Table of Contents

Apb Xbar Index

Generated: 2025-12-07

APB Crossbar Specification - Table of Contents

Component: APB Crossbar (MxN Interconnect) **Version:** 1.0 **Last Updated:** 2025-10-25 **Status:** Production Ready (All tests passing)

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/svg/*.svg` - SVG format (scalable vector graphics for all diagrams)

Architecture Diagrams

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

Timing Diagrams

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

– Rendered: assets/svg/arbitration_round_robin.svg

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

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)

Document Generated: 2025-10-25 **Maintained By:** RTL Design Sherpa Project

APB Crossbar Visual Documentation

Component: APB Crossbar (MxN Interconnect) **Version:** 1.0 **Status:** Production Ready **Last Updated:** 2025-10-25

Overview

This directory contains visual documentation for the APB Crossbar component, including architecture diagrams, timing waveforms, and generator documentation.

Component Purpose: Parametric MxN APB interconnect connecting multiple masters to multiple slaves with automatic address-based routing and round-robin arbitration.

Documentation Structure

Main Documentation Files

Document	Description	Contents
01_architecture.md	Architecture Overview	Top-level block diagram, functional blocks, signal flow, parameters
02_address_and_arbitration.md	Address Decode & Arbitration	Address map structure, decode algorithm, round-robin timing
03_rtl_generator.md	RTL Generator Guide	Generator architecture, usage, customization, advanced topics

Supporting Documentation

Document	Location	Description
PRD.md	../../PRD.md	Complete product requirements document
CLAUDE.md	../../CLAUDE.md	AI assistant integration guide
README.md	../../README.md	Quick start guide

Visual Assets

Architecture Diagrams

Diagram	Format	Description
apb_xbar_architecture	Graphviz	2x4 crossbar showing master-side slaves, arbitration, slave-side masters
address_decode_flow	Graphviz	Step-by-step address decode

Diagram	Format	Description
		example (0x10023456 → Slave 2)

Rendered Formats: - PNG: `assets/svg/*.svg` (for markdown embedding) - SVG: `assets/svg/*.svg` (for web viewing, scalable)

Timing Diagrams

Diagram	Format	Description
arbitration_round_robin	WaveJSON	2 masters competing for Slave 0 with round-robin arbitration

Rendered Format: - PNG: `assets/svg/arbitration_round_robin.svg`

Quick Start

For New Users

1. **Understand the Architecture**
   - Read 01_architecture.md
   - Study the architecture diagram
   - Understand master-side/slave-side protocol conversion
2. **Learn Address Decode & Arbitration**
   - Read 02_address_and_arbitration.md
   - Study address decode flow diagram
   - Study arbitration timing diagram
3. **Generate Custom Crossbar (if needed)**
   - Read 03_rtl_generator.md
   - Use pre-generated variants (1to1, 2to1, 1to4, 2to4) if possible
   - Run generator for custom MxN configurations
4. **Integrate Into Your Design**
   - See integration examples in CLAUDE.md
   - Reference complete specification in PRD.md

For Existing Users

Need a Crossbar? - 1 master, N slaves → Use `apb_xbar_1toN` or generate - 2 masters, N slaves → Use `apb_xbar_2toN` or generate - M masters, N slaves → Generate with `generate_xbars.py`

Understanding Behavior? - Address decode issues → See 02_address_and_arbitration.md - Arbitration questions → See 02_address_and_arbitration.md

Modifying/Generating? - Generator usage → See 03_rtl_generator.md

---

Directory Structure

```
docs/apb_xbar_spec/
├── README.md                                  ← This file
├── 01_architecture.md                         ← Architecture overview
├── 02_address_and_arbitration.md              ← Decode & arbitration
details
├── 03_rtl_generator.md                        ← Generator documentation
└── assets/
    ├── graphviz/                              ← Source diagrams
    │   ├── apb_xbar_architecture.gv
    │   └── address_decode_flow.gv
    ├── wavedrom/                              ← Source timing diagrams
    │   └── arbitration_round_robin.json
    ├── svg/                                   ← Rendered SVG (scalable)
    │   ├── apb_xbar_architecture.svg
    │   └── address_decode_flow.svg
    └── png/                                   ← Rendered PNG (embedded)
        ├── apb_xbar_architecture.svg
        ├── address_decode_flow.svg
        └── arbitration_round_robin.svg
```

