RTL Design Sherpa

APB Crossbar Hardware Architecture Specification 1.0

January 3, 2026

Table of Contents

List of Figures

List of Tables

Apb Xbar Has Index

Generated: 2026-01-04

RTL Design Sherpa · Learning Hardware Design Through Practice GitHub · Documentation Index · MIT License

---

APB Crossbar Hardware Architecture Specification Index

Version: 1.0 **Date:** 2026-01-03 **Purpose:** High-level hardware architecture specification for APB Crossbar component

---

Document Organization

Note: All chapters linked below for automated document generation.

Front Matter

- Document Information

Chapter 1: Introduction

- Purpose and Scope
- Document Conventions
- Definitions and Acronyms

Chapter 2: System Overview

- Use Cases
- Key Features
- System Context

Chapter 3: Architecture

- Block Diagram
- Data Flow
- Address Mapping

Chapter 4: Interfaces

- Master-Side APB Interfaces
- Slave-Side APB Interfaces
- Clock and Reset

Chapter 5: Performance

- Throughput Characteristics
- Latency Analysis
- Resource Estimates

Chapter 6: Integration

- System Requirements
- Parameter Configuration
- Verification Strategy

Related Documentation

- **APB Crossbar 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

RTL Design Sherpa · Learning Hardware Design Through Practice GitHub · Documentation Index · MIT License

---

Document Information

APB Crossbar Hardware Architecture Specification

Document Number: APB-XBAR-HAS-001 **Version:** 1.0 **Status:** Production Ready **Classification:** Open Source - Apache 2.0 License

---

Document Purpose

This Hardware Architecture Specification (HAS) provides a high-level architectural overview of the APB Crossbar component. It describes the system-level design, external interfaces, performance characteristics, and integration requirements without detailing internal implementation specifics.

Target Audience: - System architects evaluating APB Crossbar for SoC integration - Hardware engineers planning peripheral interconnect design - Software engineers developing drivers and firmware - Verification engineers planning system-level testing

Companion Documents: - APB Crossbar Micro-Architecture Specification (MAS) - Detailed block-level implementation - APB Crossbar Product Requirements Document (PRD) - Requirements and rationale

---

References

Reference Documents

ID	Document	Description
[REF-1]	APB Crossbar MAS v1.0	Micro-Architecture Specification
[REF-2]	APB Crossbar PRD	Product Requirements Document
[REF-3]	ARM AMBA APB	APB Protocol Specification (IHI 0024C)

Terminology

Terminology and Definitions

Term	Definition
APB	Advanced Peripheral Bus - ARM AMBA low-power peripheral bus
Arbiter	Logic that selects between competing requests using round-robin
Crossbar	NxM interconnect fabric connecting multiple

Term	Definition
	masters to multiple slaves
Master	APB initiator device that initiates read/write transactions
PSEL	APB peripheral select signal indicating slave selection
PENABLE	APB enable signal indicating data phase of transaction
PREADY	APB ready signal indicating slave response completion
Slave	APB target device that responds to master transactions
Transaction	Complete APB read or write operation (setup + data phase)

Revision History

Document Revision History

Version	Date	Author	Description
1.0	2026-01-03	seang	Initial HAS release

Last Updated: 2026-01-03

RTL Design Sherpa · Learning Hardware Design Through Practice GitHub · Documentation Index · MIT License

---

Purpose and Scope

Document Purpose

This Hardware Architecture Specification (HAS) describes the APB Crossbar component at a system integration level. It provides sufficient detail for:

- **System architects** to evaluate the component for SoC integration
- **Hardware engineers** to plan interconnect topology and address mapping
- **Software engineers** to understand register access paths and address spaces
- **Verification engineers** to develop system-level test plans

Component Overview

The APB Crossbar is a parametric MxN interconnect fabric that connects multiple APB masters to multiple APB slaves. It provides:

- **Automatic address-based routing** to appropriate slave
- **Fair round-robin arbitration** when multiple masters access the same slave
- **Zero-bubble throughput** for back-to-back transactions
- **Scalable configuration** from 1x1 to 16x16

Document Scope

In Scope

- System-level architecture and block organization
- External interface specifications (all APB signals)
- Address mapping and routing strategy
- Arbitration policy and fairness guarantees
- Performance characteristics (latency, throughput)
- Integration requirements and parameters

Out of Scope

- Internal RTL implementation details (see MAS)
- Detailed timing diagrams for internal signals
- Synthesis and implementation guidance
- Test infrastructure details

Specification Level

This HAS represents a **black-box** view of the APB Crossbar:

HAS vs MAS Coverage Comparison

Aspect	HAS Coverage	MAS Coverage
External ports	Complete	Complete
Functional behavior	Complete	Complete
Internal architecture	Overview only	Detailed
FSM states	-	Complete
Timing paths	Interface only	All paths
RTL structure	-	Complete

Next: Document Conventions

RTL Design Sherpa · Learning Hardware Design Through Practice GitHub · Documentation Index · MIT License

---

Document Conventions

Notation

Signal Naming

Signal Naming Conventions

Convention	Meaning	Example
`m<i>_apb_*`	Master-side APB signals (input to crossbar)	`m0_apb_PSEL`
`s<j>_apb_*`	Slave-side APB signals (output from crossbar)	`s2_apb_PREADY`
`P*`	APB protocol signal prefix	`PADDR`, `PWRITE`, `PRDATA`
`<i>`, `<j>`	Master/slave index placeholder	m0, s3

Parameter Naming

Parameter Naming Conventions

Convention	Meaning	Example
`UPPER_CASE`	Module parameters	`ADDR_WIDTH`, `DATA_WIDTH`
`NUM_*`	Count parameters	`NUM_MASTERS`, `NUM_SLAVES`
`*_WIDTH`	Bit width parameters	`DATA_WIDTH = 32`
`BASE_*`	Base address parameters	`BASE_ADDR = 0x10000000`

Diagrams

Block Diagrams

Block diagrams in this document use:

- **Rectangles** for functional blocks
- **Arrows** showing data flow direction
- **Dashed lines** for configuration/control paths
- **Labels** on connections indicating signal groups

Address Notation

Addresses are shown in hexadecimal with the `0x` prefix:

- `0x1000_0000` - Underscores separate 16-bit groups for readability
- `[31:0]` - Bit range notation (MSB:LSB)
- `64KB` - Size notation using standard abbreviations

Table Captions

Tables are numbered sequentially within each chapter. A colon (`:`) followed by the caption text appears below each table.

Requirement References

When referencing requirements from the PRD:

- **FR-n** - Functional Requirement number n
- **PR-n** - Performance Requirement number n
- **IR-n** - Interface Requirement number n

Next: Definitions and Acronyms

RTL Design Sherpa · Learning Hardware Design Through Practice GitHub · Documentation Index · MIT License

---

Definitions and Acronyms

Acronyms

Acronym Definitions

Acronym	Definition
AMBA	Advanced Microcontroller Bus Architecture
APB	Advanced Peripheral Bus
CPU	Central Processing Unit
DMA	Direct Memory Access
FSM	Finite State Machine
GPIO	General Purpose Input/Output
HAS	Hardware Architecture Specification
I2C	Inter-Integrated Circuit
LOC	Lines of Code
LUT	Lookup Table (FPGA resource)
MAS	Micro-Architecture Specification
PRD	Product Requirements Document
RTL	Register Transfer Level

Acronym	Definition
SoC	System on Chip
SPI	Serial Peripheral Interface
UART	Universal Asynchronous Receiver/Transmitter

Terms

Term Definitions

Term	Definition
Address Decode	Logic that determines which slave should receive a transaction based on address
Arbitration	Process of selecting one request when multiple masters compete for the same slave
Back-to-back	Consecutive transactions with no idle cycles between them
Crossbar	Interconnect topology allowing any master to communicate with any slave
Grant Persistence	Maintaining arbitration grant through entire transaction duration
Master	Device that initiates APB transactions (CPU, DMA, etc.)
Round-Robin	Arbitration scheme that rotates priority among requesters
Slave	Device that responds to APB transactions (peripherals)
Starvation	Condition where a requester is indefinitely denied access
Transaction	Complete APB read or write operation including setup and data phases
Zero-Bubble	No idle cycles inserted between consecutive transactions

APB Protocol Signals

APB Protocol Signal Definitions

Signal	Direction	Description
PCLK	Input	APB clock (all signals synchronous to rising edge)
PRESETn	Input	Active-low asynchronous reset
PADDR	Master to Slave	Address bus
PSEL	Master to Slave	Peripheral select (one per slave)
PENABLE	Master to Slave	Enable signal (high during data phase)
PWRITE	Master to Slave	Write/read direction (1=write, 0=read)
PWDATA	Master to Slave	Write data bus
PSTRB	Master to Slave	Write strobes (byte enables)
PPROT	Master to Slave	Protection attributes
PRDATA	Slave to Master	Read data bus
PREADY	Slave to Master	Ready signal (can extend transaction)
PSLVERR	Slave to Master	Slave error response

Next: Chapter 2: System Overview

Use Cases

Primary Use Cases

UC-1: Simple Peripheral Interconnect (1x4)

Scenario: Single CPU accessing multiple peripherals

A microcontroller needs to access UART, GPIO, Timer, and SPI peripherals through a single APB master interface.

Configuration: - 1 Master (CPU) - 4 Slaves (UART, GPIO, Timer, SPI) - Variant: `apb_xbar_1to4`

Address Map:

```
0x1000_0000 - 0x1000_FFFF : UART  (64KB)
0x1001_0000 - 0x1001_FFFF : GPIO  (64KB)
0x1002_0000 - 0x1002_FFFF : Timer (64KB)
0x1003_0000 - 0x1003_FFFF : SPI   (64KB)
```

UC-2: Multi-Master System (2x4)

Scenario: CPU and DMA both accessing peripherals

An SoC where both CPU and DMA controller need access to shared peripherals.

Configuration: - 2 Masters (CPU, DMA) - 4 Slaves (peripherals) - Variant: `apb_xbar_2to4`

Key Behavior: - Round-robin arbitration when both masters access same peripheral - No master starvation guaranteed - Zero-bubble throughput on uncontended access

UC-3: Protocol Conversion (1x1)

Scenario: APB timing boundary or clock domain preparation

Insert APB crossbar as a timing stage or preparation for clock domain crossing.

Configuration: - 1 Master - 1 Slave - Variant: `apb_xbar_1to1` or `apb_xbar_thin`

Benefits: - Isolates master from slave timing - Provides consistent transaction buffering - Minimal resource overhead (thin variant)

Variant Selection Guide

Variant Selection Guide

Use Case	Recommended Variant	Rationale
Simple peripheral bus	`apb_xbar_1to4`	Most common SoC configuration

Use Case	Recommended Variant	Rationale
CPU + DMA system	`apb_xbar_2to4`	Fair access for both masters
Timing isolation	`apb_xbar_thin`	Minimal overhead passthrough
Multi-master arbitration	`apb_xbar_2to1`	Focus on arbitration only
Custom topology	Generator	Any MxN configuration

Next: Key Features

RTL Design Sherpa · Learning Hardware Design Through Practice GitHub · Documentation Index · MIT License

---

Key Features

Feature Summary

The APB Crossbar provides the following key features:

F1: Parametric MxN Configuration

- Support for any combination of M masters and N slaves
- Pre-generated variants for common configurations (1x1, 2x1, 1x4, 2x4)
- Python generator for custom configurations up to 16x16
- Single RTL source serves all configurations

F2: Automatic Address-Based Routing

- Parallel address decode for all slaves
- Fixed 64KB address space per slave
- Configurable base address via `BASE_ADDR` parameter
- Zero-decode-latency routing

Address Calculation:

```
slave_index = (PADDR - BASE_ADDR) >> 16
```

F3: Round-Robin Arbitration

- Independent arbiter per slave
- Fair rotation of master priority
- No master starvation guaranteed
- Grant persistence through transaction completion

F4: Zero-Bubble Throughput

- Back-to-back transactions without idle cycles
- Grant held during entire transaction
- Immediate response routing
- Maximum APB bandwidth utilization

F5: Proven Building Blocks

- Built from production-tested `apb_slave.sv` and `apb_master.sv` modules
- No new protocol logic - pure composition
- Each component independently verified

Feature Comparison

Pre-Generated Variant Comparison

Feature	apb_xbar_1to1	apb_xbar_2to1	apb_xbar_1to4	apb_xbar_2to4	apb_xbar_thin
Masters	1	2	1	4	1
Slaves	1	1	4	4	1
Arbitration	No	Yes	No	Yes	No
Address Decode	No	No	Yes	Yes	No
Approximate LOC	200	400	500	1000	150

