width = 64
```

10.11 2.7.9 Debug and Observability

10.11.1 Recommended Debug Signals

```
APB Converter:
- FSM state
- APB phase (SETUP, ACCESS)
- Beat counter (progress through burst)
- PREADY timeout counter
- Response accumulation (for burst)

APB Bus:
- PSEL, PENABLE, PWRITE
- PADDR, PWDATA, PRDATA
- PREADY, PSLVERR
```

10.11.2 Common Issues and Debug

Symptom: APB slave not responding (timeout)
Check: - PREADY signal (stuck at 0?) - APB slave clock/reset - PSEL assertion - Timeout threshold

Symptom: Data corruption on APB
Check: - SETUP phase duration (should be 1 cycle) - PENABLE assertion timing - Data sampling on correct cycle

Symptom: Burst to APB takes too long
Check: - Burst length (consider limiting ARLEN/AWLEN) - APB slave response time (PREADY) - Alternative: Use AXI4 slave instead

10.12 2.7.10 Verification Considerations

10.12.1 Test Scenarios

1. **Single APB Transfer**:

```
- AXI ARLEN=0 (1 beat) → 1 APB read
- Verify SETUP → ACCESS sequence
- Check PREADY handling
```

2. **APB Burst Decomposition**:

```
- AXI AWLEN=7 (8 beats) → 8 APB writes
- Verify address increment
- Check each beat completes before next
```

3. **APB PREADY Extension**:

```
- Slave holds PREADY=0 for N cycles
- Verify converter waits
- Check no data corruption
```

4. **APB Error Response**:

```
- Slave asserts PSLVERR
- Verify mapped to AXI SLVERR
- Check error propagated to master
```

5. **APB Timeout**:

```
- Slave never asserts PREADY
- Verify timeout after N cycles
- Check DECERR response
```

10.13 2.7.11 Performance Considerations

10.13.1 When to Use APB

Good Use Cases: - Low-speed peripherals (UART, GPIO, timers) - Infrequent accesses - Simple register interfaces - Power-sensitive designs

Poor Use Cases: - High-bandwidth devices - Burst-intensive masters - Performance-critical paths - Memory interfaces

10.13.2 APB vs. AXI4 Comparison

Feature	APB	AXI4
Complexity	Simple	Complex
Throughput	Low (~0.3/cyc)	High (1/cyc burst)
Latency/beat	2-5 cycles	1 cycle
Burst Support	No	Yes (up to 256)
Resources	~400 LEs	Native (no converter)
Power	Very low	Moderate
Use Case	Peripherals	Memory, DMA

10.14 2.7.12 Mixed Protocol Bridges

10.14.1 Example Configuration

```
# Bridge with mixed protocols
[bridge]
num_masters = 2
num_slaves = 3

[[masters]]
name = "cpu"
protocol = "axi4"

[[masters]]
name = "dma"
protocol = "axi4"
```

```toml
[[slaves]]
name = "ddr_memory"
protocol = "axi4"          # High bandwidth

[[slaves]]
name = "sram"
protocol = "axi4"          # Medium bandwidth

[[slaves]]
name = "peripherals"
protocol = "apb"           # Low bandwidth, simple
```

10.14.2 Routing Optimization

```
CPU → DDR Memory: AXI4-to-AXI4 (native, fast)
CPU → Peripherals: AXI4-to-APB (converted, slower)
DMA → SRAM: AXI4-to-AXI4 (native, fast)
DMA → Peripherals: AXI4-to-APB (rare, acceptable slowdown)
```

10.15 2.7.13 Master-Side vs Slave-Side Conversion

10.15.1 Comparison

Feature	Master-Side (AXI4-Lite)	Slave-Side (APB)
Complexity	Very Simple	Complex
Resource Usage	~50-100 LEs	~400 LEs
Latency Added	0-1 cycles	2-5 cycles/beat
Throughput Impact	None	Severe (3x slower)
Burst Handling	Single beat only	Decompose to singles
State Machine	None (combinatorial)	Multi-state FSM
Buffering Required	Optional (timing)	Essential
Use Case	Control registers	Peripherals

10.15.2 When to Use Each

AXI4-Lite Master Adapter: - Simple control/status register interfaces - Low-complexity masters (MCUs, simple CPUs) - Minimal resource overhead acceptable - No burst performance needed

APB Slave Converter: - Legacy peripheral integration - Very simple slave devices (GPIO, timers) - Low-bandwidth acceptable - Power optimization critical

10.16 2.7.14 Future Protocol Support

10.16.1 Planned Features

AXI4-Lite: - Subset of AXI4 - No burst support (ALEN=0 always) - Simpler than full AXI4 - Common for control registers

AHB (AMBA High-performance Bus): - More capable than APB - Pipeline support - Burst support - Suitable for moderate-bandwidth peripherals

Wishbone: - Open-source bus standard - Common in FPGA designs - Multiple addressing modes - Configurable data widths

10.16.2 Under Consideration

- **Custom Protocol**: User-defined through configuration
- **Stream Interface**: AXI4-Stream for data streaming
- **PCIe TLP**: For PCIe endpoint integration
- **CHI**: ARM's Coherent Hub Interface

10.17 2.7.14 Best Practices

10.17.1 Design Recommendations

1. **Limit APB Burst Lengths**: Configure masters to use short bursts to APB slaves
2. **Proper Timeouts**: Set realistic timeout values for APB slaves
3. **Protocol Matching**: Use native AXI4 where possible, APB only when necessary
4. **Address Map Planning**: Group APB peripherals together for efficient decoding
5. **Width Matching**: Match APB data width to peripheral requirements

10.17.2 Performance Tips

```
Inefficient: 256-beat AXI burst → APB
  256 beats × 3 cyc/beat = 768 cycles

Better: Limit to 4-beat bursts → APB
  64 bursts of 4 beats each
  Still long, but more manageable

Best: Use AXI4-Lite slave for registers
  No burst, but native protocol
```

Related Sections: - Section 2.3: Crossbar Core (protocol integration point) - Section 3.3: Slave Port Interface (APB signal specifications) - Chapter 4: Programming (configuring protocol conversion) - Appendix A: Generator Deep Dive (protocol converter generation)

11 2.8 Response Routing

Response Routing is the mechanism by which slave responses (read data and write acknowledgments) are directed back to the originating master. This system uses Bridge IDs, demultiplexing logic, and optional CAM structures to ensure responses reach the correct destination.

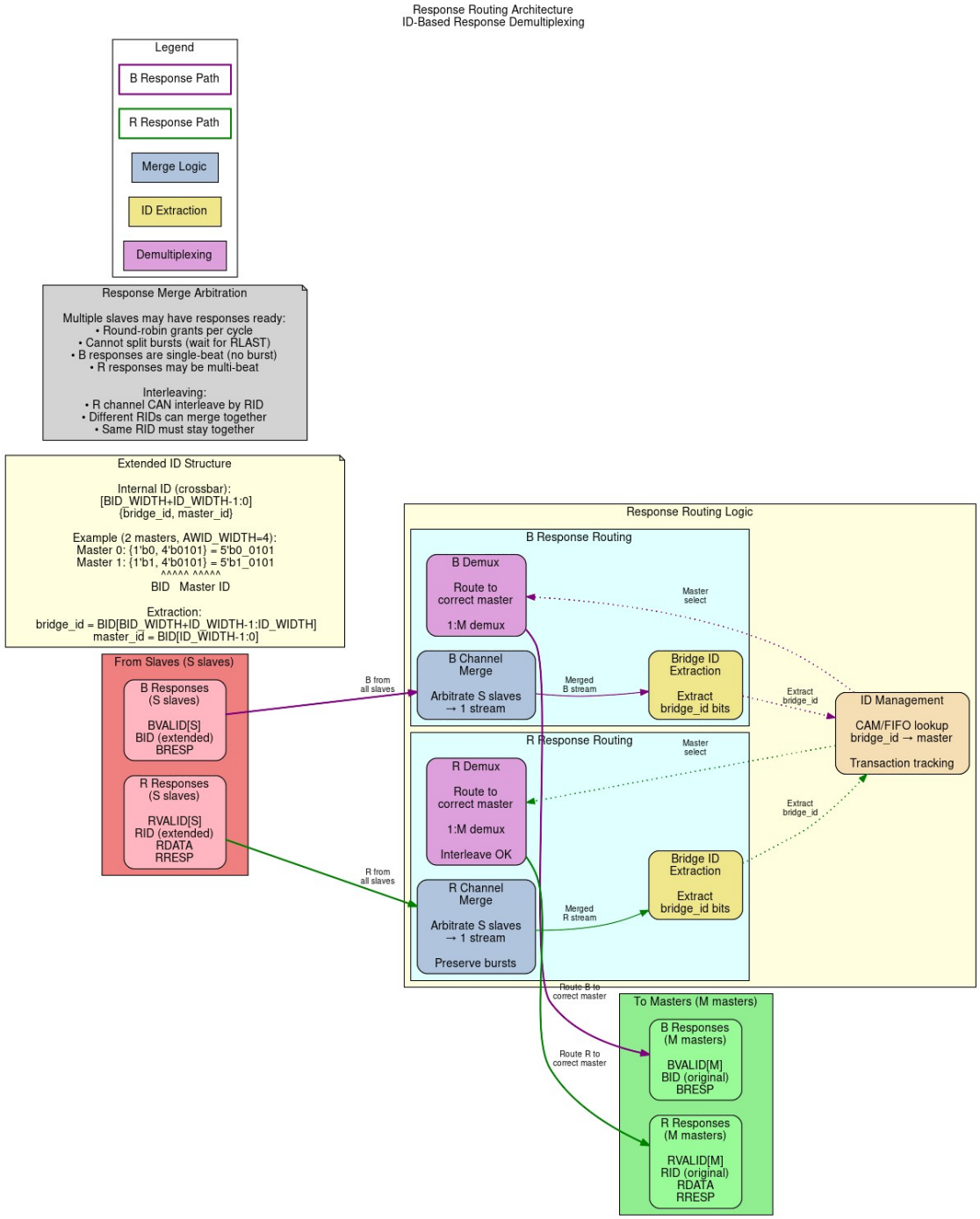
11.1 2.8.1 Purpose and Function

Response routing performs the following critical functions:

1. **Response Direction**: Routes R and B channel responses to correct master
2. **ID-Based Routing**: Uses extracted Bridge IDs to determine destination
3. **Multi-Slave Merging**: Combines responses from multiple slaves to single master
4. **Flow Control**: Manages backpressure from master on response channels
5. **Error Propagation**: Ensures error responses reach originating master

11.2 2.8.2 Block Diagram

11.2.1 Figure 2.8: Response Routing Architecture



Response Routing Architecture

Response routing architecture showing B and R channel merging from slaves and ID-based demultiplexing to masters.

11.3 2.8.3 BID Extraction

11.3.1 Simple Extraction (No CAM)

For configurations where direct BID mapping suffices:

```
// Extract Bridge ID from response ID
logic [TOTAL_RID_WIDTH-1:0] slave_rid;     // From slave
logic [BID_WIDTH-1:0] extracted_bid;       // Master index
logic [RID_WIDTH-1:0] external_rid;        // To master

// Upper bits = Bridge ID, lower bits = original ID
assign extracted_bid = slave_rid[TOTAL_RID_WIDTH-1:RID_WIDTH];
assign external_rid = slave_rid[RID_WIDTH-1:0];

// Route response based on BID
logic [NUM_MASTERS-1:0] master_rvalid;
always_comb begin
    master_rvalid = '0;  // Default: no master selected
    master_rvalid[extracted_bid] = slave_rvalid;
end
```

11.3.2 CAM-Based Extraction

For complex configurations with OOO or ID reordering:

```
// CAM lookup for response routing
logic [TOTAL_RID_WIDTH-1:0] response_id;
logic [BID_WIDTH-1:0] master_index;
logic cam_hit;

// Search CAM for matching transaction
assign {cam_hit, master_index} = cam_lookup(response_id);

// Route response to found master
if (cam_hit) begin
    master_rvalid[master_index] = slave_rvalid;
end else begin
    // Error: No matching transaction (protocol violation)
    error_response();
end
```

11.4 2.8.4 Response Demultiplexing

11.4.1 Per-Master Response Paths

Each master has dedicated response channels from the demultiplexer:

```
Master 0 Response:
  - M0_RVALID, M0_RREADY, M0_RDATA, M0_RID, M0_RRESP, M0_RLAST
  - M0_BVALID, M0_BREADY, M0_BID, M0_BRESP

Source: Responses from any slave with BID=0
```

11.4.2 Demux Implementation

```
// Response demultiplexer (4 masters, 3 slaves)
// Combines responses from all slaves, routes to correct master

// Slave responses (multiple sources)
logic s0_rvalid, s1_rvalid, s2_rvalid;
logic [63:0] s0_rdata, s1_rdata, s2_rdata;
logic [5:0] s0_rid, s1_rid, s2_rid;  // Includes BID

// Master responses (multiple destinations)
logic m0_rvalid, m1_rvalid, m2_rvalid, m3_rvalid;
logic [63:0] m0_rdata, m1_rdata, m2_rdata, m3_rdata;
logic [3:0] m0_rid, m1_rid, m2_rid, m3_rid;  // BID stripped

// Demux logic per slave
for (genvar s = 0; s < 3; s++) begin
    logic [1:0] bid;
    assign bid = slave_rid[s][5:4];  // Extract BID

    always_comb begin
        case (bid)
            2'b00: begin
                if (slave_rvalid[s] && !m0_busy) begin
                    m0_rvalid = 1'b1;
                    m0_rdata = slave_rdata[s];
                    m0_rid = slave_rid[s][3:0];  // Strip BID
                end
            end
            2'b01: /* Route to M1 */
            2'b10: /* Route to M2 */
            2'b11: /* Route to M3 */
        endcase
    end
end
```

11.5 2.8.5 Multi-Slave Response Merging

11.5.1 Problem Statement

When multiple slaves can respond to same master simultaneously:

```
Scenario:
  Master 0 has outstanding transactions to Slave 0 and Slave 1
  Both slaves return R responses in same cycle

Problem: Master 0 can only accept one R response per cycle
Solution: Arbitrate between slave responses
```

11.5.2 Response Arbitration

```
// Arbitrate between multiple slave responses for same master
logic [2:0] slave_has_response;  // Which slaves have responses for M0
logic [1:0] selected_slave;      // Which slave wins arbitration

// Detect responses destined for M0
for (genvar s = 0; s < 3; s++) begin
    assign slave_has_response[s] = slave_rvalid[s] &&
                                   (extract_bid(slave_rid[s]) ==
2'b00);
end

// Round-robin arbiter for fairness
always_ff @(posedge clk) begin
    if (!rst_n) begin
        selected_slave <= 0;
    end else if (m0_rready && (|slave_has_response)) begin
        // Rotate selection for fairness
        if (slave_has_response[selected_slave])
            ; // Keep current
        else
            selected_slave <= find_next_requesting_slave();
    end
end

// MUX selected slave's response to master
assign m0_rvalid = slave_has_response[selected_slave];
assign m0_rdata = slave_rdata[selected_slave];
assign m0_rid = strip_bid(slave_rid[selected_slave]);
```

11.5.3 Response Buffering

Optional: Add FIFOs to buffer pending responses:

```
Purpose:
  - Prevent response loss during arbitration conflicts
  - Allow slaves to return responses even if master busy
  - Smooth out bursty response patterns