Key Concepts

1. Architecture

Three-Stage Design: 1. **Master-Side:** APB slaves convert protocol → cmd/rsp 2. **Internal Logic:** Address decode + arbitration + response routing 3. **Slave-Side:** APB masters convert cmd/rsp → protocol

Why This Design? - Reuses proven `apb_slave` and `apb_master` components - Clean separation of concerns - Scalable to any MxN configuration

2. Address Decode

Fixed 64KB Per Slave:

```
Slave 0: BASE_ADDR + 0x00000 - 0x0FFFF
Slave 1: BASE_ADDR + 0x10000 - 0x1FFFF
```

```
Slave 2: BASE_ADDR + 0x20000 - 0x2FFFF
...
```

Decode Formula:

```
offset = PADDR - BASE_ADDR
slave_index = offset[19:16]  // Divide by 64KB
```

See: 02_address_and_arbitration.md

3. Arbitration

Per-Slave Round-Robin: - Each slave has independent arbiter - Priority rotates after each grant - Grant persists through transaction completion - No master starvation

See: 02_address_and_arbitration.md

4. RTL Generation

All Generator Code in Component Area: - **Convenience Wrapper:** `bin/generate_xbars.py` (top-level script) - **Core Generator:** `bin/apb_xbar_generator.py` (library implementation)

Usage:

```
# Generate standard variants
python generate_xbars.py

# Generate custom 3x6 crossbar
python generate_xbars.py --masters 3 --slaves 6
```

See: 03_rtl_generator.md

Common Questions

Q: Which pre-generated variant should I use?

Decision Tree:

```
1 master, 1 slave  → apb_xbar_1to1 (passthrough)
1 master, 4 slaves → apb_xbar_1to4 (address decode only)
2 masters, 1 slave → apb_xbar_2to1 (arbitration only)
2 masters, 4 slaves → apb_xbar_2to4 (full crossbar)
Other MxN          → Generate with generate_xbars.py
```

Q: How do I change the base address?

At Instantiation:

```
apb_xbar_1to4 #(
    .BASE_ADDR(32'h8000_0000)  // Override default 0x10000000
) u_xbar (...);
```

Or Generate with Custom Base:

```
python generate_xbars.py --masters 2 --slaves 4 --base-addr 0x80000000
```

Q: Can I change per-slave address sizes?

No (Current Limitation): Each slave is fixed at 64KB.

Workarounds: 1. Use multiple crossbars with different BASE_ADDR 2. Modify generator's decode logic (advanced) 3. Address translation in slaves

See: 03_rtl_generator.md#limitations

Q: How does arbitration work with 3+ masters?

Round-Robin Example (4 Masters, Same Slave):

```
Initial priority: M0
Transaction 1: M0 requests → M0 granted → Priority rotates to M1
Transaction 2: M0, M1, M2 request → M1 granted → Priority rotates to
M2
Transaction 3: M0, M3 request → M2 skipped (not requesting) → M3
granted → Priority rotates to M0
Transaction 4: M0, M1 request → M0 granted → Priority rotates to M1
```

Key: Priority rotates after each grant, ensuring fairness.

See: 02_address_and_arbitration.md#arbitration

Q: What's the throughput?

Single Master: - Zero-bubble (back-to-back transactions without gaps) - Limited only by slave PREADY response time

Multiple Masters (Same Slave): - Fair sharing: Each master gets ~1/M bandwidth - Example: 2 masters = 50% each, 4 masters = 25% each

Multiple Masters (Different Slaves): - Full parallelism: Each master gets 100% of its target slave - Total system bandwidth = NUM_SLAVES × slave_bandwidth

See: 02_address_and_arbitration.md#performance

Regenerating Diagrams

Graphviz Diagrams (Architecture, Address Decode)

```
cd assets/graphviz/

# Generate SVG
dot -Tsvg apb_xbar_architecture.gv -o ../svg/apb_xbar_architecture.svg
dot -Tsvg address_decode_flow.gv -o ../svg/address_decode_flow.svg

# Generate PNG
dot -Tsvg apb_xbar_architecture.gv -o ../svg/apb_xbar_architecture.svg
dot -Tsvg address_decode_flow.gv -o ../svg/address_decode_flow.svg
```