Design Philosophy

Composition Over Complexity

The crossbar is built by composing well-tested building blocks:

1. **APB Slaves** (M instances) - Convert incoming APB to internal cmd/rsp
2. **Arbiters** (N instances) - Select winning master per slave
3. **Address Decode** - Route commands to appropriate slave
4. **Response Routing** - Return responses to originating masters
5. **APB Masters** (N instances) - Convert internal cmd/rsp back to APB

Scalability

Resource usage scales predictably: - M x N routing paths - N independent arbiters - Linear growth with configuration size

---

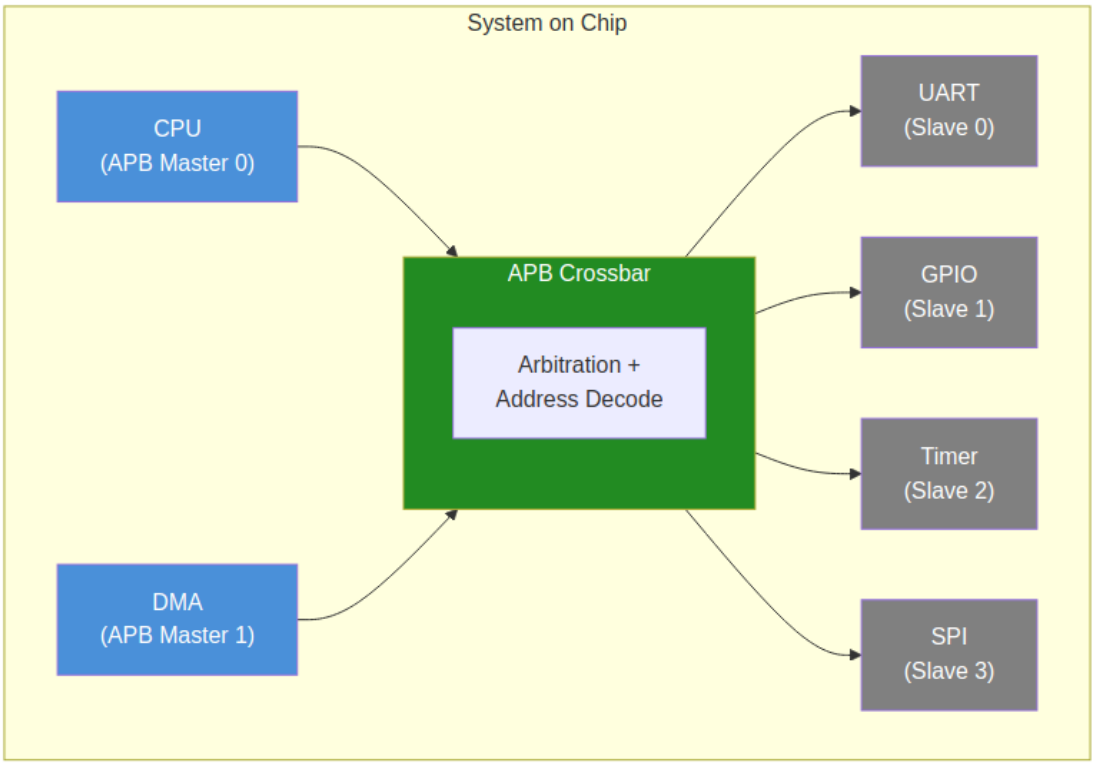
Next: System Context

System Context

System Context Diagram

The following diagram shows the APB Crossbar in a typical SoC context:

Figure 2.1: System Context Diagram



System Context Diagram

Typical Integration Topology

Master-Side Connections

The APB Crossbar connects to upstream APB masters:

Typical Master Connections

Master Type	Typical Source	Connection Point
CPU	AHB-to-APB Bridge	`m0_apb_*`
DMA	DMA Controller APB Port	`m1_apb_*`
Debug	Debug Access Port	`m2_apb_*` (if present)

Slave-Side Connections

The crossbar connects to downstream APB peripherals:

Typical Slave Connections (1to4 example)

Slave Type	Address Offset	Connection Point
UART	+0x0_0000	`s0_apb_*`
GPIO	+0x1_0000	`s1_apb_*`
Timer	+0x2_0000	`s2_apb_*`
SPI	+0x3_0000	`s3_apb_*`

Interface Boundaries

Input Boundary (Master-Side)

Each master port presents a complete APB slave interface:

- Receives PSEL, PENABLE, PADDR, PWRITE, PWDATA, PSTRB, PPROT
- Returns PRDATA, PSLVERR, PREADY

Output Boundary (Slave-Side)

Each slave port presents a complete APB master interface:

- Drives PSEL, PENABLE, PADDR, PWRITE, PWDATA, PSTRB, PPROT
- Receives PRDATA, PSLVERR, PREADY

Clock and Reset

Single clock domain: - `pclk` - APB clock (all signals synchronous) - `presetn` - Active-low asynchronous reset

System Integration Considerations

Address Map Planning

The fixed 64KB per-slave allocation requires planning:

1. **Peripheral fit:** Ensure each peripheral needs <= 64KB address space
2. **Base address:** Set `BASE_ADDR` to avoid conflicts with other subsystems
3. **Address width:** `ADDR_WIDTH` must accommodate BASE_ADDR + (NUM_SLAVES x 64KB)

Multi-Master Considerations

When using multiple masters:

1. **Arbitration overhead:** Expect 1-2 cycle delay on contended access
2. **Priority:** All masters have equal priority (round-robin)
3. **Bandwidth:** Shared access reduces per-master bandwidth

Next: Chapter 3: Architecture

RTL Design Sherpa · Learning Hardware Design Through Practice GitHub · Documentation Index · MIT License

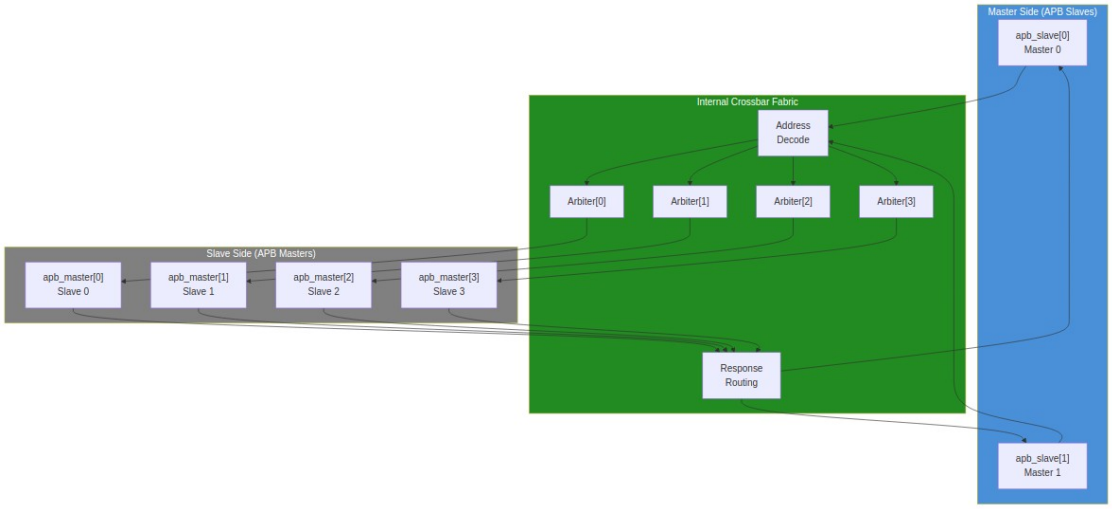
---

Block Diagram

Top-Level Architecture

The APB Crossbar consists of three main functional layers:

Figure 3.1: APB Crossbar Block Diagram



Block Diagram

Functional Blocks

Master-Side Protocol Conversion

Component: `apb_slave[M]` instances (one per master)

Each APB slave instance: - Accepts incoming APB transactions from a master - Converts APB protocol to internal command/response format - Provides transaction buffering - Generates appropriate response timing

Interface: - Input: Complete APB slave signals from external master - Output: Internal cmd/rsp bus to crossbar fabric

Internal Crossbar Fabric

Components: - **Address Decode** - Parallel decode logic - **Per-Slave Arbiters** - Round-robin arbiter per slave (N instances) - **Response Routing** - Registered response mux

The crossbar fabric: 1. Decodes transaction address to determine target slave 2. Arbitrates when multiple masters request the same slave 3. Routes commands to the selected slave 4. Routes responses back to the originating master

Slave-Side Protocol Conversion

Component: `apb_master[N]` instances (one per slave)

Each APB master instance: - Accepts internal cmd/rsp from crossbar fabric - Generates compliant APB transactions to the slave - Handles wait states via PREADY - Propagates error responses via PSLVERR

Interface: - Input: Internal cmd/rsp bus from crossbar fabric - Output: Complete APB master signals to external slave

Module Hierarchy

```
apb_xbar_MxN
+-- apb_slave[0..M-1]          # Master-side conversion
+-- addr_decode                 # Parallel address decode
+-- arbiter[0..N-1]            # Per-slave round-robin
+-- response_mux               # Response routing
+-- apb_master[0..N-1]         # Slave-side conversion
```

Data Path Width

All data paths maintain consistent width:

Data Path Width Parameters

Parameter	Default	Range	Description
ADDR_WIDTH	32	8-64	Address bus width
DATA_WIDTH	32	8-64	Data bus width
STRB_WIDTH	4	1-8	Byte strobe

Parameter	Default	Range	Description
			width (DATA_WIDTH /8)

Next: Data Flow

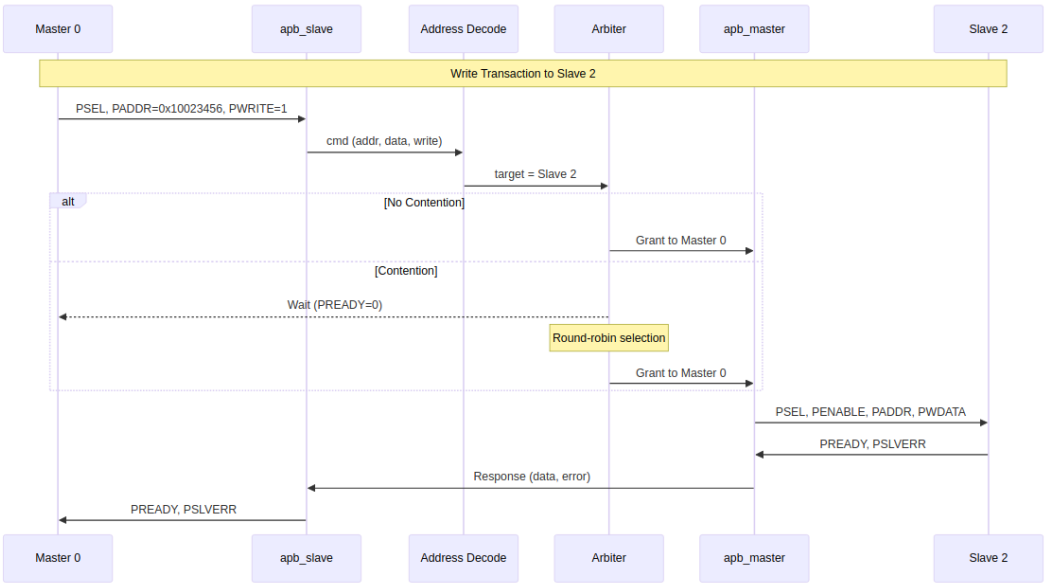
RTL Design Sherpa · Learning Hardware Design Through Practice GitHub · Documentation Index · MIT License

---

Data Flow

Transaction Flow Overview

Figure 3.2: Transaction Data Flow



Data Flow Diagram

Write Transaction Flow

A write transaction flows through the following stages:

Stage 1: Master Request

The external master initiates an APB write: 1. Master asserts PSEL and PADDR with target address 2. Master sets PWRITE=1 and provides PWDATA 3. Next cycle: Master asserts PENABLE

Stage 2: Protocol Conversion (Master-Side)

The `apb_slave` module converts to internal format: 1. Captures transaction parameters (address, data, write) 2. Generates internal command 3. Routes to address decode

Stage 3: Address Decode

Parallel decode determines target slave:

```
slave_index = (PADDR - BASE_ADDR) >> 16
```

Stage 4: Arbitration (if needed)

If multiple masters target the same slave: 1. Arbiter selects one master (round-robin priority) 2. Other masters wait (PREADY low) 3. Grant held until transaction completes