Configuration:
  per_master_response_fifo_depth = 4-16 entries

Trade-off:
  + No response loss
  + Better throughput
  - Additional resources (~200 LEs per FIFO)
  - +1-2 cycle latency
```

11.6 2.8.6 Backpressure Management

11.6.1 Ready Signal Routing

Master RREADY/BREADY must reach correct slave:

```
// Route master ready back to slaves
logic m0_rready;  // From master
logic [2:0] slave_rready;  // To slaves

// Only selected slave sees master's ready
for (genvar s = 0; s < 3; s++) begin
    assign slave_rready[s] = (selected_slave == s) ? m0_rready : 1'b0;
end

// Non-selected slaves see ready = 0 (backpressure)
```

11.6.2 Stall Conditions

Response path can stall when:

```
1. Master not ready (M_RREADY = 0)
   → Selected slave stalls (S_RREADY = 0)
   → Other slaves buffer or stall

2. Arbitration conflict (multiple slaves ready)
   → Non-selected slaves stalled
   → Selected slave proceeds

3. Response FIFO full
   → Slave stalled until space available
```

11.7 2.8.7 Response Path Latency

11.7.1 End-to-End R Channel

```
Slave R response → Bridge → Master R channel

Components:
1. Slave response valid
2. BID extraction (0 cycles, combinatorial)
3. Demux routing (0-1 cycles)
4. Response arbitration (1 cycle if conflict)
5. Optional FIFO (0-1 cycles)
6. Master adapter (1 cycle, skid buffer)

Total: 2-4 cycles typical
```

11.7.2 End-to-End B Channel

```
Similar to R channel:
  Slave B → Extraction → Demux → Arbitration → Master B
  Total: 2-4 cycles
```

11.8 2.8.8 Error Response Routing

11.8.1 Slave Error Responses

When slave returns error:

```
RRESP = 2'b10 (SLVERR) or 2'b11 (DECERR)
BRESP = 2'b10 (SLVERR) or 2'b11 (DECERR)

Routing:
  - Extract BID normally
  - Route to originating master
  - Preserve error code
  - Master sees error response
```

11.8.2 Out-of-Range Error Responses

When router generates error for OOR address:

```
Process:
1. Router captures OOR request
2. Generates internal error response
   - Uses cached request ID (with BID)
   - Sets RRESP/BRESP = DECERR
3. Injects into response path
4. Routes to master using BID
```

11.9 2.8.9 Resource Utilization

11.9.1 Response Router Resources

4 masters, 3 slaves (64-bit data):

```
Logic Elements:  ~1500-2000 LEs
Registers:       ~400-600 regs
Block RAM:       0 (unless FIFOs enabled)

Breakdown:
- BID extraction (3 slaves):     ~150 LEs
- Demux logic (4 masters):        ~600 LEs, ~150 regs
- Response arbitration:           ~300 LEs, ~100 regs
- Backpressure routing:           ~200 LEs, ~50 regs
- Control FSMs:                   ~250 LEs, ~100 regs

Optional FIFOs (4 × 8 entries):  +800 LEs, +2KB BRAM
```

11.9.2 Scaling

```
Resource usage scales with:
- Number of masters: Linear (one demux path per master)
- Number of slaves: Linear (one extraction per slave)
- Data width: Linear (wider data = wider demux)
- FIFO depth: Linear (deeper = more BRAM)
```

11.10 2.8.10 Configuration Parameters

11.10.1 Response Routing Configuration (TOML)

```toml
[bridge]
num_masters = 4
num_slaves = 3

[bridge.response_routing]
enable_response_fifos = false       # Buffer responses per master
fifo_depth = 8                      # Entries per FIFO
response_arbiter_type = "round_robin"  # "round_robin",
"fixed_priority"
registered_demux = false            # Register demux output (+1 cycle)

# Per-master response credits (optional)
[[masters]]
name = "cpu"
max_response_credits = 16           # Outstanding responses allowed
```

11.11 2.8.11 Debug and Observability

11.11.1 Recommended Debug Signals

```
BID Extraction:
- Slave RID/BID (from each slave)
- Extracted BID (master index)
- External RID/BID (to master, BID stripped)

Response Demux:
- Demux select signals (which master per slave)
- Response valid per master
- Response valid per slave

Arbitration:
- Conflict detection (multiple responses for same master)
- Arbiter grant (which slave wins)
- Stall counters

FIFOs (if enabled):
- FIFO occupancy per master
- FIFO full/empty flags
- FIFO overflow errors
```

11.11.2 Performance Counters

```
- Responses routed per master
- Arbitration conflicts (cycles with multiple responses to same
master)
- Response stall cycles (master not ready)
- Average response latency per master
- FIFO overflow counts
- CAM hits/misses (if CAM enabled)
```

11.12 2.8.12 Common Issues and Debug

Symptom: Response not reaching master
Check: - BID extraction (correct bit slicing?) - Master index (BID → master mapping) - Demux select logic - Backpressure (is master READY?)

Symptom: Response goes to wrong master
Check: - BID values (verify each master has unique BID) - BID width calculation (clog2 correct?) - CAM contents (if used) - ID corruption in transit

Symptom: Response ordering incorrect
Check: - CAM lookup (should preserve order) - Arbitration fairness (round-robin working?) - Multiple outstanding transactions per master

Symptom: Response loss
Check: - FIFO overflows (enable FIFOs if needed) - Dropped responses during arbitration conflicts - Protocol violations (slave not following AXI rules)

11.13 2.8.13 Verification Considerations

11.13.1 Test Scenarios

1. **Single Master, Single Slave**:

```
- Basic routing test
- Verify BID extraction and stripping
- Check response reaches correct master
```

2. **Multiple Masters, Single Slave**:

```
- Interleaved responses
- Verify each response routes to correct master
- Check BID uniqueness prevents collisions
```

3. **Single Master, Multiple Slaves**:

```
- Simultaneous responses from multiple slaves
- Verify arbitration fairness
- Check no response loss
```

4. **All Masters, All Slaves**:

```
- Maximum stress test
- All masters issuing to all slaves
- Responses returning in various orders
- Verify correct routing under heavy load
```

5. **OOO Responses**:

```
- Issue transactions: ID=1, ID=2, ID=3
- Return responses: ID=3, ID=1, ID=2
- Verify CAM correctly routes each
- Check response order at master
```

6. **Backpressure Test**:

```
- Master asserts RREADY=0
- Verify slave stalls (RREADY=0 propagated)
- Master asserts RREADY=1
- Verify responses flow
```

11.14 2.8.14 Performance Optimization

11.14.1 Techniques

1. Registered Demux:

```
Trade-off: +1 cycle latency for timing closure
Best for: High-frequency designs, many masters
```

2. Response FIFOs:

```
Trade-off: +2KB BRAM, +1-2 cycle latency
Benefit: Prevent response loss, smooth conflicts
Best for: High-throughput systems
```

3. Distributed Arbitration:

```
Implementation: Per-master arbiters instead of global
Benefit: Reduced logic depth, better timing
Best for: >8 masters
```

4. Priority Response Routing:

```
Implementation: High-priority masters get preference in conflicts
Benefit: Guaranteed low latency for critical masters
Best for: Real-time systems
```

11.15 2.8.15 Advanced Features

11.15.1 Response Reordering (Future)

Allow bridge to reorder responses for efficiency:

```
Scenario:
  Master issues: Req A (slow slave), Req B (fast slave)
  Fast slave responds first
  Bridge can deliver B before A (if IDs different)

Benefit: Reduced latency for fast responses
Requirement: CAM to track outstanding in-order constraints
```

11.15.2 Response Coalescing (Future)

Combine multiple responses into fewer beats:

```
Scenario:
  Multiple single-beat reads to narrow slave
  Bridge coaleses into fewer wide beats

Benefit: Reduced overhead
Complexity: High (requires buffering and alignment)
```

11.15.3 Response QoS (Future)

Prioritize responses based on master QoS:

```
Feature: High-QoS masters get response priority
Implementation: Weighted arbitration in response path
Use case: RT masters need bounded response latency
```

11.16 2.8.16 Timing Considerations

11.16.1 Critical Paths

Common critical paths in response routing:

```
1. BID extraction → Demux select → Master RDATA
   Depth: ~8-12 logic levels

2. Slave RVALID → Arbitration → Master RVALID
   Depth: ~6-10 logic levels

3. Master RREADY → Backpressure → Slave RREADY
   Depth: ~5-8 logic levels
```

11.16.2 Optimization Strategies

```
- Register demux outputs (+1 cycle, breaks path)
- Pipeline arbitration (+1-2 cycles, multi-stage)
- Use CAM for complex ID scenarios (parallel lookup)
- Limit response buffering depth (less MUX levels)
```

11.17 2.8.17 Future Enhancements

11.17.1 Planned Features

- **Dynamic Response Buffering**: Adjust FIFO depth based on utilization
- **Response Ordering Hints**: Masters specify ordering requirements
- **Multi-Cycle Arbitration**: Pipelined for >16 masters
- **Response Compression**: Pack narrow responses into wide beats

11.17.2 Under Consideration

- **Response Speculation**: Predictive routing before full ID available
- **Virtual Response Channels**: Separate paths for different QoS classes
- **Response Mirroring**: Duplicate responses for redundancy
- **Cross-Clock Response**: Async response paths with CDC

Related Sections: - Section 2.3: Crossbar Core (response path architecture) - Section 2.5: ID Management (BID extraction details) - Section 2.4: Arbitration (response arbitration algorithms) - Section 3.2: Master Port Interface (response channel signals)

12 2.9 Error Handling

Error Handling encompasses all mechanisms by which the bridge detects, reports, and recovers from error conditions. This includes protocol violations, out-of-range addresses, timeout conditions, and configuration errors.

12.1 2.9.1 Purpose and Function

Error handling performs the following critical functions:

1. **Error Detection**: Identifies violations and abnormal conditions
2. **Error Response Generation**: Creates appropriate AXI error responses
3. **Error Reporting**: Logs and signals errors for debug
4. **Graceful Degradation**: Maintains system stability during errors
5. **Recovery Support**: Enables system recovery after errors

12.2 2.9.2 Error Categories

12.2.1 Transaction Errors

Out-of-Range (OOR) Addresses:

```
Master requests address not mapped to any slave
Response: DECERR (Decode Error)
Action: Bridge generates error response internally
```

Protocol Violations:

```
Master violates AXI4 protocol rules
Examples:
  - VALID deasserted while READY=0
  - Incorrect burst parameters
  - ID width mismatches
Response: Configurable (error log, SLVERR, or ignore)
```

Slave Errors:

```
Slave returns SLVERR or DECERR response
Action: Bridge forwards error to master unchanged
```

12.2.2 System Errors

Timeout Conditions:

```
Slave fails to respond within configured timeout
Response: DECERR after timeout expires
Action: Force transaction completion
```

CAM Overflow:

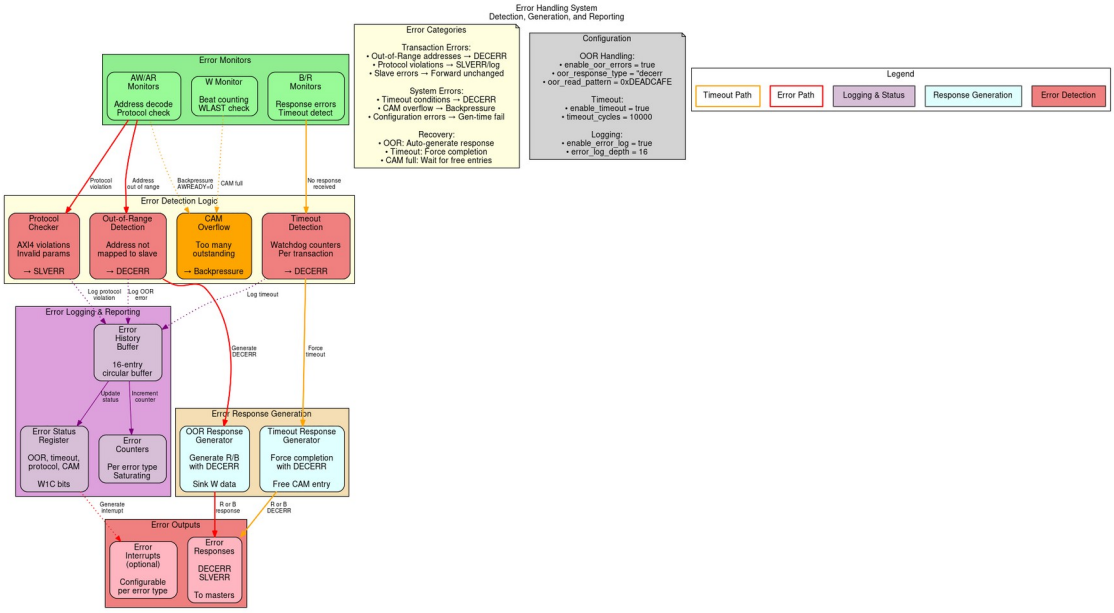
```
Too many outstanding transactions (CAM full)
Response: Backpressure master (ARREADY/AWREADY=0)
Action: Wait for responses to free entries
```

Configuration Errors:

```
Invalid bridge configuration detected at generation time
Examples:
  - Overlapping slave address ranges
  - Invalid data width combinations
  - Illegal parameter values
Response: Generation error, fail early
```

12.3 2.9.3 Block Diagram

12.3.1 Figure 2.9: Error Handling Architecture



Error Handling System

Error handling system showing detection (OOR, protocol, timeout, CAM), generation, and logging subsystems.

12.4 2.9.4 Out-of-Range Address Handling

12.4.1 Detection

Router identifies OOR when no slave address range matches:

```
// Address decode with OOR detection
logic [NUM_SLAVES-1:0] slave_match;
logic oor_detected;

for (genvar i = 0; i < NUM_SLAVES; i++) begin
    assign slave_match[i] = (addr >= SLAVE_BASE[i]) &&
                            (addr <= SLAVE_END[i]);
end

assign oor_detected = ~(|slave_match) && ~default_slave_en;
```

12.4.2 Error Response Generation

Read OOR Response:

```
// Generate R response for OOR read
always_ff @(posedge clk) begin
    if (ar_oor_detected && arvalid && arready) begin
        // Capture transaction details
        oor_arid <= arid;
        oor_arlen <= arlen;
        oor_burst_count <= 0;
        oor_generating <= 1'b1;
    end else if (oor_generating) begin
        // Generate R beats
        rvalid <= 1'b1;
        rdata <= OOR_DATA_PATTERN;  // Configurable pattern
        rid <= oor_arid;
        rresp <= 2'b11;  // DECERR
        rlast <= (oor_burst_count == oor_arlen);

        if (rready) begin
            oor_burst_count <= oor_burst_count + 1;
            if (rlast) oor_generating <= 1'b0;
        end
    end
end
```

Write OOR Response:

```
// Generate B response for OOR write
always_ff @(posedge clk) begin
    if (aw_oor_detected && awvalid && awready) begin
        // Capture write ID
        oor_awid <= awid;
        oor_w_count <= 0;
        oor_sinking_w <= 1'b1;
    end else if (oor_sinking_w) begin
        // Sink W data (accept and discard)
```

```
        wready <= 1'b1;

        if (wvalid && wlast) begin
            oor_sinking_w <= 1'b0;
            oor_gen_bresp <= 1'b1;
        end
    end else if (oor_gen_bresp) begin
        // Generate B response
        bvalid <= 1'b1;
        bid <= oor_awid;
        bresp <= 2'b11;  // DECERR

        if (bready) begin
            oor_gen_bresp <= 1'b0;
        end
    end
end
```

12.4.3 Configurable Data Patterns