WaveJSON Diagrams (Timing)

```
cd assets/wavedrom/

# Generate PNG
wavedrom-cli -i arbitration_round_robin.json -s
../svg/arbitration_round_robin.svg
```

Version History

Version 1.0 (2025-10-25): - Initial visual documentation release - 3 main documentation files - 2 Graphviz architecture diagrams - 1 WaveJSON timing diagram - RTL generator documentation

---

Maintainers

RTL Design Sherpa Project

Questions or Issues: 1. Check this documentation first 2. Review PRD.md for complete specification 3. Review CLAUDE.md for integration guidance 4. Open GitHub issue if problem persists

---

Last Updated: 2025-10-25 **Version:** 1.0 **Status:** Production Ready

APB Crossbar Architecture

Component: APB Crossbar (MxN Interconnect) **Version:** 1.0 **Status:** Production Ready

---

Overview

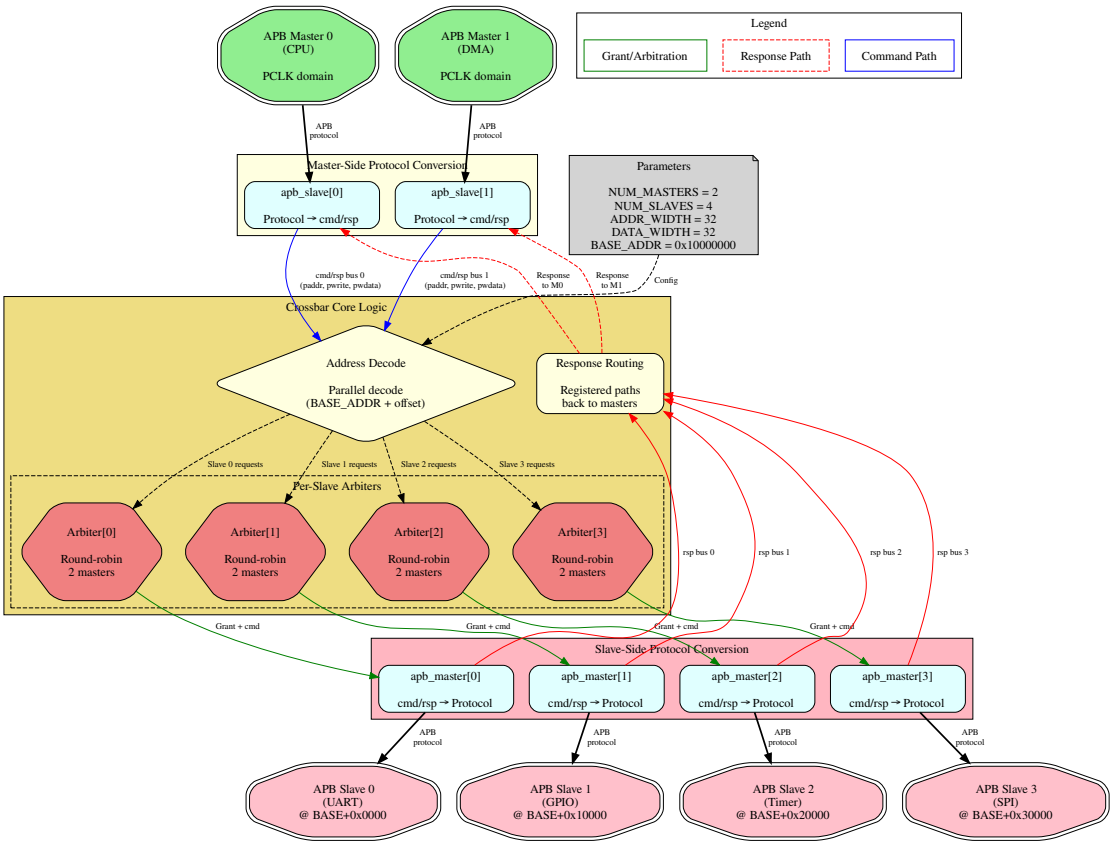
The APB Crossbar is a parametric interconnect that connects M APB masters to N APB slaves with automatic address-based routing and per-slave round-robin arbitration. Built from proven `apb_slave` and `apb_master` components, the crossbar provides a clean, scalable solution for SoC peripheral interconnect.