Stage 5: Protocol Conversion (Slave-Side)

The `apb_master` module generates APB transaction: 1. Asserts PSEL to selected slave 2. Provides PADDR, PWRITE, PWDATA 3. Asserts PENABLE next cycle 4. Waits for slave PREADY

Stage 6: Response Return

Response flows back to originating master: 1. Slave asserts PREADY (with optional PSLVERR) 2. `apb_master` captures response 3. Response routed to correct `apb_slave` 4. `apb_slave` asserts PREADY to original master

Read Transaction Flow

Read transactions follow the same flow with these differences:

- PWRITE=0 indicates read operation
- No PWDATA on forward path
- PRDATA returned on response path

Timing Summary

Transaction Timing Summary

Transaction Phase	Clock Cycles	Notes
Setup phase	1	PSEL asserted, PENABLE low

Transaction Phase	Clock Cycles	Notes
Data phase	1+	PENABLE high, wait for PREADY
Decode	0	Combinational (parallel decode)
Arbitration	0-1	0 if uncontended, 1 if contended

Back-to-Back Transactions

The crossbar supports zero-bubble back-to-back transactions:

1. Master can assert new PSEL immediately after PREADY
2. Grant persistence eliminates re-arbitration overhead
3. Maximum throughput: 1 transaction per 2 APB cycles

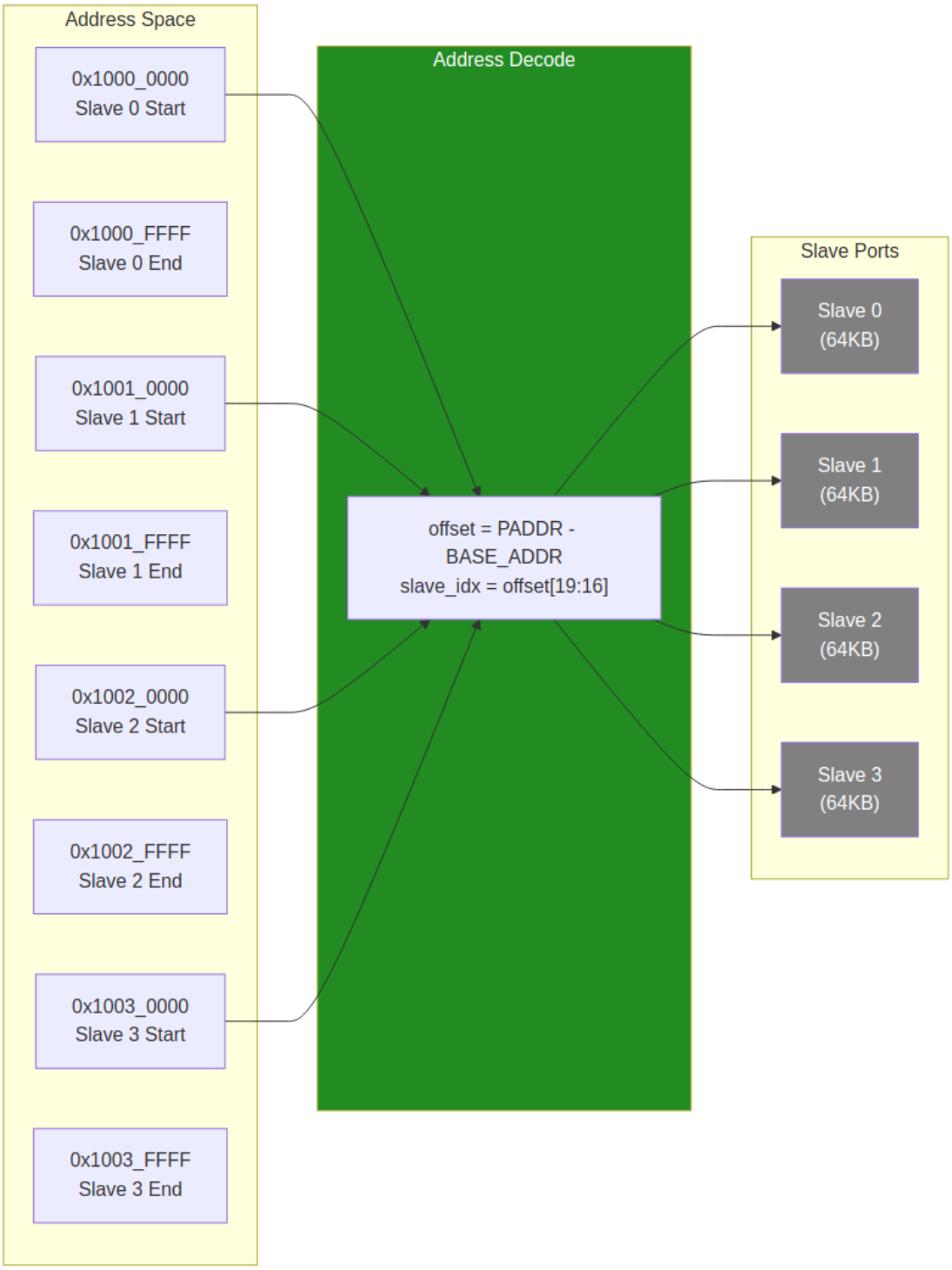
Next: Address Mapping

Address Mapping

Address Map Structure

The APB Crossbar uses a fixed 64KB address region per slave:

Figure 3.3: Address Mapping Structure



Address Mapping

Default Address Map

With default `BASE_ADDR = 0x1000_0000`:

Default Address Map (4-slave example)

Slave Index	Start Address	End Address	Size
0	0x1000_0000	0x1000_FFFF	64KB
1	0x1001_0000	0x1001_FFFF	64KB
2	0x1002_0000	0x1002_FFFF	64KB
3	0x1003_0000	0x1003_FFFF	64KB
…	…	…	…
N-1	BASE + (N-1)*64KB	BASE + N*64KB - 1	64KB

Address Decode Logic

The target slave is determined by simple bit extraction:

```
offset = PADDR - BASE_ADDR
slave_index = offset[19:16]    // Bits 19:16 select slave (0-15)
local_addr = offset[15:0]      // Bits 15:0 are local address within
slave
```

Example Decode

For address `0x1002_3456` with `BASE_ADDR = 0x1000_0000`:

Address Decode Example

Step	Calculation	Result
1. Compute offset	0x1002_3456 - 0x1000_0000	0x0002_3456
2. Extract slave index	offset[19:16]	0x2 (Slave 2)
3. Extract local address	offset[15:0]	0x3456

Customizing Base Address

The `BASE_ADDR` parameter shifts the entire address map:

```
// Peripheral space at 0x4000_0000
apb_xbar_1to4 #(
    .BASE_ADDR(32'h4000_0000)
) u_xbar (...);

// Resulting map:
// Slave 0: 0x4000_0000 - 0x4000_FFFF
// Slave 1: 0x4001_0000 - 0x4001_FFFF
// Slave 2: 0x4002_0000 - 0x4002_FFFF
// Slave 3: 0x4003_0000 - 0x4003_FFFF
```

Address Map Constraints

Fixed Slave Region Size

Each slave occupies exactly 64KB (0x10000 bytes). This is a current design constraint.

Implications: - Peripherals requiring more than 64KB need multiple slave ports - Peripherals using less than 64KB have unused address space

Maximum Configuration

With 4-bit slave index extraction: - Maximum 16 slaves - Total address space: 16 x 64KB = 1MB

Address Width Requirement

The `ADDR_WIDTH` parameter must be sufficient to address: - BASE_ADDR + (NUM_SLAVES x 64KB)

Example: For 4 slaves at BASE_ADDR = 0x8000_0000, need: - 0x8000_0000 + 0x4_0000 = 0x8004_0000 - Minimum ADDR_WIDTH: 32 bits

---

Next: Chapter 4: Interfaces

Master-Side APB Interfaces

Interface Overview

Each master port provides a complete APB slave interface to accept transactions from an external APB master (CPU, DMA, etc.).

Signal Definition

For each master index `i` (0 to M-1):

Master Port Signal Definition

Signal	Width	Direction	Description
`m<i>_apb_PSEL`	1	Input	Peripheral select
`m<i>_apb_PENABLE`	1	Input	Transfer enable
`m<i>_apb_PADDR`	ADDR_WIDTH	Input	Address bus
`m<i>_apb_PWRITE`	1	Input	Write/read direction
`m<i>_apb_PWDATA`	DATA_WIDTH	Input	Write data bus
`m<i>_apb_PSTRB`	STRB_WIDTH	Input	Write strobes
`m<i>_apb_PPROT`	3	Input	Protection attributes
`m<i>_apb_PRDATA`	DATA_WIDTH	Output	Read data bus
`m<i>_apb_PSLVERR`	1	Output	Slave error response
`m<i>_apb_PREADY`	1	Output	Transfer ready

Signal Descriptions

PSEL (Input)

Peripheral select signal. When asserted: - Indicates master wants to access the crossbar - Must remain asserted throughout the transaction - Combined with PADDR determines target slave

PENABLE (Input)

Transfer enable signal. APB state machine: - PSEL=1, PENABLE=0: Setup phase - PSEL=1, PENABLE=1: Access phase (data transfer)

PADDR (Input)

Address bus. Used for: - Target slave selection (upper bits) - Local address within slave (lower bits) - Must be stable while PSEL asserted

PWRITE (Input)

Transfer direction: - PWRITE=1: Write transaction (data from master) - PWRITE=0: Read transaction (data to master)

PWDATA (Input)

Write data bus: - Valid during write transactions - Sampled when PREADY and PENABLE both high - Width set by DATA_WIDTH parameter

PSTRB (Input)

Write byte strobes: - One bit per byte of DATA_WIDTH - Indicates which bytes are valid in PWDATA - Ignored during read transactions

PPROT (Input)

Protection attributes (3 bits): - Bit 0: Privileged/normal access - Bit 1: Secure/non-secure access - Bit 2: Instruction/data access

PRDATA (Output)

Read data bus: - Valid during read transactions when PREADY high - Undefined during write transactions - Width matches DATA_WIDTH parameter

PSLVERR (Output)

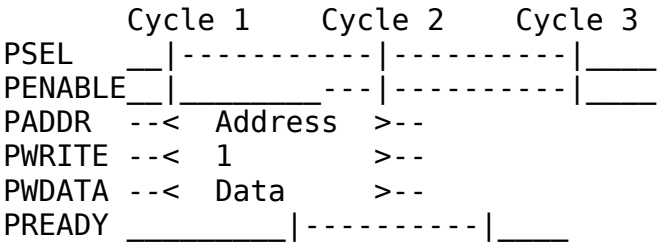
Slave error response: - Sampled when PREADY and PENABLE both high - PSLVERR=1 indicates error condition - Propagated from downstream slave

PREADY (Output)

Transfer ready signal: - PREADY=1 allows transaction to complete - PREADY=0 inserts wait states - Reflects downstream slave PREADY (or arbitration wait)

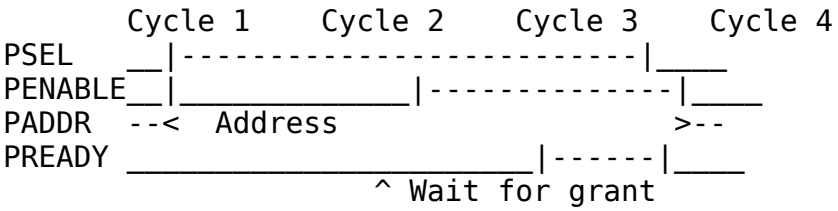
Transaction Timing

Uncontended Write (No Arbitration Wait)



Contended Transaction (Arbitration Wait)

When another master holds the target slave:



Next: Slave-Side Interfaces

RTL Design Sherpa · Learning Hardware Design Through Practice GitHub · Documentation Index · MIT License

---

Slave-Side APB Interfaces

Interface Overview

Each slave port provides a complete APB master interface to drive transactions to an external APB slave (peripheral).

Signal Definition

For each slave index `j` (0 to N-1):

Slave Port Signal Definition

Signal	Width	Direction	Description
s<j>_apb_PSEL	1	Output	Peripheral select
s<j>_apb_PENABLE	1	Output	Transfer enable
s<j>_apb_PADDR	ADDR_WIDTH	Output	Address bus
s<j>_apb_PWRITE	1	Output	Write/read direction
s<j>_apb_PWDATA	DATA_WIDTH	Output	Write data bus
s<j>_apb_PSTRB	STRB_WIDTH	Output	Write strobes
s<j>_apb_PPROT	3	Output	Protection attributes
s<j>_apb_PRDATA	DATA_WIDTH	Input	Read data bus
s<j>_apb_PSLVERR	1	Input	Slave error response
s<j>_apb_PREADY	1	Input	Transfer ready

Signal Behavior

Output Signals (Crossbar to Slave)

PSEL: Asserted when a transaction targets this slave. Remains high throughout the transaction.