```
[bridge.error_handling]
oor_read_data_pattern = 0xDEADCAFE  # Pattern for OOR reads
oor_response_latency = 2            # Cycles to generate response
```

12.5 2.9.5 Protocol Violation Handling

12.5.1 Common Violations

VALID Dropped While READY=0:

```
AXI4 Rule: Once VALID asserted, must remain until READY
Violation: Master deasserts VALID before handshake
Detection: Track VALID history per channel
Action: Log error, optionally assert error signal
```

Incorrect Burst Parameters:

```
Violations:
  - SIZE > DATA_WIDTH
  - LEN > 255
  - Address not aligned to SIZE
  - Burst crosses 4KB boundary
Detection: Parameter checking logic
Action: SLVERR response or reject transaction
```

ID Width Mismatch:

```
Violation: Master uses ID wider than configured
Detection: Check ID values against expected width
Action: Truncate or error (configurable)
```

12.5.2 Protocol Checker Implementation

```
// AXI4 protocol checker (simplified)
module axi_protocol_checker #(
    parameter ID_WIDTH = 4,
    parameter ADDR_WIDTH = 32,
    parameter DATA_WIDTH = 64
) (
    input logic clk, rst_n,
    // AXI signals
    input logic arvalid, arready,
    input logic [ADDR_WIDTH-1:0] araddr,
    input logic [7:0] arlen,
    input logic [2:0] arsize,
    // Error outputs
    output logic protocol_error,
    output logic [7:0] error_code
);

    // Check: VALID held until READY
    logic arvalid_prev;
    always_ff @(posedge clk) begin
        arvalid_prev <= arvalid;
        if (arvalid_prev && !arvalid && !arready) begin
            protocol_error <= 1'b1;
            error_code <= 8'h01;  // VALID_DROPPED
        end
    end

    // Check: SIZE <= DATA_WIDTH
    always_comb begin
        if (arvalid && (arsize > $clog2(DATA_WIDTH/8))) begin
            protocol_error = 1'b1;
            error_code = 8'h02;  // INVALID_SIZE
        end
    end

    // Check: Address alignment
    always_comb begin
        if (arvalid && (araddr[arsize-1:0] != 0)) begin
            protocol_error = 1'b1;
            error_code = 8'h03;  // UNALIGNED_ADDR
        end
    end

    // Additional checks...
endmodule
```

12.6 2.9.6 Timeout Detection

12.6.1 Per-Transaction Watchdog

```
// Timeout detector for outstanding transactions
typedef struct packed {
    logic valid;
    logic [TOTAL_ID_WIDTH-1:0] id;
    logic [15:0] timestamp;
    logic is_read;  // 1=read, 0=write
} timeout_entry_t;

timeout_entry_t watchdog [0:15];  // Track up to 16 transactions
logic [15:0] global_timer;

always_ff @(posedge clk) begin
    if (!rst_n) begin
        global_timer <= 0;
    end else begin
        global_timer <= global_timer + 1;

        // Check for timeouts
        for (int i = 0; i < 16; i++) begin
            if (watchdog[i].valid) begin
                logic [15:0] elapsed;
                elapsed = global_timer - watchdog[i].timestamp;

                if (elapsed > TIMEOUT_THRESHOLD) begin
                    // Timeout detected
                    timeout_error <= 1'b1;
                    timeout_id <= watchdog[i].id;
                    timeout_type <= watchdog[i].is_read ? "READ" :
"WRITE";

                    // Force completion with DECERR
                    force_error_response(watchdog[i]);
                    watchdog[i].valid <= 1'b0;
                end
            end
        end
    end
end
```

12.6.2 Timeout Configuration

```
[bridge.error_handling]
enable_timeout = true
timeout_cycles = 10000          # Max cycles before timeout
```

```
timeout_action = "decerr"      # "decerr", "slverr", or "hang"
per_slave_timeout = [          # Optional per-slave overrides
    {name = "slow_periph", timeout = 50000},
    {name = "fast_mem", timeout = 1000}
]
```

12.7 2.9.7 Error Logging and Reporting

12.7.1 Error Status Register

Error Status Register (Read/Clear):

Bits	Description
0	OOR Read Error
1	OOR Write Error
2	Protocol Violation
3	Timeout Error
4	CAM Overflow
5	Slave Error (SLVERR received)
6	Decode Error (DECERR received)
7	ID Match Error
15:8	Reserved
31:16	Error Count (saturating)

12.7.2 Error History Buffer

```
// Circular buffer for error history
typedef struct packed {
    logic [7:0] error_type;
    logic [31:0] address;
    logic [TOTAL_ID_WIDTH-1:0] id;
    logic [31:0] timestamp;
    logic [7:0] master_index;
    logic [7:0] slave_index;
} error_entry_t;

error_entry_t error_log [0:15];  // 16-entry history
logic [3:0] error_wr_ptr;

always_ff @(posedge clk) begin
    if (error_detected) begin
        error_log[error_wr_ptr] <= {
            error_type,
            error_addr,
            error_id,
            global_timer,
```

```
            error_master,
            error_slave
        };
        error_wr_ptr <= error_wr_ptr + 1;
    end
end
```

12.7.3 Error Interrupts

```
Optional interrupt generation:

Interrupt Enable Register:
  - OOR_INT_EN
  - TIMEOUT_INT_EN
  - PROTOCOL_INT_EN
  - etc.

Interrupt when enabled error occurs
Clear on status register read or explicit clear
```

12.8 2.9.8 Resource Utilization

12.8.1 Error Handling Resources

```
Logic Elements:  ~800-1200 LEs
Registers:       ~400-600 regs
Block RAM:       0-2 KB (if error logging enabled)

Breakdown:
- OOR detection/response:     ~300 LEs, ~100 regs
- Protocol checkers:           ~250 LEs, ~50 regs
- Timeout watchers:            ~200 LEs, ~150 regs
- Error logging:               ~150 LEs, ~100 regs
- Status registers:            ~100 LEs, ~50 regs

Optional error log (16 entries): +2KB BRAM
```

12.9 2.9.9 Configuration Parameters

12.9.1 Error Handling Configuration (TOML)

```toml
[bridge.error_handling]
# OOR handling
enable_oor_errors = true
oor_response_type = "decerr"      # "decerr", "slverr"
oor_read_pattern = 0xDEADCAFE     # Data pattern for OOR reads
```

```
# Timeout detection
enable_timeout = true
timeout_cycles = 10000
timeout_response = "decerr"

# Protocol checking
enable_protocol_checker = true
protocol_checker_mode = "log"     # "log", "error", "ignore"

# Error logging
enable_error_log = true
error_log_depth = 16              # Entries in circular buffer

# Error reporting
enable_error_interrupts = false
error_status_register_addr = 0xFFFF_FFF0  # Optional mapped register
```

12.10 2.9.10 Debug and Observability

12.10.1 Recommended Debug Signals

```
Error Detection:
- OOR flags (per master)
- Protocol violation flags
- Timeout counters
- CAM occupancy

Error Response:
- Error response generation FSM states
- Active error responses
- Error data patterns

Error Logging:
- Error count
- Last error type
- Last error address
- Error log write pointer
```

12.11 2.9.11 Common Issues and Debug

Symptom: Unexpected DECERR responses
Check: - Address map configuration (gaps?) - Slave address ranges (overlaps?) - Default slave configuration - Timeout thresholds (too short?)

Symptom: System hangs on certain addresses
Check: - Timeout detection enabled? - Slave responsiveness - Protocol violations upstream - CAM overflow

Symptom: Error log filling quickly
Check: - What errors are occurring? - Master behavior (bad addresses?) - Slave configuration - Timeout thresholds

12.12 2.9.12 Verification Considerations

12.12.1 Test Scenarios

1. **OOR Address Tests**:

```
- Read from unmapped address
- Write to unmapped address
- Burst to partially mapped range
- Verify DECERR response
```

2. **Timeout Tests**:

```
- Slave never responds
- Verify timeout after N cycles
- Check error logged
- Verify forced completion
```

3. **Protocol Violation Tests**:

```
- Drop VALID before READY
- Invalid burst parameters
- Misaligned addresses
- Verify detection and response
```

4. **Error Recovery Tests**:

```
- Error occurs
- System continues operating
- Subsequent transactions succeed
- Error status readable
```

5. **Error Logging Tests**:

```
- Generate multiple errors
- Verify all logged
- Check circular buffer wrap
- Verify timestamps
```

12.13 2.9.13 Recovery Mechanisms

12.13.1 Automatic Recovery

```
OOR Errors:
  - Generate error response
  - Continue normal operation
  - No intervention needed

Timeout Errors:
  - Force transaction completion
  - Free CAM entry
  - Allow new transactions

Protocol Violations:
  - Log error
  - Complete/reject transaction
  - Continue with next transaction
```

12.13.2 Manual Recovery

```
CAM Stuck:
  - Software reset via control register
  - Clear CAM entries
  - Restart affected masters

Persistent Errors:
  - Read error log
  - Identify root cause
  - Reconfigure or reset subsystem
```

12.14 2.9.14 Safety and Reliability

12.14.1 Error Containment

```
Goal: Prevent error propagation

Mechanisms:
- Isolate error to originating master
- Don't corrupt other masters' transactions
- Maintain crossbar integrity
- Log for post-mortem analysis
```

12.14.2 Fail-Safe Behavior

```
On Critical Error:
1. Generate safe error response (DECERR)
2. Log error details
```

```
3. Assert error signal (optional)
4. Continue operation if possible
5. Halt only if configured (safety-critical mode)
```

12.14.3 Error Injection (Debug/Test)

```toml
[bridge.debug]
enable_error_injection = true

[[bridge.debug.error_injection]]
type = "oor"
trigger_address = 0xBAD_ADDR
action = "decerr"

[[bridge.debug.error_injection]]
type = "timeout"
trigger_after_n_cycles = 100
target_slave = 2
```

12.15 2.9.15 Future Enhancements

12.15.1 Planned Features

- **Error Recovery FSM**: Automatic recovery from certain error conditions
- **Error Rate Limiting**: Prevent error storm from misbehaving master
- **Detailed Error Telemetry**: Capture full transaction context on error
- **Error Prediction**: Detect patterns indicating impending failures

12.15.2 Under Consideration

- **ECC Protection**: Error correction for internal state
- **Redundant Checking**: Duplicate checkers for safety-critical
- **Watchdog Timers**: System-level health monitoring
- **Error Severity Levels**: Classify errors by impact

Related Sections: - Section 2.2: Slave Router (OOR detection) - Section 2.5: ID Management (CAM overflow) - Section 2.8: Response Routing (error response paths) - Chapter 5: Verification (error testing strategies)

12.15.3 Bridge Arbiter Finite State Machines

12.15.3.1 Overview

The Bridge AXI4 crossbar implements independent per-slave arbitration using simple 2-state finite state machines (FSMs). Each slave has two dedicated arbiters:

- **AW Arbiter** - Write Address channel arbitration
- **AR Arbiter** - Read Address channel arbitration

Total FSM Count: `2 × NUM_SLAVES`

These arbiters provide fair, round-robin access to slave resources, preventing master starvation while maintaining simple, predictable behavior.

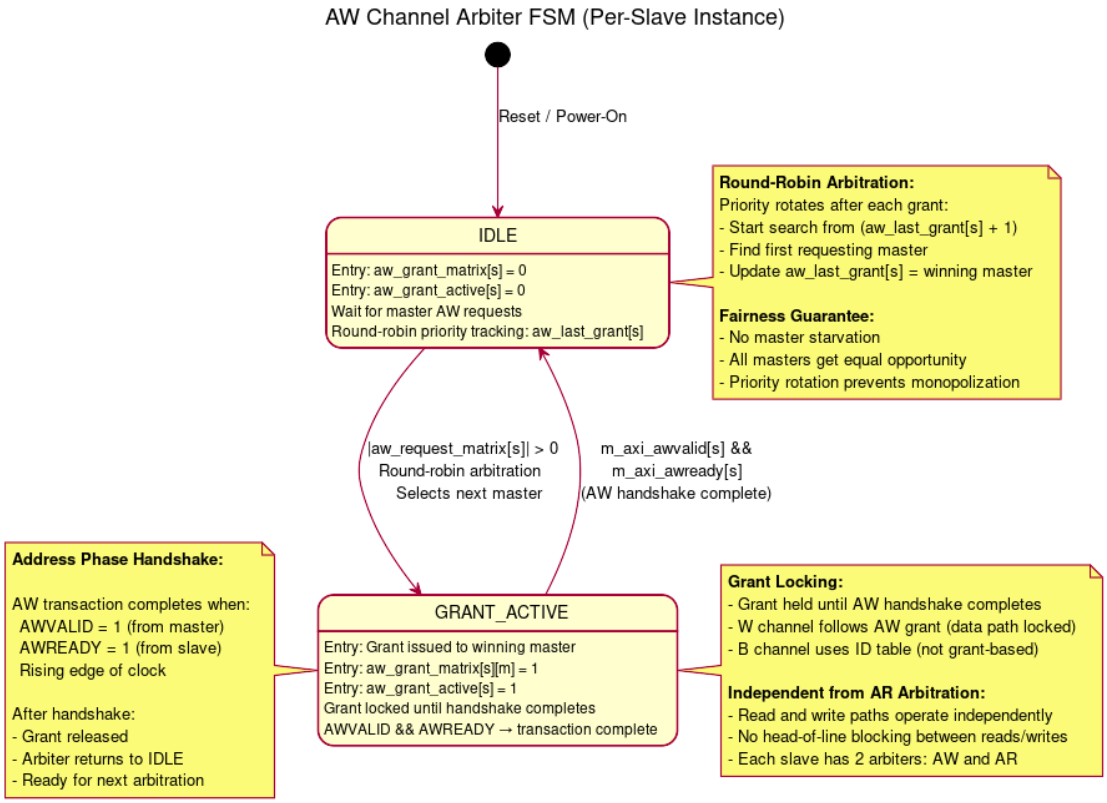
12.15.3.2 AW Channel Arbiter FSM

Purpose: Arbitrate write address requests from multiple masters to a single slave

States: 2 (IDLE, GRANT_ACTIVE)

Algorithm: Round-robin with grant locking

12.15.4 Figure 5.3: AW Arbiter FSM



AW Arbiter FSM

State Descriptions:

IDLE State: - **Entry Actions:** - `aw_grant_matrix[s] = 0` - No grant issued - `aw_grant_active[s] = 0` - Arbiter inactive - **Behavior:** - Monitor `aw_request_matrix[s]` for

master requests - Track round-robin priority via `aw_last_grant[s]` - Remain in IDLE until at least one master requests access

GRANT_ACTIVE State: - **Entry Actions:** - `aw_grant_matrix[s][m] = 1` - Issue grant to winning master - `aw_grant_active[s] = 1` - Mark arbiter as active - **Behavior:** - Hold grant until AW handshake completes (`AWVALID && AWREADY`) - W channel data multiplexing follows AW grant - B channel response uses ID table (not grant-based)

State Transitions:

From	To	Condition	Description
IDLE	GRANT_ACTIVE	`| aw_request_matrix[s] | > 0`	At least one master requesting access
GRANT_ACTIVE	IDLE	`m_axi_awvalid[s] && m_axi_awready[s]`	AW handshake complete

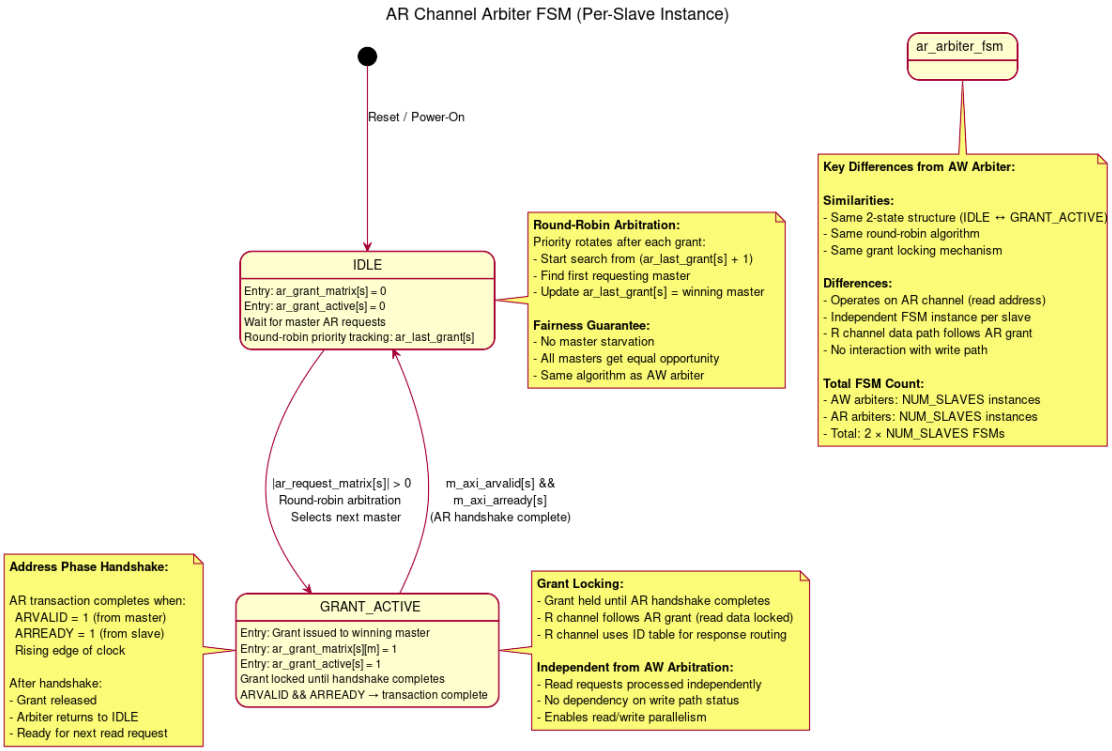
12.15.4.1 AR Channel Arbiter FSM

Purpose: Arbitrate read address requests from multiple masters to a single slave

States: 2 (IDLE, GRANT_ACTIVE)

Algorithm: Round-robin with grant locking (same as AW)

12.15.5 Figure 5.4: AR Arbiter FSM



AR Arbiter FSM

State Descriptions:

IDLE State: - **Entry Actions:** - `ar_grant_matrix[s] = 0` - No grant issued - `ar_grant_active[s] = 0` - Arbiter inactive - **Behavior:** - Monitor `ar_request_matrix[s]` for master requests - Track round-robin priority via `ar_last_grant[s]` - Independent from AW arbiter state

GRANT_ACTIVE State: - **Entry Actions:** - `ar_grant_matrix[s][m] = 1` - Issue grant to winning master - `ar_grant_active[s] = 1` - Mark arbiter as active - **Behavior:** - Hold grant until AR handshake completes (`ARVALID && ARREADY`) - R channel data multiplexing follows AR grant - R channel response uses ID table for routing

State Transitions:

From	To	Condition	Description
IDLE	GRANT_ACTIVE	`| ar_request_matrix[s] | > 0`	At least one master requesting read
GRANT_ACTIVE	IDLE	`m_axi_arvalid[s] && m_axi_arready[s]`	AR handshake complete

12.15.5.1 Round-Robin Arbitration Algorithm

Algorithm Description:

Both AW and AR arbiters use the same fair round-robin arbitration algorithm:

```
Pseudocode:
1. Start search from (last_grant[s] + 1) % NUM_MASTERS
2. Search cyclically through all masters
3. Select first master with pending request
4. Update last_grant[s] = winning_master
5. Issue grant
```

Example with 4 Masters:

```
Initial state: last_grant[0] = 0

Request Pattern:
  Master 0: request
  Master 1: request
  Master 2: no request
  Master 3: request

Arbitration Sequence:
  Grant 1: Master 1 (start from master 1, first requester)
          last_grant = 1

  Grant 2: Master 3 (start from master 2, find master 3)
          last_grant = 3

  Grant 3: Master 0 (start from master 0, first requester)
          last_grant = 0

  Grant 4: Master 1 (start from master 1, first requester)
          last_grant = 1
```

Fairness Properties:

1. **No Starvation** - Every requesting master will eventually receive grant
2. **Priority Rotation** - Priority shifts after each grant
3. **Predictable Latency** - Maximum wait time = (NUM_MASTERS - 1) × grant_duration
4. **Equal Opportunity** - All masters treated equally

12.15.5.2 Grant Locking Mechanism

Purpose: Prevent grant changes mid-transaction

AW Grant Locking:

```
// AW grant locked until address handshake completes
always_ff @(posedge aclk or negedge aresetn) begin
    if (!aresetn) begin
        aw_grant_active[s] <= 1'b0;
        aw_grant_matrix[s] <= '0;
    end else begin
        if (aw_grant_active[s]) begin
            // GRANT_ACTIVE state: Hold grant until handshake
            if (m_axi_awvalid[s] && m_axi_awready[s]) begin
                aw_grant_active[s] <= 1'b0;  // Return to IDLE
                aw_grant_matrix[s] <= '0;
            end
        end else if (|aw_request_matrix[s]) begin
            // IDLE state: Arbitrate if requests pending
            aw_grant_matrix[s] <=
round_robin_select(aw_request_matrix[s]);
            aw_grant_active[s] <= 1'b1;  // Enter GRANT_ACTIVE
        end
    end
end
```

AR Grant Locking:

```
// AR grant locked until address handshake completes
always_ff @(posedge aclk or negedge aresetn) begin
    if (!aresetn) begin
        ar_grant_active[s] <= 1'b0;
        ar_grant_matrix[s] <= '0;
    end else begin
        if (ar_grant_active[s]) begin
            // GRANT_ACTIVE state: Hold grant until handshake
            if (m_axi_arvalid[s] && m_axi_arready[s]) begin
                ar_grant_active[s] <= 1'b0;  // Return to IDLE
                ar_grant_matrix[s] <= '0;
            end
        end else if (|ar_request_matrix[s]) begin
            // IDLE state: Arbitrate if requests pending
            ar_grant_matrix[s] <=
round_robin_select(ar_request_matrix[s]);
            ar_grant_active[s] <= 1'b1;  // Enter GRANT_ACTIVE
        end
    end
end
```

Why Grant Locking Matters:

1. **Protocol Compliance** - AXI4 spec requires stable AWID/ARID during handshake
2. **Data Integrity** - W channel must follow granted master's AW request
3. **Response Routing** - B/R channels need consistent transaction tracking
4. **Simplicity** - Single grant active per slave prevents mux conflicts

12.15.5.3 Independent Read/Write Arbitration

Key Design Feature: AW and AR arbiters operate completely independently

Benefits:

1. **No Head-of-Line Blocking:**
   - Read requests don't wait for write completion
   - Write requests don't wait for read completion
2. **Parallel Operation:**
   - Master 0 can write to Slave 0 (AW path)
   - Master 1 can read from Slave 0 (AR path)
   - Both transactions proceed simultaneously
3. **Resource Efficiency:**
   - Full utilization of read/write bandwidth
   - No artificial serialization

Example Scenario:

```
Time T0: Master 0 issues write to Slave 0
         - AW arbiter grants to Master 0
         - AW_GRANT_ACTIVE = 1
         - W data follows

Time T1: Master 1 issues read to Slave 0 (during Master 0 write)
         - AR arbiter grants to Master 1 (independent!)
         - AR_GRANT_ACTIVE = 1
         - R data returns

Result: Read and write happen in parallel - no blocking
```

12.15.5.4 FSM Instance Breakdown by Configuration

2×2 Configuration (2 masters, 2 slaves): - Slave 0 AW Arbiter: 1 FSM - Slave 0 AR Arbiter: 1 FSM - Slave 1 AW Arbiter: 1 FSM - Slave 1 AR Arbiter: 1 FSM - **Total: 4 FSMs**

4×4 Configuration (4 masters, 4 slaves): - 4 slaves × 2 arbiters per slave = **8 FSMs**

8×8 Configuration (8 masters, 8 slaves): - 8 slaves × 2 arbiters per slave = **16 FSMs**

Scalability: - FSM count scales linearly with NUM_SLAVES - Arbitration complexity scales with NUM_MASTERS (search time) - Synthesis impact minimal for up to 16×16 configurations

12.15.5.5 Performance Characteristics

Latency:

- **Best Case:** 1 clock cycle (IDLE → GRANT_ACTIVE → IDLE with immediate handshake)
- **Worst Case:** (NUM_MASTERS) clock cycles (round-robin search + contention)
- **Average Case:** ~(NUM_MASTERS / 2) clock cycles

Throughput:

- **Back-to-Back Grants:** Possible if different masters (no wait)
- **Same Master Repeated:** 1 cycle minimum between grants (FSM state change)

Fairness:

- **Guaranteed:** Every master gets grant within (NUM_MASTERS - 1) arbitration cycles
- **No Priority:** All masters treated equally (can be extended for QoS)

12.15.5.6 Comparison with Other Crossbar Arbiters

Feature	Bridge Arbiter	APB Crossbar	Commercial Tools
States	2 (IDLE, GRANT_ACTIVE)	1 (passthrough)	3-5 (complex)
Algorithm	Round-robin	N/A (APB is 1:1)	Priority, QoS, weighted
Grant Locking	Yes (until handshake)	N/A	Burst-aware locking
Read/ Write Indep	Yes (2 FSMs/slave)	N/A	Yes
Complexity	Low	Minimal	High
Predictability	High (round-robin)	N/A	Medium (priority-based)

Simplicity Trade-off:

- **Advantages:**
  - Easy to verify (2-state FSM)
  - Predictable latency
  - Fair resource allocation
- **Limitations (future enhancements):**
  - No Quality-of-Service (QoS) support
  - No priority levels
  - No weighted arbitration

12.15.5.7 Future Enhancements (Phase 3+)

QoS Support:

```
// Master QoS priority field (AXI4 optional)
input logic [3:0] s_axi_awqos [NUM_MASTERS];

// Arbitration considers QoS priority
function automatic [NUM_MASTERS-1:0] qos_arbitrate(
    input logic [NUM_MASTERS-1:0] requests,
    input logic [3:0] qos [NUM_MASTERS]
);
    // Higher QoS value = higher priority
    // Round-robin among same QoS level
endfunction
```

Weighted Arbitration:

```
// Master weight configuration
parameter int MASTER_WEIGHTS [NUM_MASTERS] = '{1, 2, 1, 4};

// Grant frequency proportional to weight
// Master 3 gets 4× more grants than Master 0
```

Burst-Aware Locking:

```
// Lock grant until entire burst completes
// Currently: Lock until address handshake
// Enhanced: Lock until WLAST (write) or RLAST (read)
```

See Also: - 1.1 - Introduction - Bridge overview - 3.1 - Module Structure - Generated RTL organization - 3.3 - Crossbar Core - Internal crossbar instantiation

Reference: - ARM AMBA AXI4 Specification (IHI 0022E) - Section A7 (Arbitration)

Next: 3.3 - Crossbar Core

13 Transaction Tracking FSMs

13.1 Overview

Bridge uses several FSMs to track outstanding transactions and ensure correct response routing. These FSMs work in conjunction with the ID management CAM to maintain transaction state.

13.2 Write Data Tracking FSM

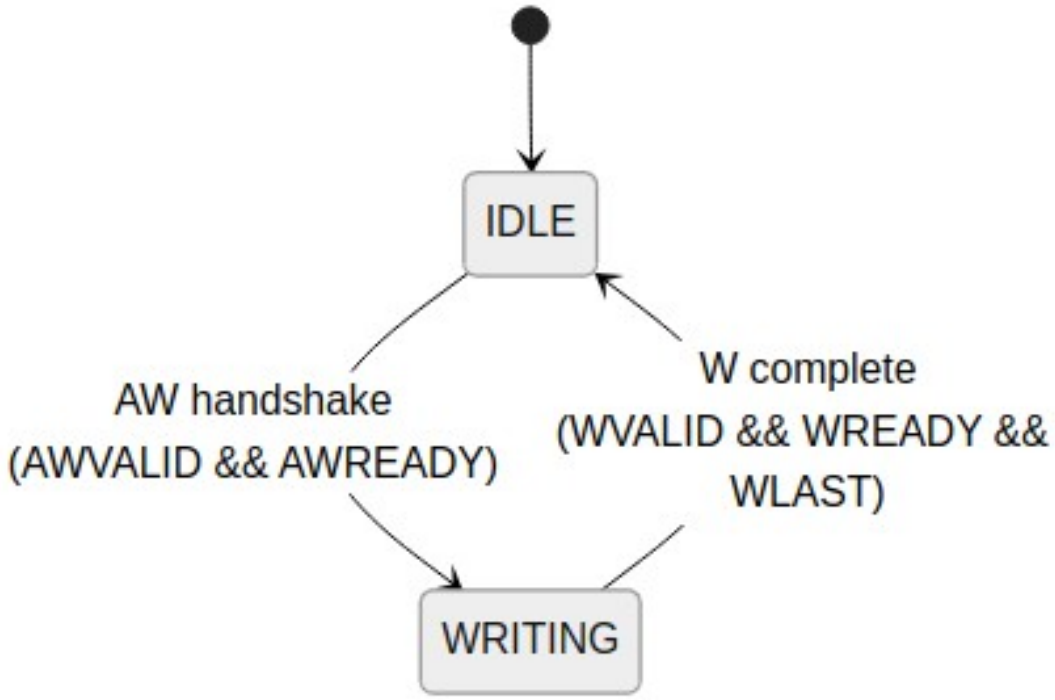
13.2.1 Purpose

Track W channel data beats to ensure correct routing after AW handshake.

13.2.2 States

State	Description
IDLE	Waiting for AW handshake
WRITING	Routing W beats to target slave

13.2.3 Figure 3.1: Write Data Tracking FSM



Write Data Tracking FSM

13.2.4 Implementation

```
typedef enum logic [0:0] {
    W_IDLE    = 1'b0,
    W_WRITING = 1'b1
} w_state_t;

w_state_t r_w_state;
logic [$clog2(NUM_SLAVES)-1:0] r_w_target;

always_ff @(posedge clk or negedge rst_n) begin
    if (!rst_n) begin
        r_w_state <= W_IDLE;
        r_w_target <= '0;
    end else begin
        case (r_w_state)
            W_IDLE: begin
                if (awvalid && awready) begin
                    r_w_state <= W_WRITING;
                    r_w_target <= decoded_slave;
                end
            end
```

```
            W_WRITING: begin
                if (wvalid && wready && wlast) begin
                    r_w_state <= W_IDLE;
                end
            end
        endcase
    end
end
```

13.3 Response Pending Counter

13.3.1 Purpose

Track number of outstanding transactions per master for flow control.

13.3.2 Implementation

```
logic [7:0] r_outstanding [NUM_MASTERS];