Key Features: - Arbitrary MxN configuration (up to 16x16) - Automatic address decode (64KB per slave) - Round-robin arbitration per slave - Zero-bubble throughput - Grant persistence through transaction completion

---

Architecture Diagram

The following diagram shows a 2x4 crossbar configuration connecting 2 masters (CPU and DMA) to 4 slaves (UART, GPIO, Timer, SPI):



APB Crossbar Architecture (2x4 Example)

Figure: APB Crossbar top-level architecture showing 2 masters connected to 4 slaves via master-side protocol conversion, internal arbitration logic, and slave-side protocol conversion. Source | SVG

Functional Blocks

1. Master-Side Protocol Conversion

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

Purpose: Convert incoming APB protocol transactions to internal cmd/rsp bus format

Features: - Full APB protocol handling (PSEL, PENABLE, PREADY) - Transaction buffering - Error response generation - Back-to-back transaction support

Dataflow:

```
APB Master → apb_slave → cmd/rsp bus → Internal Crossbar Logic
```

---

2. Internal Crossbar Logic

Components: - **Address Decode** - Parallel decode to determine target slave - **Per-Slave Arbiters** - Round-robin arbitration for each slave - **Response Routing** - Registered paths back to requesting masters

Key Operations:

Address Decode:

```
offset = PADDR - BASE_ADDR
slave_index = offset[19:16]  // Upper 4 bits of 20-bit offset (64KB
regions)
```

Arbitration: - Independent arbiter per slave - Round-robin priority rotation - Grant persistence through response - No starvation guarantee

Response Routing: - Track which master initiated each transaction - Route PRDATA/PSLVERR back to originating master - Maintain transaction ordering per master

---

3. Slave-Side Protocol Conversion

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

Purpose: Convert internal cmd/rsp bus format back to APB protocol for slaves

Features: - APB protocol generation (PSEL, PENABLE timing) - Response collection (PRDATA, PSLVERR, PREADY) - Wait state handling - Error propagation

Dataflow:

```
Internal Crossbar Logic → cmd/rsp bus → apb_master → APB Slave
```

Signal Flow Example

Transaction: Master 0 (CPU) writes to Slave 2 (Timer) at address 0x10023456

Step-by-step:

1. **Master 0 → apb_slave[0]:**
   - CPU asserts PSEL, PADDR=0x10023456, PWRITE=1

- apb_slave[0] converts to cmd/rsp format

2. **Address Decode:**
   - offset = 0x10023456 - 0x10000000 = 0x00023456
   - slave_index = 0x00023456 >> 16 = 0x2
   - Target: Slave 2
3. **Arbiter[2]:**
   - Check if Slave 2 is available
   - Grant to Master 0 (if no conflict)
   - Route cmd to apb_master[2]
4. **apb_master[2] → Slave 2:**
   - Generate APB write transaction
   - Assert PSEL[2], PENABLE, PADDR, PWDATA
   - Wait for PREADY
5. **Response Path:**
   - Slave 2 responds with PREADY, PSLVERR
   - apb_master[2] captures response
   - Response routed back to apb_slave[0]
   - apb_slave[0] returns PREADY to CPU

Total Latency: Typically 2-3 cycles for uncontended access

Parameter Configuration

Top-Level Parameters:

Parameter	Range	Default	Description
`NUM_MASTERS`	1-16	2	Number of APB masters
`NUM_SLAVES`	1-16	4	Number of APB slaves
`ADDR_WIDTH`	8-64	32	Address bus width
`DATA_WIDTH`	8-64	32	Data bus width
`BASE_ADDR`	Any	0x10000000	Base address for address map

Derived Parameters: - Slave address range: 64KB (0x10000) per slave - Total address space: NUM_SLAVES × 64KB