PENABLE: Follows standard APB timing: - Low during setup phase - High during access phase

PADDR: Contains the full address from the master. The local address within the 64KB region is in bits [15:0].

PWRITE, PWDATA, PSTRB, PPROT: Passed through from the winning master without modification.

Input Signals (Slave to Crossbar)

PRDATA: Captured by crossbar when both PREADY and PENABLE are high. Routed to originating master.

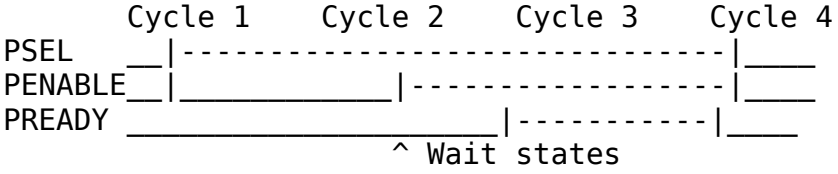
PSLVERR: Captured with PRDATA. Propagated to originating master as error indication.

PREADY: Controls transaction completion: - PREADY=1: Transaction completes this cycle - PREADY=0: Insert wait states

Slave Response Handling

Wait States

Slaves can insert wait states by holding PREADY low:



The crossbar: - Holds master PREADY low while waiting - Does not timeout (assumes slaves always respond) - Passes through all wait states to master

Error Response

When slave asserts PSLVERR:

1. Crossbar captures PSLVERR with PRDATA
2. Routes PSLVERR to originating master
3. Master sees PSLVERR=1 when its PREADY goes high

No error handling or recovery - error indication only.

Multi-Slave Considerations

Address Overlap Prevention

Each slave port is mutually exclusive: - Only one slave receives PSEL for any given address - No overlapping address regions - Address decode is deterministic

Independent Operation

Each slave operates independently: - Own PREADY timing - No synchronization between slaves - Concurrent transactions to different slaves allowed (different masters)

Next: Clock and Reset

Clock and Reset

Clock Interface

Signal Definition

Clock Signal

Signal	Width	Direction	Description
`pclk`	1	Input	APB clock

Clock Requirements

Single Clock Domain: - All crossbar logic operates on pclk rising edge - All APB signals sampled/driven synchronous to pclk - No internal clock generation or division

Frequency: - No minimum frequency requirement - Maximum frequency determined by critical path - Typical: 100-250 MHz (depends on target technology)

Duty Cycle: - No specific duty cycle requirement - Standard 50% duty cycle recommended

Reset Interface

Signal Definition

Reset Signal

Signal	Width	Direction	Description
`presetn`	1	Input	Active-low asynchronous reset

Reset Behavior

Assertion (presetn goes low): - All internal state cleared immediately - All output signals driven to safe values - Master ports: PREADY=1, PRDATA=0, PSLVERR=0 - Slave ports: PSEL=0, PENABLE=0, PADDR=0, etc.

Deassertion (presetn goes high): - Internal state begins normal operation - Reset release synchronized to pclk internally - Ready to accept transactions immediately

Reset Safe Values

Reset Safe Values

Signal Group	Reset Value	Rationale
Master PREADY	1	No false wait states
Master PRDATA	0	No false data
Master PSLVERR	0	No false errors
Slave PSEL	0	No false transactions
Slave PENABLE	0	No false enables
Arbiter grants	0	No master selected

Clock-Reset Relationship

Reset Assertion

Reset can be asserted at any time: - Asynchronous assertion (immediate effect) - Synchronous or asynchronous deassertion - All in-flight transactions aborted

Reset Deassertion

For clean operation: 1. Assert reset for minimum 2 pclk cycles 2. Deassert reset synchronous to pclk rising edge 3. Wait 1 cycle before first transaction

Reset Sequence Diagram

```
pclk      _|~|_|~|_|~|_|~|_|~|_|~|_|~|_

presetn   ________|~~~~~~~~~~~~~~~~

Internal  xxxxxxxx|--- Valid State --
State

Ready for ________|--- Transactions --
Traffic
```

Integration Notes

Clock Domain Crossing

If masters or slaves are in different clock domains: - Insert appropriate CDC logic external to crossbar - Crossbar assumes single clock domain - Consider APB clock bridge if needed

Clock Gating

The crossbar supports clock gating: - No internal activity when idle - Safe to gate pclk when no transactions pending - All registers use standard flip-flops (no clock gating cells)

Next: Chapter 5: Performance

RTL Design Sherpa · Learning Hardware Design Through Practice GitHub · Documentation Index · MIT License

---

Throughput Characteristics

Maximum Throughput

Single Master

With a single master accessing any slave:

Single Master Throughput

Metric	Value	Notes
Minimum cycles per transaction	2	APB protocol minimum
Maximum transactions per cycle	0.5	1 transaction / 2 cycles
Data throughput (32-bit @ 100MHz)	200 MB/s	Theoretical maximum
Data throughput (32-bit @ 250MHz)	500 MB/s	Theoretical maximum

Multi-Master (Uncontended)

When masters access different slaves: - Each master achieves single-master throughput - Aggregate throughput = M x single-master throughput - No arbitration overhead

Multi-Master (Contended)

When masters compete for the same slave:

Contended Access Throughput

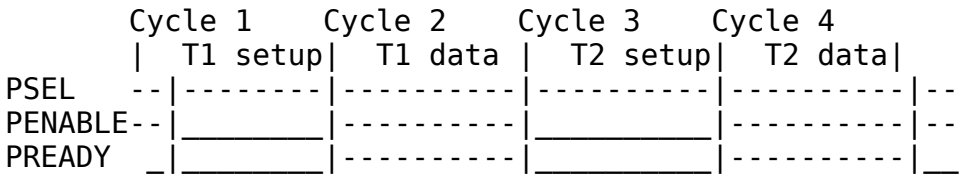
Masters	Effective Throughput	Notes
2	50% per master	Round-robin sharing
3	33% per master	Round-robin sharing
4	25% per master	Round-robin sharing

Zero-Bubble Operation

Grant Persistence

The crossbar implements grant persistence: - Once a master wins arbitration, it holds the grant - Grant released only when master releases PSEL - Enables back-to-back transactions without re-arbitration

Back-to-Back Timing



Transaction T1 and T2 are consecutive with no idle cycles.

Throughput Factors

Factors That Reduce Throughput

Throughput Reduction Factors

Factor	Impact	Mitigation
Slave wait states	Variable	Choose faster peripherals
Arbitration conflicts	1+ cycle	Distribute access across slaves
Slave response time	Variable	Design slaves for quick response

Optimal Usage Pattern

For maximum throughput: 1. Distribute master access across different slaves 2. Minimize slave wait states 3. Use burst-like access patterns (same master, same slave) 4. Avoid rapid master switching on same slave

---

Next: Latency Analysis

Latency Analysis

Transaction Latency

Uncontended Access

When a master has exclusive access to a slave:

Uncontended Transaction Latency

Phase	Cycles	Description
Setup	1	PSEL asserted, PENABLE low
Access	1+	PENABLE high, wait for PREADY
Total	**2+**	Minimum 2 cycles

Contended Access

When multiple masters compete for the same slave:

Arbitration Latency

Scenario	Additional Latency	Description
Win arbitration	0 cycles	No delay
Lose to 1 master	2+ cycles	Wait for one transaction

Scenario	Additional Latency	Description
Lose to N masters	2N+ cycles	Wait for N transactions

Latency Breakdown

Forward Path (Master to Slave)

Forward Path Latency

Component	Latency	Type
apb_slave capture	0-1 cycle	Registered
Address decode	0 cycles	Combinational
Arbitration	0-1 cycle	Combinational + wait
apb_master drive	0-1 cycle	Registered
Typical Total	**1-2 cycles**	From PSEL to slave PSEL

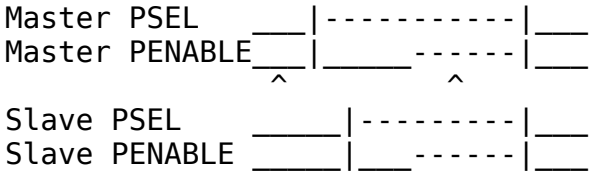
Response Path (Slave to Master)

Response Path Latency

Component	Latency	Type
Slave response	Variable	PREADY timing
apb_master capture	0-1 cycle	Registered
Response routing	0 cycles	Combinational
apb_slave drive	0-1 cycle	Registered
Typical Total	**1-2 cycles**	From slave PREADY to master PREADY

Latency Timing Diagram

Best Case (No Waits, No Contention)



```
Slave PREADY  _________|-----|___
                       ^
Master PREADY _________|-----|___
                       ^
              Total: 2 cycles (APB minimum)
```

Worst Case (Contention + Slave Wait States)

```
Master PSEL   ___|-------------------------|___
Master PENABLE___|_____________________----------|___
                 ^  Arbitration    ^
                 |  wait           |
Slave PSEL    _____________|---------------|___
Slave PENABLE _____________|______---------|___
Slave PREADY  ___________________|------|___
                                 ^
Master PREADY ___________________|------|___
                                 ^
              Total: 5+ cycles (depends on contention + slave)
```

Latency Optimization

Design Recommendations

Latency Optimization Recommendations

Goal	Recommendation
Minimize contention	Use more slaves, spread access
Reduce arbitration wait	Lower master count per slave
Reduce slave latency	Design slaves with quick PREADY

Next: Resource Estimates

RTL Design Sherpa · Learning Hardware Design Through Practice GitHub · Documentation Index · MIT License

Resource Estimates

FPGA Resource Usage

Pre-Generated Variants

Estimated resource usage for common configurations (32-bit data width):

FPGA Resource Estimates (32-bit)

Variant	LUTs	FFs	Description
apb_xbar_thin	~50	~20	Minimal passthrough
apb_xbar_1to1	~150	~80	Full 1x1 with buffering
apb_xbar_2to1	~300	~150	2 masters, arbitration
apb_xbar_1to4	~400	~200	4 slaves, decode
apb_xbar_2to4	~800	~400	Full 2x4 crossbar

Scaling Factors

Resource usage scales approximately as:

Resource Scaling Factors

Component	Scaling	Notes
apb_slave instances	M x base	Linear with masters
apb_master instances	N x base	Linear with slaves
Arbiters	N x log2(M)	Per slave, sized for masters
Address decode	N	Comparators per slave
Response mux	N x M	Full crosspoint for responses

Lines of Code

Source code metrics for pre-generated variants:

Source Code Metrics

Variant	Lines of Code	Modules	Complexity
apb_xbar_thin	~150	1	Low
apb_xbar_1to1	~200	3	Low
apb_xbar_2to1	~400	5	Medium
apb_xbar_1to4	~500	7	Medium
apb_xbar_2to4	~1000	11	Medium-High

Power Considerations

Static Power

Static power scales with: - Total register count (FFs) - Routing resources used - Technology node

Dynamic Power

Dynamic power depends on: - Transaction rate (activity factor) - Data width (toggling bits) - Clock frequency

Power Optimization

Power Optimization Techniques

Technique	Benefit	Trade-off
Clock gating	Reduce idle power	Adds gating logic
Lower data width	Fewer toggling bits	Reduced throughput
Fewer slaves	Less decode logic	Reduced flexibility

Area Comparison

Relative area comparison (normalized to apb_xbar_1to1):

Relative Area Comparison

Variant	Relative Area
apb_xbar_thin	0.3x
apb_xbar_1to1	1.0x (baseline)

Variant	Relative Area
apb_xbar_2to1	2.0x
apb_xbar_1to4	2.5x
apb_xbar_2to4	5.0x

Custom Configuration Estimation

For custom MxN configuration:

```
Estimated LUTs ~= 50 + (M * 75) + (N * 50) + (M * N * 10)
Estimated FFs  ~= 20 + (M * 40) + (N * 30)
```

Example: 4x8 crossbar - LUTs: 50 + (4*75) + (8*50) + (4*8*10) = 50 + 300 + 400 + 320 = ~1070 - FFs: 20 + (4*40) + (8*30) = 20 + 160 + 240 = ~420

Next: Chapter 6: Integration

System Requirements

Hardware Requirements

Clock and Reset

Clock and Reset Requirements

Requirement	Specification
Clock input	Single clock (pclk)
Clock frequency	No minimum, technology-dependent maximum
Reset input	Active-low asynchronous (presetn)
Reset duration	Minimum 2 clock cycles