// Increment on AR/AW acceptance
always_ff @(posedge clk or negedge rst_n) begin
    if (!rst_n) begin
        for (int i = 0; i < NUM_MASTERS; i++)
            r_outstanding[i] <= '0;
    end else begin
        for (int m = 0; m < NUM_MASTERS; m++) begin
            logic inc = (ar_accepted[m] || aw_accepted[m]);
            logic dec = (r_delivered[m] || b_delivered[m]);

            case ({inc, dec})
                2'b10: r_outstanding[m] <= r_outstanding[m] + 1;
                2'b01: r_outstanding[m] <= r_outstanding[m] - 1;
                default: r_outstanding[m] <= r_outstanding[m];
            endcase
        end
    end
end

// Backpressure when outstanding limit reached
assign accept_ar[m] = (r_outstanding[m] < MAX_OUTSTANDING);
assign accept_aw[m] = (r_outstanding[m] < MAX_OUTSTANDING);
```

13.4 Burst Counter FSM

13.4.1 Purpose

Track burst beat count for responses spanning multiple cycles.

13.4.2 States

State	Description
IDLE	No burst in progress
COUNTING	Tracking burst beats

13.4.3 Implementation

```
logic [7:0] r_burst_count;
logic r_burst_active;

always_ff @(posedge clk or negedge rst_n) begin
    if (!rst_n) begin
        r_burst_count <= '0;
        r_burst_active <= 1'b0;
    end else begin
        if (!r_burst_active && rvalid && rready) begin
            // Start new burst
            r_burst_active <= !rlast;
            r_burst_count <= rlast ? '0 : 8'd1;
        end else if (r_burst_active && rvalid && rready) begin
            // Continue burst
            r_burst_count <= r_burst_count + 1;
            r_burst_active <= !rlast;
        end
    end
end
```

13.5 Related Documentation

- Arbiter FSMs - Address channel arbitration
- ID Management - CAM for response tracking

14 CAM Architecture

14.1 Overview

Bridge uses Content-Addressable Memory (CAM) structures to track outstanding transactions and enable out-of-order response routing. The CAM allows fast lookup of master origin based on transaction ID.

14.2 CAM Purpose

14.2.1 Transaction Tracking

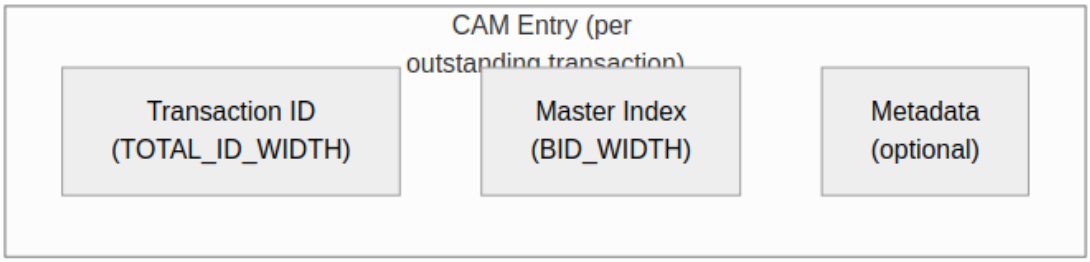
The CAM stores: - Transaction ID (key) - Originating master index (value) - Transaction metadata (optional)

14.2.2 Response Routing

When a response arrives: 1. Extract transaction ID from response 2. CAM lookup returns originating master 3. Route response to correct master

14.3 CAM Structure

14.3.1 Figure 4.1: CAM Entry Format



CAM Entry Format

14.3.2 CAM Sizing

```
CAM Depth = MAX_OUTSTANDING × NUM_MASTERS
CAM Width = TOTAL_ID_WIDTH + BID_WIDTH + METADATA_WIDTH
```

14.3.3 Example Configuration

```
4 masters, 4-bit external ID, max 16 outstanding per master:
  BID_WIDTH = clog2(4) = 2
  TOTAL_ID_WIDTH = 4 + 2 = 6
  CAM Depth = 16 × 4 = 64 entries
  CAM Width = 6 + 2 + 8 = 16 bits
```

14.4 CAM Operations

14.4.1 Insertion (AR/AW Acceptance)

```
// On AR handshake
if (arvalid && arready) begin
    cam_write_en <= 1'b1;
    cam_write_id <= {bid, arid};  // Key
```

```
    cam_write_master <= master_idx;  // Value
end
```

14.4.2 Lookup (R/B Response)

```
// On R response from slave
logic [BID_WIDTH-1:0] response_master;
logic cam_hit;

cam_lookup(slave_rid, response_master, cam_hit);

if (cam_hit) begin
    route_response_to(response_master);
end else begin
    // Error: Unknown transaction
end
```

14.4.3 Deletion (Response Complete)

```
// On RLAST or B response
if ((rvalid && rready && rlast) || (bvalid && bready)) begin
    cam_delete_en <= 1'b1;
    cam_delete_id <= response_id;
end
```

14.5 Implementation Options

14.5.1 Distributed RAM CAM

For small outstanding counts (< 16 entries):

```
// Parallel comparator-based CAM
logic [DEPTH-1:0] valid;
logic [ID_WIDTH-1:0] id_table [DEPTH];
logic [BID_WIDTH-1:0] master_table [DEPTH];

// Parallel compare for lookup
logic [DEPTH-1:0] match;
for (genvar i = 0; i < DEPTH; i++) begin
    assign match[i] = valid[i] && (id_table[i] == lookup_id);
end
```

14.5.2 Block RAM CAM

For larger outstanding counts (> 16 entries):

```
// Sequential search CAM (saves logic)
logic [$clog2(DEPTH)-1:0] search_idx;
logic searching;
```

```
always_ff @(posedge clk) begin
    if (start_lookup) begin
        searching <= 1'b1;
        search_idx <= 0;
    end else if (searching) begin
        if (match_found || search_idx == DEPTH-1)
            searching <= 1'b0;
        else
            search_idx <= search_idx + 1;
    end
end
```

14.6 CAM Overflow Handling

14.6.1 Prevention

```
// Backpressure when CAM full
assign ar_ready = !cam_full && downstream_ready;
assign aw_ready = !cam_full && downstream_ready;
```

14.6.2 Error Response

If CAM overflow would occur: 1. Assert backpressure (ARREADY/AWREADY = 0) 2. Wait for responses to free entries 3. Never drop transactions

14.7 Related Documentation

- ID Tracking - ID table implementation
- Response Routing - Using CAM for routing

15 ID Tracking Tables

15.1 Overview

ID tracking tables maintain the mapping between extended IDs (with Bridge ID) and originating masters. These tables enable correct response routing in multi-master systems.

15.2 Table Structure

15.2.1 Per-Slave ID Table

Each slave has its own ID tracking table:

15.2.2 Figure 4.2: ID Table Structure


Slave 0 ID Table
Index | Transaction ID | Master | Valid
0 | 6'b00_0101 | 0 | 1
1 | 6'b01_0011 | 1 | 1
2 | 6'b10_1010 | 2 | 0
3 | 6'b11_0001 | 3 | 1
... | ... | ... | ...

ID Table Structure

15.2.3 Table Sizing

```
Entries = MAX_OUTSTANDING_PER_SLAVE
Width = TOTAL_ID_WIDTH + clog2(NUM_MASTERS) + 1 (valid)
```

15.3 ID Extension

15.3.1 Master-Side Extension

When transaction enters Bridge:

```
// Extend ID with Bridge ID (master index)
logic [TOTAL_ID_WIDTH-1:0] extended_id;
assign extended_id = {master_bid, external_id};

// Example: Master 2, external ID = 4'hA
// master_bid = 2'b10, external_id = 4'b1010
// extended_id = 6'b10_1010
```

15.3.2 Slave-Side Presentation

Extended ID goes to slave:

```
assign s_axi_arid = extended_id;  // 6 bits to slave
assign s_axi_awid = extended_id;  // 6 bits to slave
```

15.4 ID Extraction

15.4.1 Response Parsing

When response returns from slave:

```
// Extract Bridge ID and external ID
logic [BID_WIDTH-1:0] bridge_id;
logic [ID_WIDTH-1:0] external_id;

assign bridge_id = slave_rid[TOTAL_ID_WIDTH-1:ID_WIDTH];
assign external_id = slave_rid[ID_WIDTH-1:0];

// Route to master based on bridge_id
assign m_rvalid[bridge_id] = s_rvalid;
assign m_rid[bridge_id] = external_id;  // Strip Bridge ID
```

15.5 Table Operations

15.5.1 Allocation (AR/AW Phase)

```
always_ff @(posedge clk) begin
    if (arvalid && arready) begin
        // Find free entry
        for (int i = 0; i < DEPTH; i++) begin
            if (!valid[i]) begin
                id_table[i] <= extended_arid;
                master_table[i] <= master_idx;
                valid[i] <= 1'b1;
                break;
            end
        end
    end
end
```

15.5.2 Lookup (R/B Phase)

```
// Combinational lookup
logic [NUM_MASTERS-1:0] master_onehot;
always_comb begin
    master_onehot = '0;
    for (int i = 0; i < DEPTH; i++) begin
        if (valid[i] && (id_table[i] == response_id)) begin
            master_onehot[master_table[i]] = 1'b1;
        end
    end
end
```

15.5.3 Deallocation (Response Complete)

```
always_ff @(posedge clk) begin
    if (rvalid && rready && rlast) begin
        // Find and invalidate matching entry
        for (int i = 0; i < DEPTH; i++) begin
            if (valid[i] && (id_table[i] == response_rid)) begin
                valid[i] <= 1'b0;
                break;
            end
        end
    end
end
```

15.6 Multi-ID Considerations

15.6.1 Same External ID, Different Masters

```
Master 0 issues ID=5 → Extended: 00_0101
Master 1 issues ID=5 → Extended: 01_0101
Master 2 issues ID=5 → Extended: 10_0101

All three can be outstanding simultaneously!
The extended ID ensures uniqueness.
```

15.6.2 Same Extended ID (Error)

This cannot happen: - Bridge ID is unique per master - Extended ID = {unique BID, external ID} - Same extended ID implies same master + same external ID - AXI4 requires unique IDs per master for outstanding transactions

15.7 Resource Utilization

15.7.1 Table Resources

```
4 masters, 4-bit external ID, 16 outstanding per slave:

Per-slave table:
  Entries: 16
  Width: 6 + 2 + 1 = 9 bits
  Total: 16 × 9 = 144 bits

4 slaves:
  Total: 4 × 144 = 576 bits (~72 bytes)

Implementation:
  Distributed RAM: ~150 LEs
  Block RAM: 1 BRAM (minimum)
```

15.8 Related Documentation

- CAM Architecture - CAM implementation details
- Response Routing - Using tables for routing

16 Width Converters

16.1 Overview

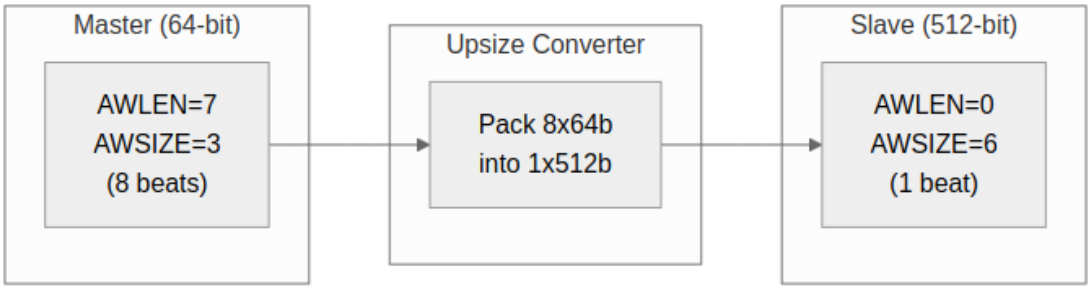
Width converters handle data width mismatches between masters and slaves. Bridge automatically inserts upsizers (narrow to wide) and downsizers (wide to narrow) as needed.

16.2 Upsize Converter

16.2.1 Purpose

Convert narrow master data to wide slave interface.

16.2.2 Figure 5.1: Upsize Converter (64-bit to 512-bit)



Upsize Converter

16.2.3 Implementation

```
module width_upsize #(
    parameter IN_WIDTH = 64,
    parameter OUT_WIDTH = 512
) (
    input  logic clk, rst_n,

    // Narrow input
    input  logic                    s_valid,
    output logic                    s_ready,
    input  logic [IN_WIDTH-1:0]     s_data,
    input  logic                    s_last,

    // Wide output
    output logic                    m_valid,
    input  logic                    m_ready,
    output logic [OUT_WIDTH-1:0]    m_data,
    output logic                    m_last
);
```

```
    localparam RATIO = OUT_WIDTH / IN_WIDTH;

    logic [OUT_WIDTH-1:0] r_buffer;
    logic [$clog2(RATIO)-1:0] r_count;

    always_ff @(posedge clk or negedge rst_n) begin
        if (!rst_n) begin
            r_count <= '0;
            r_buffer <= '0;
        end else if (s_valid && s_ready) begin
            // Pack input into buffer
            r_buffer[r_count * IN_WIDTH +: IN_WIDTH] <= s_data;

            if (s_last || r_count == RATIO - 1) begin
                r_count <= '0;
            end else begin
                r_count <= r_count + 1;
            end
        end
    end

    // Output when buffer full or last beat
    assign m_valid = (r_count == RATIO - 1) || s_last;
    assign m_data = r_buffer;
    assign s_ready = !m_valid || m_ready;

endmodule
```

16.2.4 Burst Length Conversion

Table 5.3: Burst Length Conversion Examples

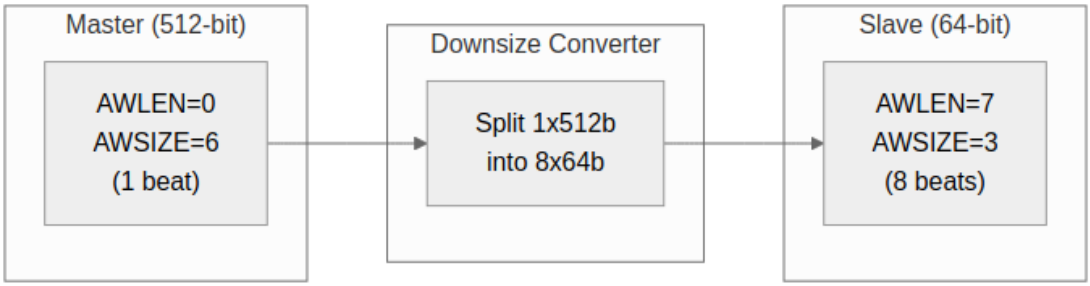
Master AWLEN	Master Beats	Slave AWLEN	Slave Beats
7 (8 beats)	8 × 64b	0 (1 beat)	1 × 512b
15 (16 beats)	16 × 64b	1 (2 beats)	2 × 512b
255 (256 beats)	256 × 64b	31 (32 beats)	32 × 512b

16.3 Downsize Converter

16.3.1 Purpose

Convert wide master data to narrow slave interface.

16.3.2 Figure 5.2: Downsize Converter (512-bit to 64-bit)



Downsize Converter

16.3.3 Implementation

```
module width_downsize #(
    parameter IN_WIDTH = 512,
    parameter OUT_WIDTH = 64
) (
    input  logic clk, rst_n,

    // Wide input
    input  logic                    s_valid,
    output logic                    s_ready,
    input  logic [IN_WIDTH-1:0]     s_data,
    input  logic                    s_last,