Pre-Generated Variants

Module	M×N	LOC	Use Case
`apb_xbar_1to1`	1×1	~200	Passthrough, protocol conversion
`apb_xbar_2to1`	2×1	~400	Multi-master arbitration
`apb_xbar_1to4`	1×4	~500	Simple SoC peripheral bus
`apb_xbar_2to4`	2×4	~1000	Typical SoC with CPU+DMA
`apb_xbar_thin`	1×1	~150	Minimal overhead passthrough

Custom Variants: Generated on-demand via Python script

Design Philosophy

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

Scalability: - Parametric generation for any MxN configuration - Resource usage scales linearly with M×N - No centralized bottlenecks

Predictability: - Round-robin arbitration provides deterministic behavior - Fixed address map simplifies software integration - Zero-bubble design ensures maximum throughput

---

Next Steps

- See Address Decode and Arbitration for detailed operation examples

- See PRD.md for complete specification
- See CLAUDE.md for integration guidance
- See README.md for quick start guide

Version: 1.0 **Last Updated:** 2025-10-25 **Maintained By:** RTL Design Sherpa Project

Address Decode and Arbitration

Component: APB Crossbar (MxN Interconnect) **Version:** 1.0 **Status:** Production Ready

---

Overview

This document details the two core mechanisms of the APB Crossbar: 1. **Address Decode** - How incoming addresses map to specific slaves 2. **Arbitration** - How multiple masters share access to the same slave

---

Address Decode

Address Map Structure

The crossbar uses a fixed 64KB (0x10000 bytes) region per slave:

```
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: 0x10000000

Total Address Space: NUM_SLAVES × 64KB

Decode Algorithm

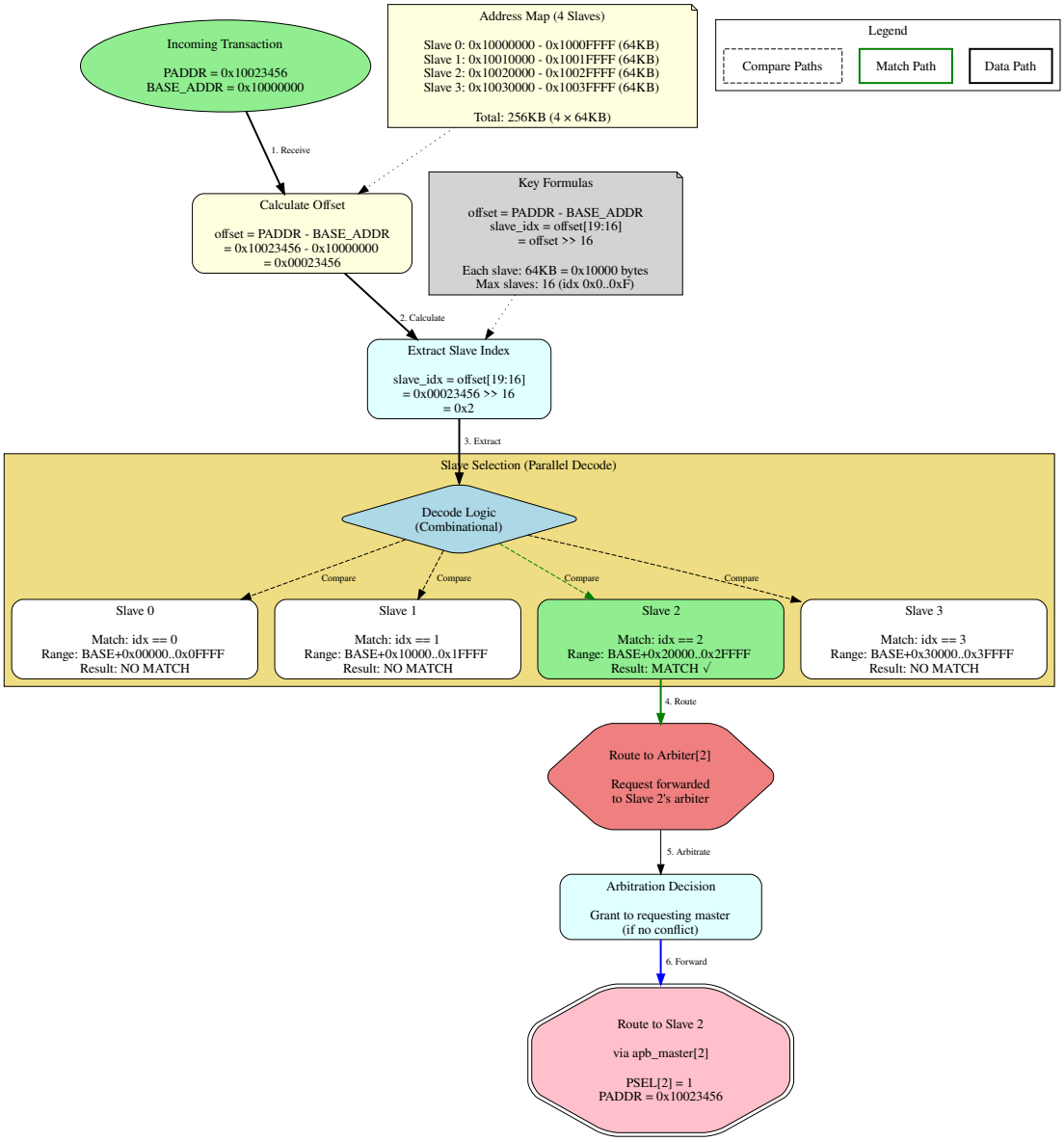
The crossbar extracts the slave index from the upper bits of the address offset:

```
offset = PADDR - BASE_ADDR
slave_index = offset[19:16]  // Upper 4 bits of 20-bit offset
```

Why offset[19:16]? - 64KB = 0x10000 = 2^16 bytes - Lower 16 bits (offset[15:0]) are byte address within slave - Upper 4 bits (offset[19:16]) select which of 16 slaves - Supports up to 16 slaves (0x0 through 0xF)

Address Decode Flow Diagram

The following diagram shows a concrete example of how address 0x10023456 routes to Slave 2:



Address Decode Flow

Figure: Address decode flow showing step-by-step routing of address 0x10023456 to Slave 2. Source | SVG

Decode Example Walkthrough

Scenario: Master accesses address 0x10023456 with BASE_ADDR = 0x10000000

Step 1: Calculate Offset

```
offset = PADDR - BASE_ADDR
       = 0x10023456 - 0x10000000
       = 0x00023456
```

Step 2: Extract Slave Index

```
slave_index = offset[19:16]
            = 0x00023456 >> 16
            = 0x2
```

Step 3: Parallel Decode

The crossbar checks all slave ranges in parallel: - Slave 0: 0x00000 - 0x0FFFF → NO MATCH - Slave 1: 0x10000 - 0x1FFFF → NO MATCH - **Slave 2: 0x20000 - 0x2FFFF → MATCH** √ - Slave 3: 0x30000 - 0x3FFFF → NO MATCH

Step 4: Route to Arbiter

Transaction request forwarded to Arbiter[2] (Slave 2's arbiter)

Step 5: Forward to Slave

After arbitration grant, transaction forwarded via apb_master[2] to physical Slave 2

Final Address Sent to Slave: 0x10023456 (full address preserved)

Multiple Address Maps

You can create multiple distinct address maps by using different BASE_ADDR values:

Example: Two Crossbars

```
// Peripheral Bus: 0x1000_0000 - 0x1003_FFFF
apb_xbar_1to4 #(
    .BASE_ADDR(32'h1000_0000)
) u_periph_xbar (...);
// Slaves at: 0x1000_0000, 0x1001_0000, 0x1002_0000, 0x1003_0000

// Memory-Mapped I/O: 0x8000_0000 - 0x8003_FFFF
apb_xbar_1to4 #(
    .BASE_ADDR(32'h8000_0000)
```