APB Compliance

The crossbar requires APB-compliant masters and slaves:

APB Compliance Requirements

Requirement	Description
Protocol version	APB4 (with PSTRB, PPROT)
Wait states	Supported (PREADY-based)
Error response	Supported (PSLVERR)
Slave timeout	Not handled internally

Software Requirements

Address Map Configuration

Software must be configured with the correct address map:

```
// Example address definitions
#define PERIPHERAL_BASE     0x10000000
#define UART_BASE           (PERIPHERAL_BASE + 0x00000)
#define GPIO_BASE           (PERIPHERAL_BASE + 0x10000)
#define TIMER_BASE          (PERIPHERAL_BASE + 0x20000)
#define SPI_BASE            (PERIPHERAL_BASE + 0x30000)
```

Driver Considerations

Driver Considerations

Consideration	Recommendation
Polling vs interrupt	Both supported
Atomic operations	Not guaranteed across arbitration
Cache coherency	APB typically uncached

Constraints

Address Map Constraints

Address Map Constraints

Constraint	Value	Notes
Slave region size	64KB (fixed)	Cannot be changed

Constraint	Value	Notes
Maximum slaves	16	Generator limit
Maximum masters	16	Generator limit
Address alignment	Byte aligned	No alignment requirement

Transaction Constraints

Transaction Constraints

Constraint	Description
Outstanding transactions	1 per master
Transaction interleaving	Not supported
Burst transactions	Not supported

Integration Checklist

Pre-Integration

- ☐ Verify BASE_ADDR does not conflict with other subsystems
- ☐ Confirm all slaves fit within 64KB regions
- ☐ Validate ADDR_WIDTH is sufficient for address map
- ☐ Check DATA_WIDTH matches system data width

Post-Integration

- ☐ Verify address decode routing to correct slaves
- ☐ Test all master-to-slave paths
- ☐ Validate arbitration behavior (if multi-master)
- ☐ Check reset behavior

Next: Parameter Configuration

Parameter Configuration

Module Parameters

Core Parameters

Core Parameter Definition

Parameter	Type	Default	Range	Description
ADDR_WIDTH	int	32	8-64	Address bus width in bits
DATA_WIDTH	int	32	8, 16, 32, 64	Data bus width in bits
BASE_ADDR	logic[31:0]	0x10000000	Any	Base address for slave map

Derived Parameters

These are calculated automatically:

Derived Parameters

Parameter	Derivation	Description
STRB_WIDTH	DATA_WIDTH / 8	Byte strobe width
NUM_MASTERS	Fixed per variant	Number of master ports
NUM_SLAVES	Fixed per variant	Number of slave ports

Parameter Usage Examples

Default Configuration

```
apb_xbar_2to4 u_xbar (
    .pclk(apb_clk),
    .presetn(apb_rst_n),
    // ... port connections
);
// Uses defaults: ADDR_WIDTH=32, DATA_WIDTH=32, BASE_ADDR=0x10000000
```

Custom Base Address

```
apb_xbar_2to4 #(
    .BASE_ADDR(32'h4000_0000)
```

```
) u_xbar (
    .pclk(apb_clk),
    .presetn(apb_rst_n),
    // ... port connections
);
// Slaves at 0x4000_0000, 0x4001_0000, 0x4002_0000, 0x4003_0000
```

64-bit Data Width

```
apb_xbar_2to4 #(
    .DATA_WIDTH(64)
) u_xbar (
    .pclk(apb_clk),
    .presetn(apb_rst_n),
    // ... port connections with 64-bit data buses
);
// STRB_WIDTH automatically becomes 8
```

Wide Address

```
apb_xbar_2to4 #(
    .ADDR_WIDTH(40),
    .BASE_ADDR(40'h80_0000_0000)
) u_xbar (
    .pclk(apb_clk),
    .presetn(apb_rst_n),
    // ... port connections with 40-bit address buses
);
```

Parameter Validation

Address Width Check

Ensure ADDR_WIDTH can accommodate the full address range:

```
Required minimum: log2(BASE_ADDR + NUM_SLAVES * 64KB)
```

Example: 4 slaves at BASE_ADDR=0x1000_0000 - Max address: 0x1000_0000 + 4*0x10000 = 0x1004_0000 - Required: 29 bits minimum - ADDR_WIDTH=32 is sufficient

Data Width Constraints

Valid Data Width Configurations

DATA_WIDTH	STRB_WIDTH	Valid APB Widths
8	1	Yes
16	2	Yes
32	4	Yes (most common)

DATA_WIDTH	STRB_WIDTH	Valid APB Widths
64	8	Yes

Custom Generation Parameters

When using the Python generator for custom configurations:

```
python generate_xbars.py \
    --masters 3 \
    --slaves 6 \
    --base-addr 0x80000000 \
    --output apb_xbar_3to6.sv
```

Generator Command Line Options

Option	Description	Default
–masters	Number of master ports	Required
–slaves	Number of slave ports	Required
–base-addr	Base address (hex)	0x10000000
–output	Output file path	Auto-generated
–thin	Generate minimal variant	False

Next: Verification Strategy

Verification Strategy

Pre-Verified Component

The APB Crossbar is pre-verified with comprehensive tests:

Pre-Verification Status

Variant	Test Transactions	Status
apb_xbar_1to1	100+	Pass
apb_xbar_2to1	130+	Pass
apb_xbar_1to4	200+	Pass
apb_xbar_2to4	350+	Pass

Test Categories

Basic Connectivity

Tests for each variant include: - Single read transaction to each slave - Single write transaction to each slave - Read-modify-write sequences - All address regions exercised

Address Decode Verification

For multi-slave variants: - Verify each slave selected by correct address range - Boundary address testing - Invalid address behavior

Arbitration Testing

For multi-master variants: - Round-robin fairness verification - Contention handling - Grant persistence validation - No starvation testing

Stress Testing

High-intensity scenarios: - Back-to-back transactions - Random delays - Mixed read/write patterns - Concurrent multi-master access

Integration Testing Recommendations

Level 1: Basic Integration

Level 1 Integration Tests

Test	Description	Pass Criteria
Reset test	Apply reset, verify outputs	All outputs at safe values
Single transaction	Read from each slave	Correct data returned
Write verify	Write and readback	Data matches

Level 2: Functional Integration

Level 2 Integration Tests

Test	Description	Pass Criteria
All masters	Each master accesses each slave	No errors
Arbitration	Concurrent master access	Fair grant rotation
Error propagation	Force PSLVERR from slave	Error reaches master

Level 3: System Integration

Level 3 Integration Tests

Test	Description	Pass Criteria
CPU access	Software reads/writes peripherals	Correct behavior
DMA access	DMA transfers via crossbar	Data integrity
Mixed traffic	CPU and DMA concurrent	No deadlock, fair access

Verification Collateral

Available Test Infrastructure

Test Infrastructure

Item	Location	Description
CocoTB tests	`dv/tests/`	Python testbenches
Wave configs	`dv/tests/GTKW/`	GTKWave configurations
Coverage	(implicit)	Functional coverage

Running Verification

```
# Run all crossbar tests
cd projects/components/apb_xbar/dv/tests/
pytest test_apb_xbar_*.py -v
```

```
# Run specific variant
pytest test_apb_xbar_2to4.py -v

# Generate waveforms
pytest test_apb_xbar_2to4.py --vcd=debug.vcd

# View waveforms
gtkwave debug.vcd
```

Known Limitations

Not Tested

Known Limitations

Scenario	Reason	Recommendation
Slave timeout	Not implemented	Add external watchdog
>16 masters/slaves	Generator limit	Custom modification needed
Non-64KB regions	Not supported	Use multiple crossbars

End of Hardware Architecture Specification

Related Documentation

- **APB Crossbar MAS** - Detailed implementation
- **PRD.md** - Product requirements
- **README.md** - Quick start guide

RTL Design Sherpa · Learning Hardware Design Through Practice GitHub · Documentation Index · MIT License

---

APB Crossbar Micro-Architecture Specification Index

Component: APB Crossbar (MxN Interconnect) **Version:** 1.0 **Date:** 2026-01-03 **Purpose:** Detailed micro-architecture specification for APB Crossbar component

---

Document Organization

This specification covers the APB Crossbar component - a parametric MxN interconnect for connecting multiple APB masters to multiple APB slaves with automatic address-based routing and round-robin arbitration.

Main Documentation

README.md

Chapter 1: Architecture

chapters/01_architecture.md

Chapter 2: Address Decode and Arbitration

chapters/02_address_and_arbitration.md

Chapter 3: RTL Generator

chapters/03_rtl_generator.md

Quick Navigation

For New Users

1. Start with README.md for overview and quick start
2. Read 01_architecture.md to understand the design
3. Study 02_address_and_arbitration.md for operational details
4. Reference 03_rtl_generator.md if custom configuration needed

For Integration

- **Pre-generated variants:** See 01_architecture.md Section “Pre-Generated Variants”

- **Custom generation:** See 03_rtl_generator.md Section “Quick Start”
- **Address mapping:** See 02_address_and_arbitration.md Section “Address Decode”
- **Arbitration behavior:** See 02_address_and_arbitration.md Section “Arbitration”

Common Questions

All answered in README.md Section “Common Questions”

Visual Assets

All diagrams referenced in the documentation are available in:

- **Source Files:**
  - `assets/graphviz/*.gv` - Graphviz source diagrams
  - `assets/wavedrom/*.json` - WaveJSON timing diagrams
- **Rendered Files:**
  - `assets/png/*.png` - PNG format (for document generation)

Architecture Diagrams

1. **APB Crossbar Architecture (2x4 Example)**
   - Source: assets/graphviz/apb_xbar_architecture.gv
   - Rendered: assets/png/apb_xbar_architecture.png
2. **Address Decode Flow**
   - Source: assets/graphviz/address_decode_flow.gv
   - Rendered: assets/png/address_decode_flow.png

Timing Diagrams

1. **Round-Robin Arbitration**
   - Source: assets/wavedrom/arbitration_round_robin.json
   - Rendered: assets/wavedrom/arbitration_round_robin.png

Component Overview

Key Features

- **Parametric MxN Configuration:** Any combination of M masters and N slaves (up to 16x16)
- **Automatic Address Decode:** 64KB per slave, simple offset-based routing
- **Round-Robin Arbitration:** Per-slave fair arbitration, no master starvation
- **Zero-Bubble Throughput:** Back-to-back transactions without idle cycles
- **Grant Persistence:** Hold grant through transaction completion
- **RTL Generation:** Python-based generator for custom configurations

Pre-Generated Variants

Module	M×N	Use Case
apb_xbar_1to1	1×1	Passthrough, protocol conversion
apb_xbar_2to1	2×1	Multi-master arbitration
apb_xbar_1to4	1×4	Simple SoC peripheral bus
apb_xbar_2to4	2×4	Typical SoC with CPU+DMA
apb_xbar_thin	1×1	Minimal overhead passthrough

Design Philosophy

Proven Building Blocks: - Built from production-tested `apb_slave` and `apb_master` modules - No new protocol logic - pure composition - Each component independently verified

Parametric Generation: - Generator creates any MxN configuration - Pre-generated common variants for fast integration - Custom variants generated on-demand

Clean Separation: - Master-side: APB slaves convert protocol → cmd/rsp - Internal: Arbitration + address decoding - Slave-side: APB masters convert cmd/rsp → protocol

---

Related Documentation

Companion Specifications

- **APB Crossbar HAS** - Hardware Architecture Specification (high-level)

Project-Level

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

Test Infrastructure

- **Test Directory:** `../../dv/tests/` - CocoTB + pytest test suite
- **Test Results:** All pre-generated variants 100% passing

RTL

- **Core Modules:** `../../rtl/apb_xbar_*.sv` - Pre-generated crossbars
- **Wrappers:** `../../rtl/wrappers/` - Pre-configured wrappers
- **Generator:** `../../bin/generate_xbars.py` - Python generator script

Version History

Version 1.0 (2025-10-25): - Initial specification release - Complete visual documentation (3 diagrams) - Comprehensive generator documentation - All pre-generated variants verified (100% passing)

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

RTL Design Sherpa · Learning Hardware Design Through Practice GitHub · Documentation Index · MIT License

---

Product Requirements Document (PRD)

APB Crossbar Generator

Version: 1.0 **Date:** 2025-10-19 **Status:** Production Ready **Owner:** RTL Design Sherpa Project
Parent Document: `/PRD.md`

---

1. Executive Summary

The **APB Crossbar** is a parametric APB interconnect generator that creates configurable MxN crossbar fabrics for connecting multiple APB masters to multiple APB slaves. Built using proven `apb_slave` and `apb_master` modules, the crossbar provides independent round-robin arbitration per slave and automatic address-based routing.