    // Narrow output
    output logic                    m_valid,
    input  logic                    m_ready,
    output logic [OUT_WIDTH-1:0]    m_data,
    output logic                    m_last
);

    localparam RATIO = IN_WIDTH / OUT_WIDTH;

    logic [IN_WIDTH-1:0] r_buffer;
    logic [$clog2(RATIO)-1:0] r_count;
    logic r_active;

    always_ff @(posedge clk or negedge rst_n) begin
        if (!rst_n) begin
            r_count <= '0;
            r_buffer <= '0;
            r_active <= 1'b0;
        end else begin
            if (!r_active && s_valid) begin
```

```
                // Load new wide word
                r_buffer <= s_data;
                r_active <= 1'b1;
                r_count <= '0;
            end else if (r_active && m_ready) begin
                if (r_count == RATIO - 1) begin
                    r_active <= 1'b0;
                end else begin
                    r_count <= r_count + 1;
                end
            end
        end
    end

    // Output from buffer slice
    assign m_valid = r_active;
    assign m_data = r_buffer[r_count * OUT_WIDTH +: OUT_WIDTH];
    assign m_last = r_active && (r_count == RATIO - 1);
    assign s_ready = !r_active;

endmodule
```

16.4 Strobe Handling

16.4.1 Upsize Strobe Packing

```
// Pack 8-byte strobes into 64-byte strobes
logic [7:0] in_strb;   // 64-bit input
logic [63:0] out_strb; // 512-bit output

always_ff @(posedge clk) begin
    if (s_valid && s_ready) begin
        out_strb[r_count * 8 +: 8] <= in_strb;
    end
end
```

16.4.2 Downsize Strobe Splitting

```
// Split 64-byte strobes into 8-byte strobes
logic [63:0] in_strb;  // 512-bit input
logic [7:0] out_strb;  // 64-bit output

assign out_strb = in_strb[r_count * 8 +: 8];
```

16.5 Resource Utilization

16.5.1 Upsize Converter (64 to 512)

```
Registers: ~520 (buffer + control)
Logic: ~200 LEs (packing + control)
```

16.5.2 Downsize Converter (512 to 64)

```
Registers: ~520 (buffer + control)
Logic: ~150 LEs (selection + control)
```

16.6 Related Documentation

- Width Conversion Block - Block-level description
- APB Converters - Protocol conversion

17 APB Converters

17.1 Overview

APB converters transform AXI4 transactions into APB protocol. This is necessary when high-performance masters need to access low-speed peripherals.

17.2 Conversion Requirements

17.2.1 Protocol Differences

Table 5.1: AXI4 vs APB Protocol Comparison

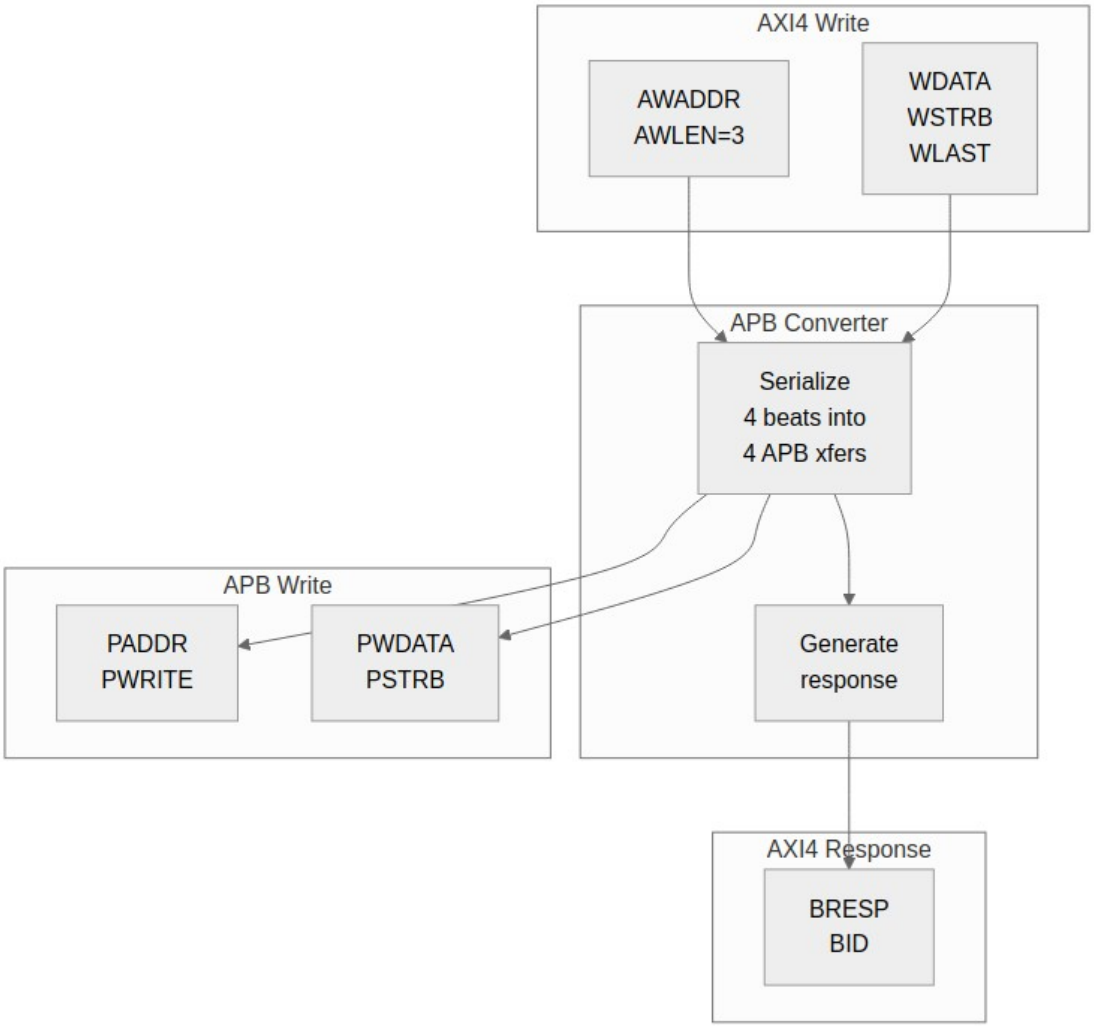
Feature	AXI4	APB
Channels	5 (AW, W, B, AR, R)	1 (combined)
Bursts	Up to 256 beats	None (single)
Pipelining	Yes	No
Min cycles	1 per beat	2 per transfer

17.2.2 Conversion Strategy

1. Split AXI4 bursts into individual APB transfers
2. Serialize AW+W into APB write sequence
3. Convert AR into APB read sequence
4. Generate AXI4 responses from APB completions

17.3 Write Conversion

17.3.1 Figure 5.3: AXI4 Write to APB Write



AXI4 to APB Write

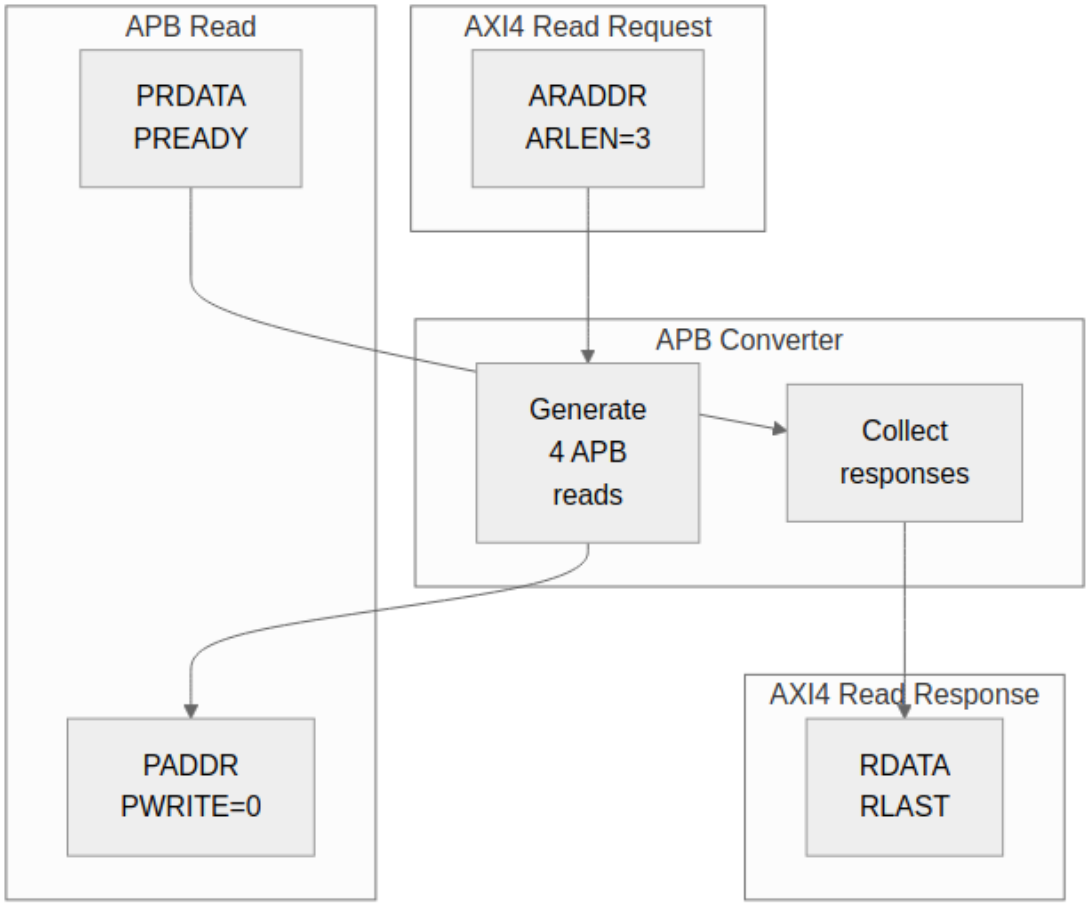
17.3.2 State Machine

```
typedef enum logic [2:0] {
    IDLE,
    AW_ACCEPT,
    W_ACCEPT,
    APB_SETUP,
    APB_ACCESS,
    B_GENERATE
} apb_wr_state_t;
```

```
apb_wr_state_t r_state;

always_ff @(posedge clk or negedge rst_n) begin
    if (!rst_n) begin
        r_state <= IDLE;
    end else begin
        case (r_state)
            IDLE: if (awvalid) r_state <= AW_ACCEPT;

            AW_ACCEPT: begin
                if (awready) r_state <= W_ACCEPT;
            end

            W_ACCEPT: begin
                if (wvalid && wready) r_state <= APB_SETUP;
            end

            APB_SETUP: r_state <= APB_ACCESS;

            APB_ACCESS: begin
                if (pready) begin
                    if (r_beat_count == r_awlen)
                        r_state <= B_GENERATE;
                    else
                        r_state <= W_ACCEPT;  // Next beat
                end
            end

            B_GENERATE: begin
                if (bready) r_state <= IDLE;
            end
        endcase
    end
end
```

17.4 Read Conversion

17.4.1 Figure 5.4: AXI4 Read to APB Read



AXI4 to APB Read

17.4.2 Read State Machine

```
typedef enum logic [2:0] {
    IDLE,
    AR_ACCEPT,
    APB_SETUP,
    APB_ACCESS,
    R_GENERATE
} apb_rd_state_t;
```

17.5 Burst Handling

17.5.1 Burst to Single Conversion

```
AXI4 Burst (AWLEN=7, 8 beats):
  Beat 0: AWADDR + 0×SIZE
  Beat 1: AWADDR + 1×SIZE
  Beat 2: AWADDR + 2×SIZE
  ...
  Beat 7: AWADDR + 7×SIZE

APB Sequence:
  Transfer 0: PADDR = AWADDR + 0×SIZE
  Transfer 1: PADDR = AWADDR + 1×SIZE
  ...
  Transfer 7: PADDR = AWADDR + 7×SIZE
```

17.5.2 Address Calculation

```
// Calculate APB address for each beat
logic [ADDR_WIDTH-1:0] apb_addr;
logic [7:0] beat_count;

always_comb begin
    case (r_burst_type)
        FIXED: apb_addr = r_base_addr;
        INCR:  apb_addr = r_base_addr + (beat_count << r_size);
        WRAP:  apb_addr = wrap_address(r_base_addr, beat_count,
r_size, r_len);
    endcase
end
```

17.6 Error Handling

17.6.1 APB Error to AXI4 Response

Table 5.2: APB to AXI4 Error Mapping

APB PSLVERR	AXI4 Response
0 (OK)	OKAY (2'b00)
1 (Error)	SLVERR (2'b10)

17.6.2 Error Propagation

```
// Latch first error in burst
logic r_error_seen;

always_ff @(posedge clk) begin
```

```
    if (r_state == APB_ACCESS && pready && pslverr)
        r_error_seen <= 1'b1;
    if (r_state == IDLE)
        r_error_seen <= 1'b0;
end

// Final response reflects any errors
assign bresp = r_error_seen ? 2'b10 : 2'b00;
```

17.7 Performance Considerations

17.7.1 Throughput Impact

```
AXI4 alone: 1 beat per cycle (peak)
AXI4 to APB: 1 beat per 2+ cycles (minimum)

Throughput reduction: ~50% or more
```

17.7.2 Latency Impact

```
AXI4 read: 1-2 cycles (typical)
AXI4-to-APB read: 4+ cycles minimum
  - 1 cycle: AR accept
  - 1 cycle: APB setup
  - 1+ cycles: APB access (PREADY)
  - 1 cycle: R generate
```

17.8 Resource Utilization

17.8.1 APB Converter Resources

```
Logic Elements: ~300-400 LEs
Registers: ~100-150 regs
Block RAM: 0

Breakdown:
- State machine: ~50 LEs, ~20 regs
- Address calc: ~100 LEs, ~50 regs
- Beat counter: ~30 LEs, ~8 regs
- Data buffering: ~100 LEs, ~DATA_WIDTH regs
```

17.9 Related Documentation

- Protocol Conversion Block - Block description
- Width Converters - Data width conversion

18 Module Structure

18.1 Generated RTL Overview

Bridge generator creates parameterized SystemVerilog modules from configuration files. The generated RTL follows a consistent structure for all topologies.

18.2 Top-Level Module

18.2.1 Module Declaration

```
module bridge_4x3 #(
    parameter int NUM_MASTERS = 4,
    parameter int NUM_SLAVES = 3,
    parameter int ADDR_WIDTH = 32,
    parameter int DATA_WIDTH = 64,
    parameter int ID_WIDTH = 4
) (
    // Clock and reset
    input  logic aclk,
    input  logic aresetn,

    // Master interfaces (M0-M3)
    // ... master port signals ...