```
) u_mmio_xbar (...);
// Slaves at: 0x8000_0000, 0x8001_0000, 0x8002_0000, 0x8003_0000
```

Arbitration

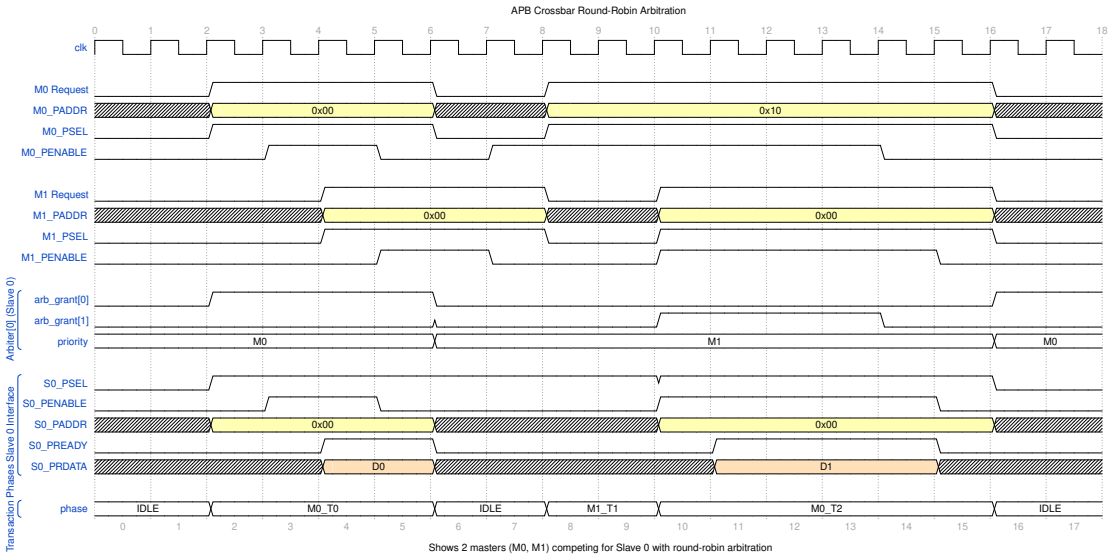
Per-Slave Round-Robin

Each slave has an **independent arbiter** that implements round-robin scheduling:

Key Properties: - **Fair:** No master can starve another - **Independent:** Each slave arbitrates separately - **Predictable:** Priority rotates after each grant - **Persistent:** Grant held from command through response

Round-Robin Timing Diagram

The following timing diagram shows 2 masters (M0, M1) competing for access to Slave 0:



Round-Robin Arbitration Timing

Source: arbitration_round_robin.json

Arbitration Example Walkthrough

Scenario: Master 0 and Master 1 both want to access Slave 0

Initial State: - Priority: M0 (M0 has priority initially) - Slave 0: IDLE

Transaction 1: M0 Requests Slave 0

```
Cycle 1: M0 asserts request (M0_PSEL, M0_PADDR=0x1000_0000)
Cycle 2: Arbiter[0] grants to M0 (only requester)
         M0_PENABLE asserted
         S0_PSEL asserted to Slave 0
Cycle 3: Slave 0 responds (S0_PREADY)
         Transaction completes
         Priority rotates: M1 now has priority
```

Transaction 2: M0 and M1 Both Request Slave 0

```
Cycle 4: M0 asserts request (M0_PADDR=0x1000_0010)
         M1 asserts request (M1_PADDR=0x1000_0000) -- CONFLICT!
Cycle 5: Arbiter[0] grants to M1 (has priority)
         M1_PENABLE asserted
         S0_PSEL asserted to Slave 0
         M0 blocked, waits
Cycle 6: Slave 0 responds (S0_PREADY)
         M1 transaction completes
         Priority rotates: M0 now has priority
```

Transaction 3: M0 Request (After Rotation)

```
Cycle 7: M0 still asserting request (was blocked)
Cycle 8: Arbiter[0] grants to M0 (now has priority)
         M0_PENABLE asserted
         S0_PSEL asserted to Slave 0
Cycle 9: Slave 0 responds (S0_PREADY)
         Transaction completes
         Priority rotates: M1 now has priority
```

Result: Fair access - each master gets served in turn when both request

Multi-Slave Parallelism

Key Feature: Different slaves can be accessed simultaneously by different masters

Example: Parallel Transactions

```
Time T0:
- Master 0 accesses Slave 0 (UART) - GRANTED
- Master 1 accesses Slave 2 (Timer) - GRANTED
Both transactions proceed in parallel (no conflict)