1.1 Quick Stats

- **Modules:** 5 pre-generated variants + generator for custom sizes
- **Max Capacity:** Up to 16x16 (configurable)
- **Architecture:** apb_slave → arbitration + decode → apb_master
- **Status:** Production ready, all tests passing
- **Generator:** Python-based parametric code generation

1.2 Project Goals

- **Primary:** Provide production-quality APB interconnect for SoC integration
- **Secondary:** Demonstrate parametric RTL generation best practices
- **Tertiary:** Support both fixed and dynamically-generated variants

---

2. Key Design Principles

2.1 Architecture Philosophy

Proven Building Blocks: - Uses `apb_slave.sv` and `apb_master.sv` from rtl/amba/apb - Each module independently tested and production-proven - Crossbar = composition of proven components

Parametric Generation: - Generator creates any MxN configuration - Pre-generated common variants (1to1, 2to1, 1to4, 2to4, thin) - Custom variants generated on-demand

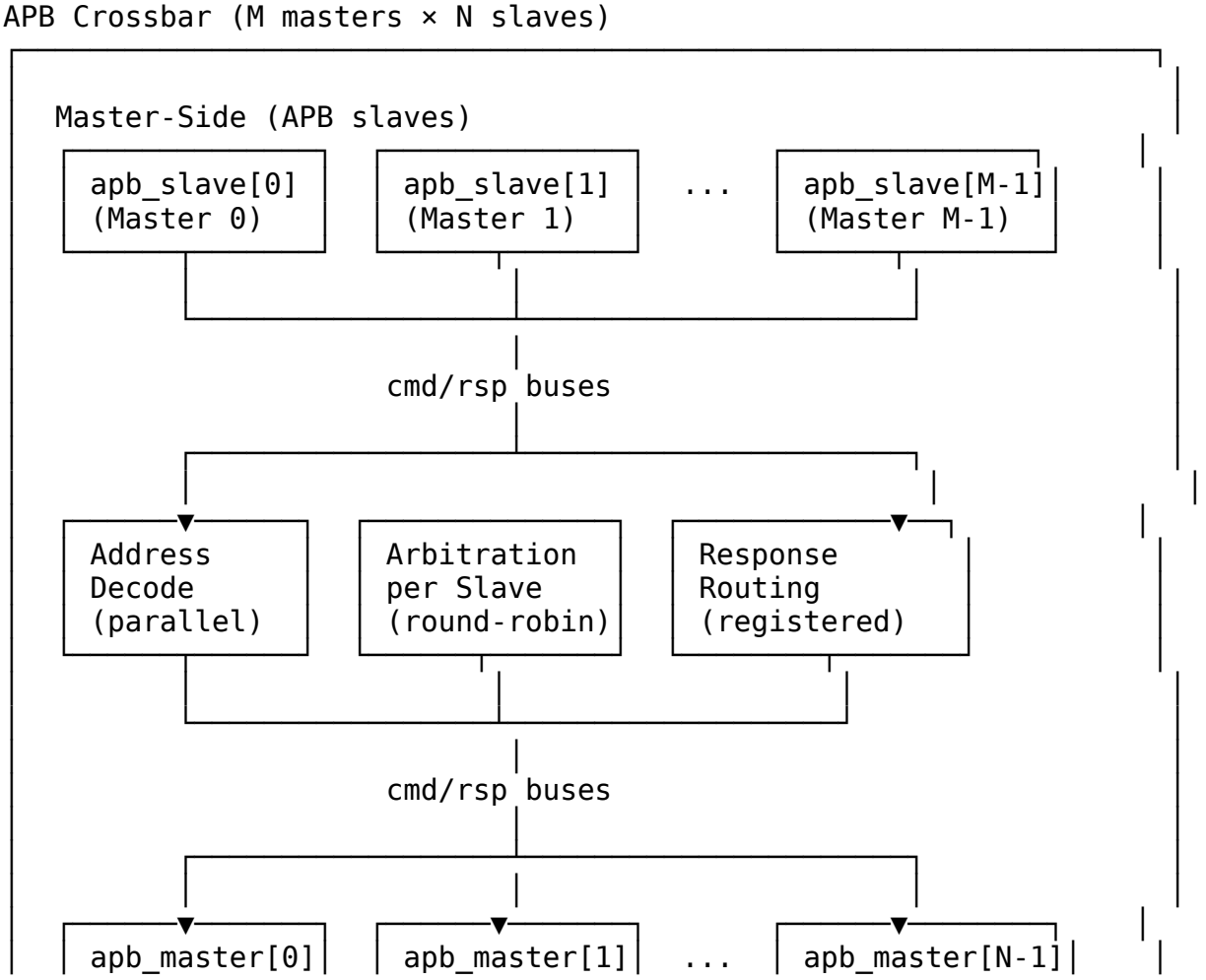
Clean Separation: - Master-side: APB slaves convert protocol to cmd/rsp - Internal: Arbitration + address decoding - Slave-side: APB masters convert cmd/rsp back to protocol

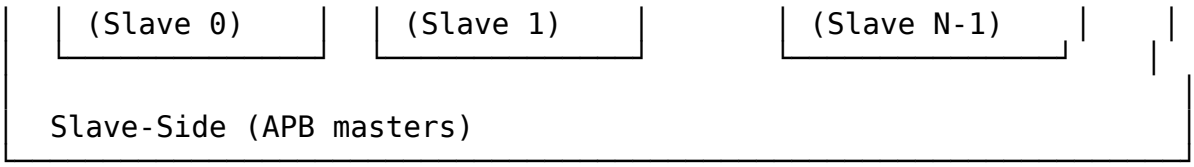
2.2 Design Trade-offs

Feature	Choice	Rationale
Arbitration	Round-robin per slave	Fair, simple, predictable
Address Map	64KB per slave	Sufficient for most peripherals
Grant Persistence	Hold through response	Zero-bubble throughput
Address Decode	Parallel decode	Low latency, simple logic

3. Architecture Overview

3.1 Top-Level Block Diagram





APB Crossbar Architecture (2x4 Example) *Figure: APB Crossbar top-level architecture showing 2 masters connected to 4 slaves.* *Source: docs/apb_xbar_spec/assets/graphviz/apb_xbar_architecture.gv* | *SVG*

3.2 Address Mapping

```
BASE_ADDR + 0x0000_0000 → 0x0000_FFFF : Slave 0 (64KB)
BASE_ADDR + 0x0001_0000 → 0x0001_FFFF : Slave 1 (64KB)
BASE_ADDR + 0x0002_0000 → 0x0002_FFFF : Slave 2 (64KB)
BASE_ADDR + 0x0003_0000 → 0x0003_FFFF : Slave 3 (64KB)
...
BASE_ADDR + 0x000F_0000 → 0x000F_FFFF : Slave 15 (64KB)
```

Default BASE_ADDR: 0x1000_0000

Address Decode Example:

The crossbar extracts the slave index from the upper bits of the address offset:

```
slave_index = (address - BASE_ADDR) >> 16  // Divide by 64KB (0x10000)
```

Address Decode Flow *Figure: Address decode flow showing how address 0x10023456 routes to Slave 2.* *Source: docs/apb_xbar_spec/assets/graphviz/address_decode_flow.gv* | *SVG*

3.3 Arbitration Strategy

Per-Slave Round-Robin: - Each slave has independent arbiter - Master priority rotates after each grant - Grant held from command acceptance through response completion - No master can starve another master

Example:

```
Slave 0 accessed by M0, M1, M0 → Next grant goes to M1
Slave 1 accessed by M1, M1, M0 → Next grant goes to M0
```

Round-Robin Arbitration Timing *Figure: Round-robin arbitration timing showing 2 masters competing for Slave 0. Master priority rotates after each grant to ensure fair access.* *Source: docs/apb_xbar_spec/assets/wavedrom/arbitration_round_robin.json*

---

4. Available Variants

4.1 Pre-Generated Modules

Module	Masters	Slaves	Use Case	File Size
apb_xbar_1to1	1	1	Protocol conversion, testing	~200 LOC
apb_xbar_2to1	2	1	Multi-master arbitration	~400 LOC
apb_xbar_1to4	1	4	Address decode, simple SoC	~500 LOC
apb_xbar_2to4	2	4	Full crossbar, typical SoC	~1000 LOC
apb_xbar_thin	1	1	Minimal passthrough	~150 LOC

4.2 Wrapper Modules

Pre-configured wrappers for common topologies:

Wrapper	Configuration	Purpose
`apb_xbar_1to1_wrap`	1×1	Simple connection
`apb_xbar_2to1_wrap`	2×1	Dual-master arbitration
`apb_xbar_1to4_wrap`	1×4	Single-master decode
`apb_xbar_2to4_wrap`	2×4	Full SoC crossbar
`apb_xbar_wrap_m10_s10`	10×10	Large crossbar
`apb_xbar_thin_wrap_m10_s10`	10×10	Minimal version

5. Functional Requirements

FR-1: Arbitrary MxN Configuration

Priority: P0 (Critical) **Status:** √ Implemented and verified

Description: Generate crossbar for any M masters and N slaves (up to 16x16)

Verification: Generator creates valid RTL for all tested configurations

FR-2: Round-Robin Arbitration

Priority: P0 (Critical) **Status:** √ Implemented and verified

Description: Fair per-slave arbitration with rotating master priority

Verification: Arbitration stress tests (130+ transactions for 2to1)

FR-3: Address-Based Routing

Priority: P0 (Critical) **Status:** √ Implemented and verified

Description: Automatic slave selection based on address ranges

Verification: Address decode validation (200+ transactions for 1to4)

FR-4: Zero-Bubble Throughput

Priority: P1 (High) **Status:** √ Implemented and verified

Description: Back-to-back transactions supported without idle cycles

Verification: Performance tests show consecutive transactions

FR-5: Configurable Address Map

Priority: P1 (High) **Status:** √ Implemented and verified

Description: BASE_ADDR parameter sets base of address map

Verification: Tests with multiple BASE_ADDR values

---

6. Interface Specifications

6.1 Master-Side Interface (APB Slave)

Per-master APB slave interface:

```
input  logic                      m<i>_apb_PSEL
input  logic                      m<i>_apb_PENABLE
input  logic [ADDR_WIDTH-1:0]     m<i>_apb_PADDR
input  logic                      m<i>_apb_PWRITE
input  logic [DATA_WIDTH-1:0]     m<i>_apb_PWDATA
input  logic [STRB_WIDTH-1:0]     m<i>_apb_PSTRB
input  logic [2:0]                m<i>_apb_PPROT
output logic [DATA_WIDTH-1:0]     m<i>_apb_PRDATA
output logic                      m<i>_apb_PSLVERR
output logic                      m<i>_apb_PREADY
```

6.2 Slave-Side Interface (APB Master)

Per-slave APB master interface:

```
output logic                      s<j>_apb_PSEL
output logic                      s<j>_apb_PENABLE
output logic [ADDR_WIDTH-1:0]     s<j>_apb_PADDR
output logic                      s<j>_apb_PWRITE
output logic [DATA_WIDTH-1:0]     s<j>_apb_PWDATA
output logic [STRB_WIDTH-1:0]     s<j>_apb_PSTRB
output logic [2:0]                s<j>_apb_PPROT
input  logic [DATA_WIDTH-1:0]     s<j>_apb_PRDATA
input  logic                      s<j>_apb_PSLVERR
input  logic                      s<j>_apb_PREADY
```

---

7. Parameter Configuration

Parameter	Type	Default	Range	Description
ADDR_WIDTH	int	32	1-64	Address bus width
DATA_WIDTH	int	32	8,16,32,64	Data bus width
STRB_WIDTH	int	DATA_WIDTH/8	-	Strobe width (auto-calc)
BASE_ADDR	logic[31:0]	0x10000000	Any	Base address for slave map

8. Generator Usage

8.1 Pre-Generated Variants

Use existing modules directly:

```
# Available in projects/components/apb_xbar/rtl/
apb_xbar_1to1.sv
apb_xbar_2to1.sv
apb_xbar_1to4.sv
apb_xbar_2to4.sv
apb_xbar_thin.sv
```

8.2 Custom Generation

Generate custom MxN crossbar:

```
cd projects/components/apb_xbar/bin/
python generate_xbars.py --masters 3 --slaves 6
python generate_xbars.py --masters 4 --slaves 8 --base-addr 0x80000000
```