    // Slave interfaces (S0-S2)
    // ... slave port signals ...
);
```

18.2.2 Port Organization

```
Module Ports:
├── Clock/Reset
│   ├── aclk
│   └── aresetn
├── Master Ports (per master)
│   ├── AW Channel (if rw or wr)
│   ├── W Channel (if rw or wr)
│   ├── B Channel (if rw or wr)
│   ├── AR Channel (if rw or rd)
│   └── R Channel (if rw or rd)
└── Slave Ports (per slave)
    ├── AW Channel
    ├── W Channel
    ├── B Channel
    ├── AR Channel
    └── R Channel
```

18.3 Internal Structure

18.3.1 Component Instantiation

```
// Generated module internal structure

// Master adapters (per master)
master_adapter_rw u_m0_adapter (...);
master_adapter_wr u_m1_adapter (...);
master_adapter_rd u_m2_adapter (...);
master_adapter_rw u_m3_adapter (...);

// Address decoders (per master)
address_decoder u_m0_decoder (...);
address_decoder u_m1_decoder (...);
address_decoder u_m2_decoder (...);
address_decoder u_m3_decoder (...);

// Per-slave arbiters
arbiter_aw u_s0_aw_arb (...);
arbiter_ar u_s0_ar_arb (...);
arbiter_aw u_s1_aw_arb (...);
arbiter_ar u_s1_ar_arb (...);
arbiter_aw u_s2_aw_arb (...);
arbiter_ar u_s2_ar_arb (...);

// Width converters (as needed)
width_upsize_64_512 u_m0_s0_upsizer (...);

// Protocol converters (as needed)
axi4_to_apb u_s2_apb_conv (...);

// Response routing
response_router u_resp_router (...);
```

18.4 Signal Naming Convention

18.4.1 Master-Side Signals (External)

```
{prefix}_aw{signal}    - Write address channel
{prefix}_w{signal}     - Write data channel
{prefix}_b{signal}     - Write response channel
{prefix}_ar{signal}    - Read address channel
{prefix}_r{signal}     - Read data channel

Example (prefix = "cpu_m_axi"):
```

```
  cpu_m_axi_awvalid
  cpu_m_axi_awready
  cpu_m_axi_awaddr
  cpu_m_axi_wdata
  cpu_m_axi_bvalid
```

18.4.2 Slave-Side Signals (External)

```
{prefix}_aw{signal}    - Write address channel
{prefix}_w{signal}     - Write data channel
{prefix}_b{signal}     - Write response channel
{prefix}_ar{signal}    - Read address channel
{prefix}_r{signal}     - Read data channel

Example (prefix = "ddr_s_axi"):
  ddr_s_axi_awvalid
  ddr_s_axi_awready
  ddr_s_axi_awaddr
  ddr_s_axi_wdata
  ddr_s_axi_bvalid
```

18.4.3 Internal Signals

```
// Crossbar internal signals
xbar_m{N}_aw_{signal}  - Master N to crossbar AW
xbar_m{N}_ar_{signal}  - Master N to crossbar AR
xbar_s{N}_aw_{signal}  - Crossbar to slave N AW
xbar_s{N}_ar_{signal}  - Crossbar to slave N AR

// Arbitration signals
grant_aw_s{N}[M-1:0]   - AW grants for slave N
grant_ar_s{N}[M-1:0]   - AR grants for slave N

// ID tracking signals
id_table_s{N}[...]     - ID table for slave N
```

18.5 Generated File Structure

18.5.1 Single-File Output

```
bridge_{name}.sv
├── Module declaration
├── Parameter section
├── Port declarations
├── Internal signal declarations
├── Master adapter instances
├── Address decoder instances
├── Arbiter instances
├── Converter instances
```

```
├── Response router instance
└── Debug signals (optional)
```

18.5.2 Multi-File Output (Optional)

```
bridge_{name}/
├── bridge_{name}.sv        - Top-level wrapper
├── master_adapter_m0.sv    - Master 0 adapter
├── master_adapter_m1.sv    - Master 1 adapter
├── address_decoder.sv      - Shared decoder
├── arbiter_aw.sv           - AW arbiter
├── arbiter_ar.sv           - AR arbiter
├── width_upsize_64_512.sv  - Width converter
├── axi4_to_apb.sv          - Protocol converter
└── response_router.sv      - Response routing
```

18.6 Parameterization

18.6.1 Compile-Time Parameters

```
// Core parameters
parameter int NUM_MASTERS = 4;
parameter int NUM_SLAVES = 3;
parameter int ADDR_WIDTH = 32;
parameter int DATA_WIDTH = 64;
parameter int ID_WIDTH = 4;

// Derived parameters
localparam int BID_WIDTH = $clog2(NUM_MASTERS);
localparam int TOTAL_ID_WIDTH = ID_WIDTH + BID_WIDTH;
localparam int STRB_WIDTH = DATA_WIDTH / 8;
```

18.6.2 Per-Port Parameters

```
// Generated per-port widths
localparam int M0_DATA_WIDTH = 64;
localparam int M1_DATA_WIDTH = 256;
localparam int S0_DATA_WIDTH = 512;
localparam int S1_DATA_WIDTH = 32;  // APB
```

18.7 Related Documentation

- Signal Naming - Detailed naming conventions
- Generator Usage - How to run the generator

19 Signal Naming

19.1 Naming Convention Overview

Bridge uses consistent signal naming to enable pattern-based verification and clear integration.

19.2 External Port Naming

19.2.1 Pattern

```
{prefix}_{channel}{signal}
```

19.2.2 Components

Component	Description	Example
prefix	User-defined port prefix	cpu_m_axi, ddr_s_axi
channel	AXI4 channel code	aw, w, b, ar, r
signal	Signal within channel	valid, ready, addr, data

19.2.3 Master Port Examples

```
// Master with prefix "cpu_m_axi"
input  logic        cpu_m_axi_awvalid,
output logic        cpu_m_axi_awready,
input  logic [31:0] cpu_m_axi_awaddr,
input  logic [3:0]  cpu_m_axi_awid,
input  logic [7:0]  cpu_m_axi_awlen,
input  logic [2:0]  cpu_m_axi_awsize,
input  logic [1:0]  cpu_m_axi_awburst,
input  logic        cpu_m_axi_awlock,
input  logic [3:0]  cpu_m_axi_awcache,
input  logic [2:0]  cpu_m_axi_awprot,
input  logic [3:0]  cpu_m_axi_awqos,
input  logic        cpu_m_axi_awuser,
```

19.2.4 Slave Port Examples

```
// Slave with prefix "ddr_s_axi"
output logic        ddr_s_axi_awvalid,
input  logic        ddr_s_axi_awready,
output logic [31:0] ddr_s_axi_awaddr,
output logic [5:0]  ddr_s_axi_awid,   // Extended ID
output logic [7:0]  ddr_s_axi_awlen,
```

```
output logic [2:0]  ddr_s_axi_awsize,
// ...
```

19.3 APB Port Naming

19.3.1 Pattern

```
{prefix}p{signal}
```

19.3.2 APB Signal Examples

```
// APB slave with prefix "uart_apb_"
output logic        uart_apb_psel,
output logic        uart_apb_penable,
output logic [11:0] 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,
```

19.4 Internal Signal Naming

19.4.1 Crossbar Signals

```
// Master to crossbar
logic        xbar_m0_awvalid;
logic        xbar_m0_awready;
logic [31:0] xbar_m0_awaddr;

// Crossbar to slave
logic        xbar_s0_awvalid;
logic        xbar_s0_awready;
logic [31:0] xbar_s0_awaddr;
```

19.4.2 Arbitration Signals

```
// Request/grant matrices
logic [NUM_MASTERS-1:0] aw_request_s0;  // AW requests to slave 0
logic [NUM_MASTERS-1:0] aw_grant_s0;    // AW grants from slave 0

// Arbiter state
logic aw_grant_active_s0;               // Grant locked
logic [$clog2(NUM_MASTERS)-1:0] aw_last_grant_s0;  // Round-robin
state
```

19.4.3 ID Signals

```
// Extended IDs (internal)
logic [TOTAL_ID_WIDTH-1:0] xbar_m0_awid;  // BID + external ID
logic [TOTAL_ID_WIDTH-1:0] xbar_s0_bid;   // Response with BID

// External IDs (to master)
logic [ID_WIDTH-1:0] cpu_m_axi_bid;       // Stripped ID
```

19.5 Debug Signal Naming

19.5.1 Pattern

```
dbg_{category}_{description}
```

19.5.2 Debug Signal Examples

```
// Arbitration debug
output logic [NUM_MASTERS-1:0] dbg_aw_grant_s0,
output logic [NUM_MASTERS-1:0] dbg_ar_grant_s0,

// Transaction debug
output logic [7:0] dbg_outstanding_m0,
output logic [7:0] dbg_outstanding_m1,

// State machine debug
output logic [1:0] dbg_aw_state_s0,
output logic [1:0] dbg_ar_state_s0,
```

19.6 Verification Pattern Matching

19.6.1 Factory Pattern Support

Signal naming enables automatic BFM connection:

```
# CocoTB/GAXI pattern matching
master = AXI4Master(
    dut=dut,
    prefix="cpu_m_axi_",  # Matches all cpu_m_axi_* signals
    clock=dut.aclk
)
```

19.6.2 Pattern Rules

1. Prefix must end with underscore or be directly followed by channel code
2. Channel codes are lowercase (aw, w, b, ar, r)
3. Signal names match AXI4 specification (valid, ready, addr, data, etc.)

19.7 Related Documentation

- Module Structure - Overall RTL organization
- Verification - Pattern-based testing

20 Test Strategy

20.1 Overview

Bridge verification uses CocoTB-based testing with parameterized configurations. The test infrastructure supports all bridge topologies and protocol combinations.

20.2 Test Categories

20.2.1 Unit Tests

Per-block verification:

```
Unit Test Coverage:
├── Master Adapter Tests
│   ├── ID extension verification
│   ├── Channel routing
│   └── Backpressure handling
├── Address Decoder Tests
│   ├── Address mapping
│   ├── Multi-region support
│   └── Default slave routing
├── Arbiter Tests
│   ├── Round-robin fairness
│   ├── Grant/lock behavior
│   └── Multi-master contention
└── Converter Tests
    ├── Width up/down sizing
    ├── APB protocol conversion
    └── Burst handling
```

20.2.2 Integration Tests

Full bridge verification:

```
Integration Test Matrix:
├── 2x2 Configuration
│   ├── Basic read/write
│   ├── Concurrent transactions
│   └── ID collision avoidance
├── 4x4 Configuration
```

```
│   ├── Full arbitration coverage
│   ├── Width conversion paths
│   └── Mixed protocol targets
└── NxM Configurations
    ├── Asymmetric topologies
    ├── Channel-specific masters
    └── Protocol mixing
```

20.3 Test Parameterization

20.3.1 Configuration Matrix

Table 7.1: Test Parameter Matrix

Parameter	Test Values	Purpose
NUM_MASTERS	2, 4, 8	Scalability
NUM_SLAVES	2, 3, 4	Routing coverage
DATA_WIDTH	32, 64, 128, 256	Width paths
ID_WIDTH	4, 6, 8	ID space
OUTSTANDING	4, 8, 16	Depth stress

20.3.2 Protocol Combinations

```
# Test protocol matrix
PROTOCOL_COMBOS = [
    {"masters": ["axi4"], "slaves": ["axi4"]},
    {"masters": ["axi4"], "slaves": ["axi4", "apb"]},
    {"masters": ["axi4", "axil"], "slaves": ["axi4", "apb"]},
]
```

20.4 Factory-Based Testing

20.4.1 BFM Instantiation

```
from CocoTBFramework.components.gaxi import GAXIMaster, GAXISlave

class BridgeTB(TBBase):
    def __init__(self, dut):
        super().__init__(dut)

        # Create masters using factory pattern
        self.masters = []
        for i in range(NUM_MASTERS):
            prefix = f"m{i}_axi_"
            master = GAXIMaster(
```

```
                dut=dut,
                prefix=prefix,
                clock=dut.aclk,
                reset=dut.aresetn
            )
            self.masters.append(master)

        # Create slaves
        self.slaves = []
        for i in range(NUM_SLAVES):
            prefix = f"s{i}_axi_"
            slave = GAXISlave(
                dut=dut,
                prefix=prefix,
                clock=dut.aclk,
                reset=dut.aresetn
            )
            self.slaves.append(slave)
```

20.4.2 Transaction Generation

```
async def test_concurrent_writes(self):
    """Test multiple masters writing simultaneously."""

    # Generate transactions for each master
    tasks = []
    for i, master in enumerate(self.masters):
        addr = 0x1000 + (i * 0x100)
        data = [0xDEAD0000 + i] * 4
        task = cocotb.start_soon(
            master.write(addr=addr, data=data, size=2)
        )
        tasks.append(task)

    # Wait for all to complete
    for task in tasks:
        await task

    # Verify arbitration occurred correctly
    self.verify_arbitration_order()
```

20.5 Coverage Goals

20.5.1 Functional Coverage

```
Coverage Targets:
├── Protocol Coverage
│   ├── All burst types (FIXED, INCR, WRAP)
```

```
│   ├── All sizes (1, 2, 4, 8, ... bytes)
│   └── All response types
├── Routing Coverage
│   ├── Every master to every slave path
│   ├── Default slave routing
│   └── Error response paths
├── Arbitration Coverage
│   ├── All grant combinations
│   ├── Priority scenarios
│   └── Lock sequences
└── Corner Cases
    ├── Maximum outstanding
    ├── ID exhaustion
    └── Backpressure saturation
```

20.5.2 Code Coverage

```
Code Coverage Targets:
├── Line Coverage: >95%
├── Branch Coverage: >90%
├── FSM State Coverage: 100%
└── Toggle Coverage: >85%
```

20.6 Test Execution

20.6.1 Running Tests

```bash
# Run all bridge tests
cd projects/components/bridge/dv/tests
pytest -v

# Run specific configuration
pytest test_bridge_4x4.py -v

# Run with coverage
pytest --cov=bridge -v

# Run with waveforms
WAVES=1 pytest test_bridge_2x2.py -v
```

20.6.2 Test Levels

Table 7.2: Test Level Definitions

Level	Duration	Coverage	Use Case
basic	~30s	Smoke test	Quick verification

Level	Duration	Coverage	Use Case
medium	~90s	Core paths	Development
full	~180s	Comprehensive	Pre-commit

20.7 Related Documentation

- Debug Guide - Debugging failed tests
- Signal Naming - Pattern matching for BFMs

21 Debug Guide

21.1 Overview

This guide provides debugging strategies for Bridge verification failures. Common issues and their resolution approaches are documented.

21.2 Debug Signal Access

21.2.1 Enabling Debug Outputs

```
# Enable debug signals in test
@cocotb.test()
async def test_with_debug(dut):
    # Access debug signals
    grant_s0 = dut.dbg_aw_grant_s0.value
    outstanding_m0 = dut.dbg_outstanding_m0.value
    state_s0 = dut.dbg_aw_state_s0.value
```

21.2.2 Debug Signal Categories

Table 7.3: Debug Signal Categories

Category	Signals	Purpose
Arbitration	`dbg_*_grant_s*`	Grant state per slave
Outstanding	`dbg_outstanding_m*`	Transaction depth per master
State	`dbg_*_state_s*`	FSM states per channel
ID Tracking	`dbg_id_table_*`	CAM contents

21.3 Common Issues

21.3.1 Issue: Transaction Timeout

Symptoms: - Test hangs waiting for response - `TimeoutError` after configured timeout

Debug Steps:

```
# 1. Check if request was accepted
await RisingEdge(dut.aclk)
print(f"AWVALID: {dut.m0_axi_awvalid.value}")
print(f"AWREADY: {dut.m0_axi_awready.value}")

# 2. Check arbitration state
print(f"AW Grant S0: {dut.dbg_aw_grant_s0.value}")

# 3. Check outstanding count
print(f"Outstanding M0: {dut.dbg_outstanding_m0.value}")

# 4. Check slave response
print(f"BVALID: {dut.s0_axi_bvalid.value}")
print(f"BREADY: {dut.s0_axi_bready.value}")
```

Common Causes: - Arbiter locked by another master - Outstanding limit reached - Slave not responding - ID table full

21.3.2 Issue: Wrong Response Routing

Symptoms: - Response arrives at wrong master - BID mismatch errors

Debug Steps:

```
# Check ID extension
print(f"Outgoing AWID: {dut.xbar_m0_awid.value}")
print(f"Slave received ID: {dut.s0_axi_awid.value}")

# Check response ID
print(f"Slave BID: {dut.s0_axi_bid.value}")
print(f"Routed to master: {dut.dbg_resp_master.value}")
```

Common Causes: - ID extension incorrect - CAM lookup failure - ID width mismatch

21.3.3 Issue: Data Corruption

Symptoms: - Read data doesn't match written data - WSTRB/data alignment issues

Debug Steps:

```
# Check width conversion
print(f"Master data width: {dut.M0_DATA_WIDTH.value}")
print(f"Slave data width: {dut.S0_DATA_WIDTH.value}")

# Check strobe packing
print(f"Master WSTRB: {dut.m0_axi_wstrb.value}")
print(f"Slave WSTRB: {dut.s0_axi_wstrb.value}")

# Check address alignment
print(f"AWADDR: {dut.m0_axi_awaddr.value}")
print(f"AWSIZE: {dut.m0_axi_awsize.value}")
```

Common Causes: - Width converter byte lane error - Unaligned access handling - Burst boundary crossing

21.4 Waveform Analysis

21.4.1 Generating Waveforms

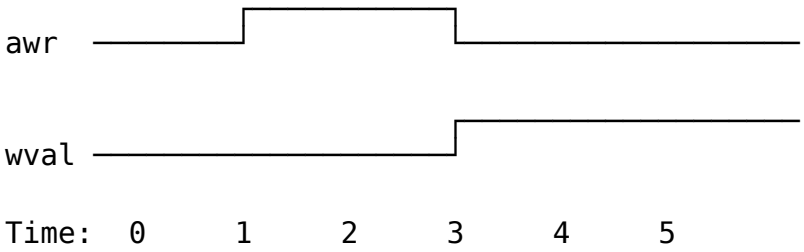
```
# Enable VCD dump
WAVES=1 pytest test_bridge_2x2.py::test_basic -v

# View with GTKWave
gtkwave sim_build/dump.vcd
```

21.4.2 Key Signals to Observe

```
Waveform Signal Groups:
├── Clock/Reset
│   └── aclk, aresetn
├── Master 0 AW Channel
│   └── m0_axi_awvalid, m0_axi_awready, m0_axi_awaddr, m0_axi_awid
├── Slave 0 AW Channel
│   └── s0_axi_awvalid, s0_axi_awready, s0_axi_awaddr, s0_axi_awid
├── Arbitration
│   └── dbg_aw_grant_s0, dbg_ar_grant_s0
└── Response
    └── s0_axi_bvalid, m0_axi_bvalid, dbg_resp_master
```

21.4.3 Timing Analysis

```
Expected Transaction Timing:

aclk ┌─┐ ┌─┐ ┌─┐ ┌─┐ ┌─┐ ┌─┐ ┌─
     └─┘ └─┘ └─┘ └─┘ └─┘ └─┘ └─

awv  ┌─────────────────┐
     └─────────────────────────
```



21.5 Assertion Failures

21.5.1 Built-in Assertions

```
// Protocol assertions in generated RTL
assert property (@(posedge aclk) disable iff (!aresetn)
    s_awvalid && !s_awready |=> s_awvalid
) else $error("AW handshake violation");

assert property (@(posedge aclk) disable iff (!aresetn)
    s_wvalid && s_wlast |-> s_awid == r_current_awid
) else $error("W data ID mismatch");
```

21.5.2 Handling Assertion Failures

1. **Locate assertion in RTL** - Search for error message
2. **Check signal history** - Review 10-20 cycles before failure
3. **Identify root cause** - Usually protocol or timing issue
4. **Fix test or RTL** - Depending on where bug exists

21.6 Performance Debugging

21.6.1 Throughput Issues

```
# Measure transaction throughput
start_time = cocotb.utils.get_sim_time('ns')

for i in range(100):
    await master.write(addr=0x1000 + i*4, data=[i])

end_time = cocotb.utils.get_sim_time('ns')
throughput = 100 / (end_time - start_time) * 1e9  # tx/sec
print(f"Throughput: {throughput:.2f} transactions/sec")
```

21.6.2 Latency Issues

```
# Measure single transaction latency
start = cocotb.utils.get_sim_time('ns')
await master.read(addr=0x1000)
```

```
latency = cocotb.utils.get_sim_time('ns') - start
print(f"Read latency: {latency} ns")
```

21.7 Related Documentation

- Test Strategy - Overall test approach
- Arbiter FSMs - FSM state details
- ID Tracking - ID table operation

22 Bridge Hardware Architecture Specification Index

Version: 1.0 **Date:** 2026-01-03 **Purpose:** High-level hardware architecture specification for Bridge component

---

22.1 Document Organization

Note: All chapters linked below for automated document generation.

22.1.1 Front Matter

- Document Information

22.1.2 Chapter 1: Introduction

- Purpose and Scope
- Document Conventions
- Definitions and Acronyms

22.1.3 Chapter 2: System Overview

- Use Cases
- Key Features
- System Context

22.1.4 Chapter 3: Architecture

- Block Diagram
- Data Flow
- Protocol Support

22.1.5 Chapter 4: Interfaces

- AXI4 Interface Overview

- APB Interface Overview
- Clock and Reset

22.1.6 Chapter 5: Performance

- Throughput Characteristics
- Latency Analysis
- Resource Estimates

22.1.7 Chapter 6: Integration

- System Requirements
- Parameter Configuration
- Verification Strategy

22.2 Related Documentation

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

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

23 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

23.1 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.

23.2 Design Philosophy

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

23.2.1 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

Parameter	Hard Limit	Rationale
		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

23.2.2 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.

23.3 ⚠️ 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.

23.4 1. Product Overview

23.4.1 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

23.4.2 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

23.4.3 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)

23.4.4 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

---

23.5 2. Architecture Overview

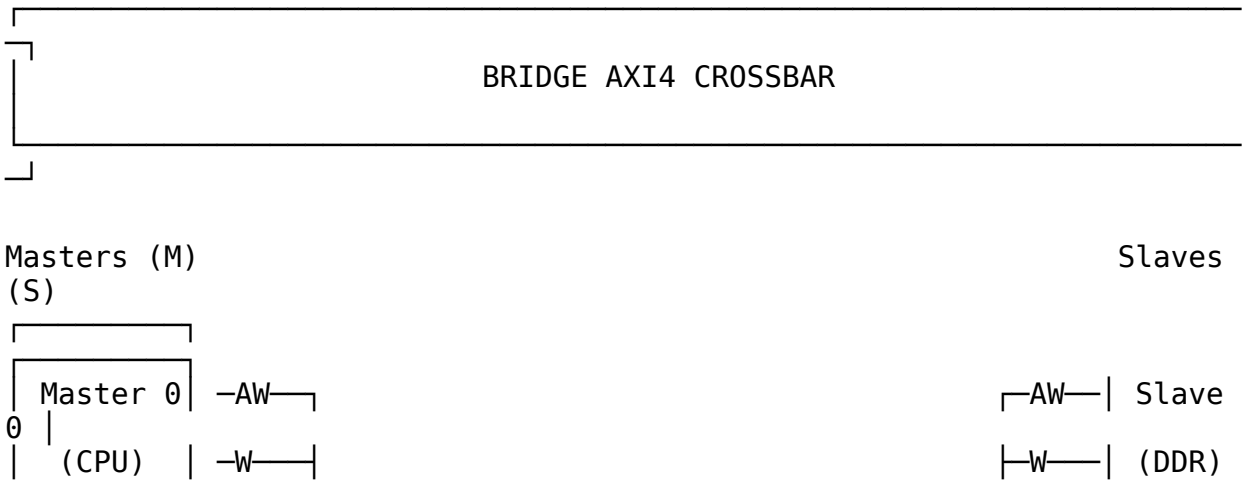
23.5.1 2.1 AXI4 vs AXIS vs APB

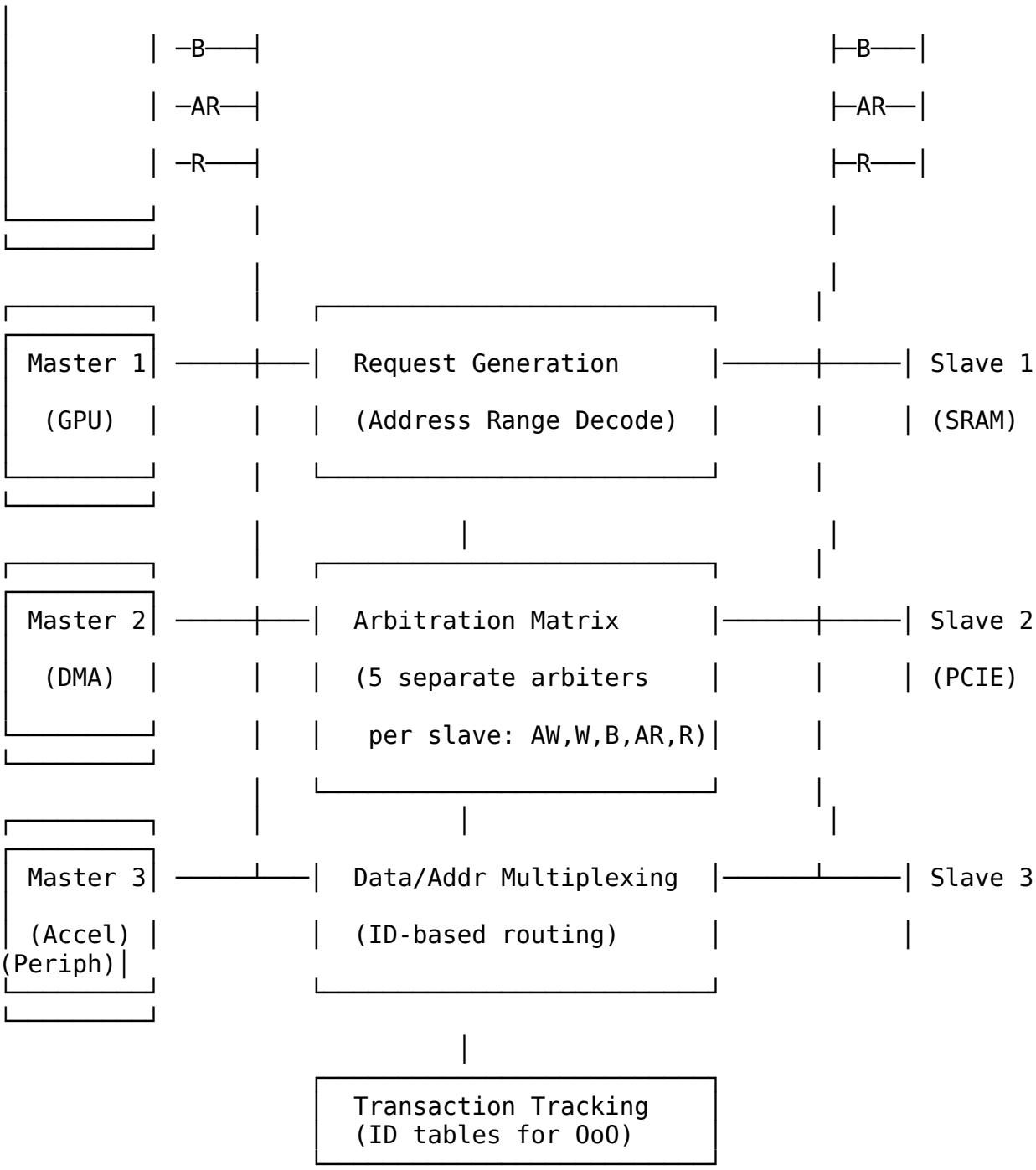
Feature	APB	AXI-Stream (Delta)	AXI4 (Bridge)
Protocol Type	Simple register bus	Streaming data	Memory-mapped burst
Channels	1 (address + data)	1 (data stream)	5 (AW, W, B, AR, R)
Request Generation	Address range	TDEST decode	Address range decode

Feature	APB	AXI-Stream (Delta)	AXI4 (Bridge)
	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

23.5.2 2.2 Block Diagram





23.5.3 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

---

23.6 3. Functional Requirements

23.6.1 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

23.6.2 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
}
```

23.6.3 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)

23.6.4 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)

23.7 4. Non-Functional Requirements

23.7.1 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

23.7.2 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

Resource	Flat Crossbar	Notes
DSP	0	No arithmetic operations

Scaling: ~150 LUTs per M×S connection

23.7.3 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

---

23.8 5. Interface Specifications

23.8.1 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];
```

23.8.2 5.2 AXI4 Slave Interfaces (S × 5 channels)

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

23.8.3 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
```

23.9 6. Performance Modeling

23.9.1 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
```

23.9.2 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
```

23.9.3 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.

23.10 7. Generator Architecture

23.10.1 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
```

23.10.2 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
```

23.11 8. Comparison with APB and Delta

23.11.1 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

23.11.2 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)

23.11.3 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-order	No	No	Yes (ID tables) ★
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

23.12 9. Use Cases

23.12.1 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)

23.12.2 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

23.12.3 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

---

23.13 10. Testing Strategy

23.13.1 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

23.13.2 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

23.13.3 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

---

23.14 11. Documentation Plan

23.14.1 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

23.14.2 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

23.14.3 11.3 Code Generation

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

23.14.4 11.4 Verification

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

23.15 12. Success Metrics

23.15.1 12.1 Functional Completeness

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

23.15.2 12.2 Performance Targets

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

23.15.3 12.3 Educational Value

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

23.15.4 12.4 Reusability

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

23.16 12. Attribution and Contribution Guidelines

23.16.1 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.

23.17 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

23.18 13. Future Enhancements

23.18.1 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

23.18.2 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

23.19 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

23.20 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

---

23.21 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

---

23.22 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 

24 Claude Code Guide: Bridge Subsystem

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

---

24.1 🚨 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.

24.2 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

---

24.3 Target Architecture: Intelligent Width-Aware Routing

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

24.3.1 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.

24.3.2 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

24.3.3 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
```

24.3.4 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

24.3.5 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)

24.4 🚨 CRITICAL RULE #0: RTL Regeneration Requirements

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

24.4.1 The Golden Rule

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

24.4.2 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**

24.4.3 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
```

24.4.4 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/`

24.4.5 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
```

24.4.6 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

24.4.7 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**.

24.4.8 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"

24.5 ⚠️ MANDATORY: Project Organization Pattern

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

24.5.1 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
```

24.5.2 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/`).

24.5.3 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,
        # ...
    )
```

24.6 Critical Rules for This Subsystem

24.6.1 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.

24.6.2 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

---

24.6.3 Rule #1: All Testbenches Inherit from TBBase

Every testbench class MUST inherit from TBBase:

```
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)

24.6.4 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

---

24.6.5 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.

24.6.6 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

---

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

24.7.1 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

24.7.2 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

24.7.3 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)

24.7.4 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

24.7.5 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)

24.7.6 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
```

24.7.7 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

24.7.8 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.

24.8 Bridge Architecture Quick Reference

24.8.1 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
);
```

24.8.2 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

24.8.3 Address Map

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

24.9 Test Organization

24.9.1 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
```

24.9.2 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

---

24.10 Common User Questions and Responses

24.10.1 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

24.10.2 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

24.10.3 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
```

24.11 Anti-Patterns to Avoid

24.11.1 ✗ 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
```

24.11.2 ✗ 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)
```

24.11.3 ✗ 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
```

24.12 Quick Reference

24.12.1 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/
```

24.12.2 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
```

24.13 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

---

24.14 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