8.3 Generator Options

```
python generate_xbars.py --help

Options:
  --masters M       Number of masters (1-16)
  --slaves N        Number of slaves (1-16)
  --base-addr ADDR  Base address (default: 0x10000000)
  --output FILE     Output file path
  --thin            Generate thin variant (minimal logic)
```

---

9. Performance Characteristics

9.1 Latency

Path	Latency	Notes
Command path	1 cycle	APB slave → decode → APB master
Response path	1 cycle	APB master → routing → APB slave
Total	2 cycles min	APB protocol overhead

9.2 Throughput

- **Back-to-back transactions:** Supported
- **Zero-bubble overhead:** Yes (with grant persistence)
- **Maximum rate:** 1 transaction per 2 APB cycles per master

9.3 Resource Utilization (Estimated)

Configuration	LUTs	FFs	Notes
1to1	~50	~20	Passthrough
2to1	~150	~80	Arbitration
1to4	~200	~100	Address decode
2to4	~400	~200	Full crossbar
10to10	~5K	~2K	Large crossbar

10. Verification Status

10.1 Test Coverage

Test	Transactions	Status	Coverage
test_apb_xbar_1to1	100+	√ Pass	Basic connectivity
test_apb_xbar_2to1	130+	√ Pass	Arbitration stress
test_apb_xbar_1to4	200+	√ Pass	Address decode
test_apb_xbar_2to4	350+	√ Pass	Full crossbar stress

Overall: 100% passing, >750 total transactions tested

10.2 Test Methodology

Test Structure: - CocoTB framework - Random transaction generation - Variable delay profiles - Address range validation - Arbitration fairness checks

Verification Points: - Correct address routing - Round-robin arbitration - Response integrity - Back-to-back transactions - Error propagation

---

11. Integration Guide

11.1 Simple Integration (1to4)

```
apb_xbar_1to4 #(
    .ADDR_WIDTH(32),
    .DATA_WIDTH(32),
    .BASE_ADDR(32'h1000_0000)
) u_xbar (
    .pclk(apb_clk),
    .presetn(apb_rst_n),

    // Master interface
    .m0_apb_PSEL(cpu_psel),
    .m0_apb_PENABLE(cpu_penable),
    .m0_apb_PADDR(cpu_paddr),
    .m0_apb_PWRITE(cpu_pwrite),
    .m0_apb_PWDATA(cpu_pwdata),
    .m0_apb_PSTRB(cpu_pstrb),
    .m0_apb_PPROT(cpu_pprot),
    .m0_apb_PRDATA(cpu_prdata),
    .m0_apb_PSLVERR(cpu_pslverr),
    .m0_apb_PREADY(cpu_pready),

    // Slave 0: UART (0x1000_0000 - 0x1000_FFFF)
    .s0_apb_PSEL(uart_psel),
    .s0_apb_PENABLE(uart_penable),
    // ... (similar connections)

    // Slave 1: GPIO (0x1001_0000 - 0x1001_FFFF)
    .s1_apb_PSEL(gpio_psel),
    // ... (similar connections)

    // Slave 2: Timer (0x1002_0000 - 0x1002_FFFF)
    // Slave 3: SPI (0x1003_0000 - 0x1003_FFFF)
);
```

11.2 Multi-Master Integration (2to4)

```
apb_xbar_2to4 #(
    .ADDR_WIDTH(32),
    .DATA_WIDTH(32),
    .BASE_ADDR(32'h1000_0000)
) u_xbar (
    .pclk(apb_clk),
    .presetn(apb_rst_n),

    // Master 0: CPU
```

```
    .m0_apb_PSEL(cpu_psel),
    // ... (full interface)

    // Master 1: DMA
    .m1_apb_PSEL(dma_psel),
    // ... (full interface)

    // Slaves 0-3: Peripherals
    // ... (slave connections)
);
```

---

12. Design Constraints

12.1 Limitations

Constraint	Value	Rationale
Max masters	16	Generator limit (configurable)
Max slaves	16	Generator limit (configurable)
Slave region size	64KB	Fixed per slave
No slave disable	N/A	All slaves always active
No timeout	N/A	Assumes slaves always respond

12.2 Assumptions

1. **Slave responses:** All slaves respond eventually (no timeout handling)
2. **Address ranges:** Slaves occupy 64KB regions starting at BASE_ADDR
3. **APB compliance:** All masters and slaves follow APB protocol
4. **Single clock domain:** All signals synchronous to pclk

---

13. Future Enhancements

13.1 Potential Features

- **Configurable slave region sizes:** Not just 64KB

- **Slave enable/disable:** Dynamic slave activation
- **Timeout detection:** Watchdog for hung slaves
- **Transaction ordering:** Optional in-order completion
- **Priority arbitration:** Weighted instead of round-robin

13.2 Generator Improvements

- **HDL output formats:** Support Verilog, VHDL, Chisel
- **Automated testing:** Generate tests alongside RTL
- **Documentation generation:** Auto-generate integration guides
- **Synthesis scripts:** Generate vendor-specific scripts

---

14. References

14.1 Internal Documentation

- `README.md` - Quick start guide
- `CLAUDE.md` - AI assistant guidelines
- `bin/generate_xbars.py` - Generator script
- `dv/tests/` - Test implementations

14.2 External Standards

- **AMBA APB Protocol v2.0** - ARM IHI 0024C
- **APB4 Extensions** - AMBA 5 specification

14.3 Related Components

- `rtl/amba/apb/apb_slave.sv` - Master-side building block
- `rtl/amba/apb/apb_master.sv` - Slave-side building block
- `projects/components/apb_xbar/bin/apb_xbar_generator.py` - Main generator

Document Version: 1.0 **Last Review:** 2025-10-19 **Next Review:** 2026-01-01 **Maintained By:** RTL Design Sherpa Project

RTL Design Sherpa · Learning Hardware Design Through Practice GitHub · Documentation Index · MIT License

Claude Code Guide: APB Crossbar Component

Version: 1.0 **Last Updated:** 2025-10-19 **Purpose:** AI-specific guidance for working with APB Crossbar component

---

Quick Context

What: APB Crossbar Generator - Parametric APB interconnect for connecting M masters to N slaves **Status:** √ Production Ready (all tests passing) **Your Role:** Help users generate, integrate, and customize APB crossbars

📖 **Complete Specification:** `projects/components/apb_xbar/PRD.md` ← **Always reference this for technical details**

Critical Rules for This Component

Rule #0: This is a GENERATOR, Not Just Modules

IMPORTANT: APB Crossbar is both: 1. **Pre-generated modules** (1to1, 2to1, 1to4, 2to4, thin) in `rtl/` 2. **Python generator** (`bin/generate_xbars.py`) for custom configurations

When users ask for crossbar: - √ **Check if pre-generated variant exists first** - √ **Only suggest generation if custom size needed**

Decision Tree:

```
User needs crossbar?
├─ 1x1, 2x1, 1x4, 2x4? → Use pre-generated module
├─ Thin/minimal? → Use apb_xbar_thin
└─ Custom MxN? → Run generator script
```

Rule #1: Address Map is Fixed Per-Slave

Each slave occupies 64KB region:

```
Slave 0: BASE_ADDR + 0x0000_0000 → 0x0000_FFFF
Slave 1: BASE_ADDR + 0x0001_0000 → 0x0001_FFFF
Slave 2: BASE_ADDR + 0x0002_0000 → 0x0002_FFFF
...
```

Users CANNOT change per-slave size (current limitation)

If user asks for different sizes:

```
✗ WRONG: "Let me modify the generator to support custom sizes per
slave"

✓ CORRECT: "Current design uses fixed 64KB per slave. You can:
1. Use BASE_ADDR parameter to shift entire map
2. For custom sizes, modify generator's addr_offset calculation
3. Or use multiple crossbars with different BASE_ADDR values"
```

Rule #2: Know the Pre-Generated Variants

Available in `rtl/` directory:

Module	M×N	Use Case	When to Recommend
apb_xbar_1to1	1×1	Passthrough	Protocol conversion, testing
apb_xbar_2to1	2×1	Arbitration	Multi-master to single peripheral
apb_xbar_1to4	1×4	Decode	Single CPU to multiple peripherals
apb_xbar_2to4	2×4	Full crossbar	CPU + DMA to peripherals
apb_xbar_thin	1×1	Minimal	Low-overhead passthrough

Always suggest pre-generated first!

Rule #3: Generator Syntax

Generate custom crossbar:

```
cd projects/components/apb_xbar/bin/
python generate_xbars.py --masters 3 --slaves 6
```

Options:

```
--masters M         Number of masters (1-16)
--slaves N          Number of slaves (1-16)
--base-addr ADDR    Base address (default: 0x10000000)
--output FILE       Output file path
--thin              Generate thin variant (minimal logic)
```

Example:

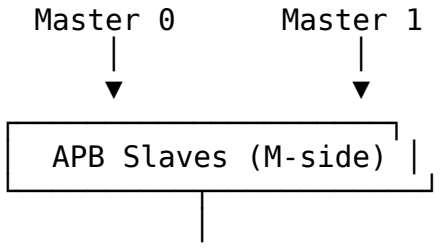
```bash
# Generate 3x8 crossbar with custom base
python generate_xbars.py --masters 3 --slaves 8 --base-addr 0x80000000

# Generate thin 5x5 variant
python generate_xbars.py --masters 5 --slaves 5 --thin
```

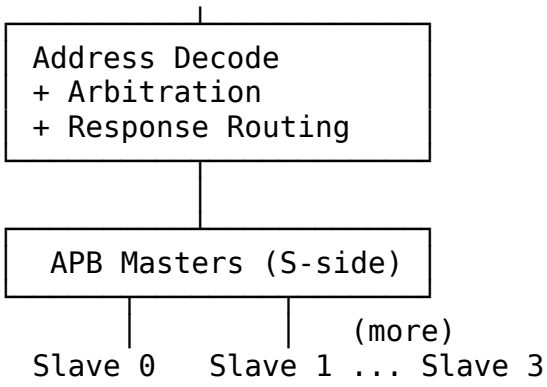
Architecture Quick Reference

Module Organization

```
projects/components/apb_xbar/
├── rtl/
│   ├── apb_xbar_1to1.sv        Core crossbar modules
│   ├── apb_xbar_2to1.sv
│   ├── apb_xbar_1to4.sv
│   ├── apb_xbar_2to4.sv
│   ├── apb_xbar_thin.sv
│   └── wrappers/               Pre-configured wrappers
│       ├── apb_xbar_1to1_wrap.sv
│       ├── apb_xbar_2to1_wrap.sv
│       ├── apb_xbar_1to4_wrap.sv
│       ├── apb_xbar_2to4_wrap.sv
│       ├── apb_xbar_wrap.sv
│       ├── apb_xbar_wrap_m10_s10.sv
│       └── apb_xbar_thin_wrap_m10_s10.sv
├── bin/
│   └── generate_xbars.py       Generator script
├── dv/
│   └── tests/
│       ├── test_apb_xbar_1to1.py
│       ├── test_apb_xbar_2to1.py
│       ├── test_apb_xbar_1to4.py
│       ├── test_apb_xbar_2to4.py
│       └── GTKW/                Waveform configs
├── PRD.md                       Complete specification
├── CLAUDE.md                    This file
└── README.md                    Quick start guide
```

Block Diagram (2×4 Example)





Common User Questions and Responses

Q: “How do I connect a CPU to 4 peripherals?”

A: Use the pre-generated 1to4 crossbar:

```
apb_xbar_1to4 #(
    .ADDR_WIDTH(32),
    .DATA_WIDTH(32),
    .BASE_ADDR(32'h1000_0000)  // Start of peripheral space
) u_peripheral_xbar (
    .pclk(apb_clk),
    .presetn(apb_rst_n),

    // CPU connection (master 0)
    .m0_apb_PSEL(cpu_psel),
    .m0_apb_PENABLE(cpu_penable),
    .m0_apb_PADDR(cpu_paddr),
    .m0_apb_PWRITE(cpu_pwrite),
    .m0_apb_PWDATA(cpu_pwdata),
    .m0_apb_PSTRB(cpu_pstrb),
    .m0_apb_PPROT(cpu_pprot),
    .m0_apb_PRDATA(cpu_prdata),
    .m0_apb_PSLVERR(cpu_pslverr),
    .m0_apb_PREADY(cpu_pready),

    // Peripheral 0: UART @ 0x1000_0000
    .s0_apb_PSEL(uart_psel),
    .s0_apb_PENABLE(uart_penable),
    // ... (full interface)

    // Peripheral 1: GPIO @ 0x1001_0000
    // Peripheral 2: Timer @ 0x1002_0000
```

```
    // Peripheral 3: SPI @ 0x1003_0000
);
```

Address map: - UART: 0x1000_0000 - 0x1000_FFFF - GPIO: 0x1001_0000 - 0x1001_FFFF - Timer: 0x1002_0000 - 0x1002_FFFF - SPI: 0x1003_0000 - 0x1003_FFFF

📖 **See:** PRD.md Section 11.1

Q: "I need CPU and DMA to access peripherals. Which crossbar?"

A: Use the 2to4 crossbar:

```
apb_xbar_2to4 #(
    .ADDR_WIDTH(32),
    .DATA_WIDTH(32),
    .BASE_ADDR(32'h1000_0000)
) u_soc_xbar (
    .pclk(apb_clk),
    .presetn(apb_rst_n),

    // Master 0: CPU
    .m0_apb_PSEL(cpu_psel),
    // ... (full interface)

    // Master 1: DMA
    .m1_apb_PSEL(dma_psel),
    // ... (full interface)

    // Slaves 0-3: Peripherals
    // ... (slave connections)
);
```

Key feature: Round-robin arbitration per slave ensures fair access between CPU and DMA.

📖 **See:** PRD.md Section 11.2

Q: "What if I need 3 masters and 8 slaves?"

A: Generate custom crossbar:

```
cd projects/components/apb_xbar/bin/
python generate_xbars.py --masters 3 --slaves 8 --output
../rtl/apb_xbar_3to8.sv

# Check generated file
ls -lh ../rtl/apb_xbar_3to8.sv
```

Then instantiate like any other crossbar:

```
apb_xbar_3to8 #(
    .ADDR_WIDTH(32),
    .DATA_WIDTH(32),
    .BASE_ADDR(32'h1000_0000)
) u_custom_xbar (
    // ... connections for 3 masters and 8 slaves
);
```

📖 **See:** PRD.md Section 8

Q: "How does arbitration work?"

A: Per-slave round-robin:

Each slave has its own arbiter that rotates master priority:

```
Example with 2 masters accessing Slave 0:

Transaction 1: M0 requests → M0 granted
Transaction 2: M0 and M1 request → M1 granted (rotated)
Transaction 3: M1 requests → M1 granted
Transaction 4: M0 and M1 request → M0 granted (rotated)
```

Key points: - **Independent per slave:** Each slave arbitrates independently - **Fair:** No master can starve another - **Grant persistence:** Once granted, master holds slave until response completes - **Zero-bubble:** Back-to-back transactions supported

📖 **See:** PRD.md Section 3.3

Q: "Can I change the address map?"

A: BASE_ADDR parameter shifts entire map, but per-slave size is fixed:

```
// Default: 0x1000_0000
apb_xbar_1to4 #(.BASE_ADDR(32'h1000_0000)) u_xbar1 (...);
// Slaves at: 0x1000_0000, 0x1001_0000, 0x1002_0000, 0x1003_0000

// Shifted: 0x8000_0000
apb_xbar_1to4 #(.BASE_ADDR(32'h8000_0000)) u_xbar2 (...);
// Slaves at: 0x8000_0000, 0x8001_0000, 0x8002_0000, 0x8003_0000
```

Limitation: Each slave always occupies 64KB (0x10000 bytes)

Workaround for different sizes: 1. Use multiple crossbars with different BASE_ADDR 2. Modify generator's `addr_offset` calculation 3. Use address masking in slaves

📖 **See:** PRD.md Section 3.2

Q: "How do I run tests?"

A: Use pytest:

```
cd projects/components/apb_xbar/dv/tests/

# Run specific test
pytest test_apb_xbar_1to4.py -v

# Run all tests
pytest test_apb_xbar_*.py -v

# Run with waveforms
pytest test_apb_xbar_2to4.py --vcd=waves.vcd
gtkwave waves.vcd
```

Test coverage: - **test_apb_xbar_1to1**: 100+ transactions (passthrough) - **test_apb_xbar_2to1**: 130+ transactions (arbitration) - **test_apb_xbar_1to4**: 200+ transactions (address decode) - **test_apb_xbar_2to4**: 350+ transactions (full stress)

All tests passing √

📖 **See:** PRD.md Section 10

Q: "What's the thin variant for?"

A: Minimal overhead passthrough:

```
apb_xbar_thin #(
    .ADDR_WIDTH(32),
    .DATA_WIDTH(32)
) u_thin_xbar (
    // ... 1×1 connection
);
```

Use cases: - Protocol conversion - Timing boundary - Clock domain crossing preparation - Testing/debugging

Difference from apb_xbar_1to1: - Fewer internal registers - Lower latency - Smaller area (~30% reduction) - No arbitration overhead

📖 **See:** README.md and PRD.md Section 4

Integration Patterns

Pattern 1: Simple Peripheral Interconnect

```
// CPU to 4 peripherals
apb_xbar_1to4 #(
    .ADDR_WIDTH(32),
    .DATA_WIDTH(32),
    .BASE_ADDR(32'h1000_0000)
) u_periph_xbar (
    .pclk(sys_clk),
    .presetn(sys_rst_n),

    // CPU side
    .m0_apb_PSEL(cpu_apb_psel),
    .m0_apb_PENABLE(cpu_apb_penable),
    .m0_apb_PADDR(cpu_apb_paddr),
    .m0_apb_PWRITE(cpu_apb_pwrite),
    .m0_apb_PWDATA(cpu_apb_pwdata),
    .m0_apb_PSTRB(cpu_apb_pstrb),
    .m0_apb_PPROT(cpu_apb_pprot),
    .m0_apb_PRDATA(cpu_apb_prdata),
    .m0_apb_PSLVERR(cpu_apb_pslverr),
    .m0_apb_PREADY(cpu_apb_pready),

    // Peripherals
    .s0_apb_PSEL(uart_psel), /* ... full interface ... */
    .s1_apb_PSEL(gpio_psel), /* ... */
    .s2_apb_PSEL(timer_psel), /* ... */
    .s3_apb_PSEL(spi_psel) /* ... */
);
```

Pattern 2: Multi-Master System

```
// CPU + DMA to peripherals
apb_xbar_2to4 #(
    .ADDR_WIDTH(32),
    .DATA_WIDTH(32),
    .BASE_ADDR(32'h4000_0000)
) u_soc_xbar (
    .pclk(apb_clk),
    .presetn(apb_rst_n),

    // Master 0: CPU
    .m0_apb_PSEL(cpu_psel),
    /* ... full CPU interface ... */

    // Master 1: DMA Controller
```

```
    .m1_apb_PSEL(dma_psel),
    /* ... full DMA interface ... */

    // Slave 0-3: Memory-mapped peripherals
    .s0_apb_PSEL(mem_ctrl_psel), /* ... */
    .s1_apb_PSEL(uart_psel), /* ... */
    .s2_apb_PSEL(i2c_psel), /* ... */
    .s3_apb_PSEL(adc_psel) /* ... */
);
```

Pattern 3: Hierarchical Interconnect

```
// Top-level crossbar
apb_xbar_1to4 u_top_xbar (
    .m0_apb_* (cpu_apb_*),
    .s0_apb_* (periph_bus0_*),  // To sub-crossbar 0
    .s1_apb_* (periph_bus1_*),  // To sub-crossbar 1
    .s2_apb_* (mem_ctrl_*),     // Direct to memory controller
    .s3_apb_* (dma_ctrl_*)      // Direct to DMA
);

// Sub-crossbar 0 for low-speed peripherals
apb_xbar_1to4 u_periph_xbar0 (
    .m0_apb_* (periph_bus0_*),
    .s0_apb_* (uart0_*),
    .s1_apb_* (gpio0_*),
    .s2_apb_* (i2c0_*),
    .s3_apb_* (spi0_*)
);

// Sub-crossbar 1 for high-speed peripherals
apb_xbar_1to4 u_periph_xbar1 (
    .m0_apb_* (periph_bus1_*),
    .s0_apb_* (uart1_*),
    .s1_apb_* (timer_*),
    .s2_apb_* (pwm_*),
    .s3_apb_* (adc_*)
);
```

Anti-Patterns to Catch

✗ Anti-Pattern 1: Generating When Pre-Generated Exists

```
✗ WRONG:
User: "I need a 2×4 crossbar"
You: "Let me generate that for you..."
```

```
python generate_xbars.py --masters 2 --slaves 4

✓ CORRECTED:
"Use the pre-generated apb_xbar_2to4.sv in the rtl/ directory.
No generation needed!"
```

✗ Anti-Pattern 2: Assuming Custom Per-Slave Sizes

```
✗ WRONG:
User: "Can I make slave 0 256KB and slave 1 4KB?"
You: "Sure, let me modify the parameters..."

✓ CORRECTED:
"Current design uses fixed 64KB per slave. For custom sizes:
1. Modify generator's addr_offset calculation
2. Or use multiple crossbars with different BASE_ADDR
3. Or implement address masking in slaves"
```

✗ Anti-Pattern 3: Not Mentioning Address Map

```
✗ WRONG:
User: "How do I integrate the crossbar?"
You: *Shows port connections only*

✓ CORRECTED:
"Here's the integration with address map:
apb_xbar_1to4 #(.BASE_ADDR(32'h1000_0000)) u_xbar (...);

Address map:
- Slave 0: 0x1000_0000 - 0x1000_FFFF
- Slave 1: 0x1001_0000 - 0x1001_FFFF
- Slave 2: 0x1002_0000 - 0x1002_FFFF
- Slave 3: 0x1003_0000 - 0x1003_FFFF"
```

✗ Anti-Pattern 4: Forgetting About Wrappers

```
✗ WRONG:
User: "I need a quick 10×10 crossbar"
You: "Run the generator with --masters 10 --slaves 10"

✓ CORRECTED:
"We have pre-configured wrappers in rtl/wrappers/:
- apb_xbar_wrap_m10_s10.sv (full version)
- apb_xbar_thin_wrap_m10_s10.sv (thin version)

Use those for faster integration!"
```

Debugging Workflow

Issue: Address Not Routing Correctly

Check in order: 1. ✓ BASE_ADDR parameter set correctly? 2. ✓ Address within slave's 64KB region? 3. ✓ PSEL signal asserted by master? 4. ✓ All APB signals properly connected?

Calculate expected slave:

```
slave_index = (address - BASE_ADDR) >> 16  // Divide by 64KB
```

Debug commands:

```
pytest dv/tests/test_apb_xbar_1to4.py --vcd=debug.vcd -v
gtkwave debug.vcd  # Check address decode logic
```

Issue: Arbitration Not Fair

Symptoms: - One master dominates - Other masters starved

Check: 1. ✓ Arbiters instantiated per slave? 2. ✓ Round-robin logic correct? 3. ✓ Grant persistence working?

Verify with tests:

```
pytest dv/tests/test_apb_xbar_2to1.py -v  # Arbitration stress test
```

Issue: Back-to-Back Transactions Stalling

Check: 1. ✓ Grant persistence enabled? 2. ✓ Slaves responding with PREADY? 3. ✓ No unintended pipeline bubbles?

View waveforms:

```
pytest dv/tests/test_apb_xbar_2to4.py --vcd=perf.vcd
gtkwave perf.vcd  # Check for idle cycles
```

Quick Commands

```
# List available crossbars
ls projects/components/apb_xbar/rtl/*.sv

# Generate custom crossbar
cd projects/components/apb_xbar/bin/
python generate_xbars.py --masters 3 --slaves 6

# Run all tests
cd projects/components/apb_xbar/dv/tests/
```

```
pytest test_apb_xbar_*.py -v

# Run specific test with waveforms
pytest test_apb_xbar_2to4.py --vcd=debug.vcd -v

# View waveforms
gtkwave debug.vcd

# Check documentation
cat projects/components/apb_xbar/PRD.md
cat projects/components/apb_xbar/README.md
```

Remember

1. 🔍 **Check pre-generated first** - Don't generate unnecessarily
2. 📍 **Address map matters** - Always mention BASE_ADDR + 64KB regions
3. ⚖️ **Fair arbitration** - Round-robin per slave
4. 🔗 **Complete connections** - All APB signals must be wired
5. ✓ **Tests available** - 100% passing, comprehensive coverage
6. 📚 **Wrappers exist** - Check rtl/wrappers/ for common configs
7. 🎯 **Generator limits** - Up to 16×16 (configurable)

Version: 1.0 **Last Updated:** 2025-10-19 **Maintained By:** RTL Design Sherpa Project