Time T1:
- Master 0 accesses Slave 0 (UART) - GRANTED
- Master 1 accesses Slave 0 (UART) - BLOCKED (arbitration)
Master 1 waits for Master 0 to complete

Time T2:
```

```
- Master 0 completes, releases Slave 0
- Master 1 accesses Slave 0 (UART) - GRANTED
Priority rotated for Slave 0's arbiter
```

Benefit: Maximum throughput when masters access different slaves

Grant Persistence

Critical Property: Once granted, a master holds the slave until response completes

Why This Matters:

```
WITHOUT Grant Persistence:
- Master 0 granted Slave 0
- Master 0 asserts PENABLE
- *Grant could change here* ← BREAKS PROTOCOL!
- Slave 0 responds to wrong master

WITH Grant Persistence:
- Master 0 granted Slave 0
- Master 0 asserts PENABLE
- Grant held until PREADY asserted ← SAFE
- Slave 0 responds to correct master
- Grant released for next transaction
```

Implementation: Grant signal registered and held from PSEL assertion through PREADY response

Arbitration Latency

Best Case (No Contention): - 0 cycles arbitration delay - Request → Grant in same cycle - APB protocol overhead: 2 cycles minimum (PSEL + PENABLE)

Worst Case (Maximum Contention): - M-1 cycles wait time (other masters ahead in round-robin) - Example: 2 masters, worst case = 1 cycle wait - Example: 4 masters, worst case = 3 cycles wait

Average Case (Random Access Pattern): - (M-1)/2 cycles average wait time - Statistical fairness over time

---

Integration Examples

Example 1: CPU to 4 Peripherals (No Contention)

```
apb_xbar_1to4 #(
    .BASE_ADDR(32'h1000_0000)
```

```
) u_periph_xbar (
    // Single master (CPU)
    .m0_apb_* (cpu_apb_*),

    // 4 slaves (UART, GPIO, Timer, SPI)
    .s0_apb_* (uart_*),   // 0x1000_0000
    .s1_apb_* (gpio_*),   // 0x1001_0000
    .s2_apb_* (timer_*),  // 0x1002_0000
    .s3_apb_* (spi_*)     // 0x1003_0000
);
```

Behavior: - No arbitration needed (single master) - Pure address decode functionality - Zero arbitration overhead

Example 2: CPU + DMA to 4 Peripherals (Potential Contention)

```
apb_xbar_2to4 #(
    .BASE_ADDR(32'h4000_0000)
) u_soc_xbar (
    // Two masters
    .m0_apb_* (cpu_apb_*),
    .m1_apb_* (dma_apb_*),

    // 4 slaves
    .s0_apb_* (mem_ctrl_*),  // 0x4000_0000
    .s1_apb_* (uart_*),      // 0x4001_0000
    .s2_apb_* (i2c_*),       // 0x4002_0000
    .s3_apb_* (adc_*)        // 0x4003_0000
);
```

Behavior: - Each slave has independent arbiter - CPU and DMA can access different slaves simultaneously - If both access same slave, round-robin arbitration - Fair access guaranteed

---

Performance Characteristics

Throughput

Single Master: - Back-to-back transactions supported - Zero bubble (no idle cycles between transactions) - Limited only by slave PREADY response time

Multiple Masters (Same Slave): - Round-robin introduces fair sharing - Each master gets ~1/M of bandwidth - Example: 2 masters = 50% bandwidth each

Multiple Masters (Different Slaves): - Full parallelism - Each master gets 100% of its target slave - Total system bandwidth = NUM_SLAVES × slave_bandwidth

Latency

Components:

1. **Address Decode:** 0 cycles (combinational, parallel)
2. **Arbitration Decision:** 0 cycles (combinational)
3. **APB Protocol:** 2 cycles minimum (PSEL + PENABLE)
4. **Slave Response:** Variable (depends on slave)

Total Minimum Latency: 2 cycles (uncontended access)

With Contention: +1 to +(M-1) cycles arbitration wait time

Design Notes

Why 64KB Per Slave?

Rationale: - Sufficient for most APB peripherals (typically 256B - 4KB register space) - Simple decode logic (single shift operation) - Allows up to 16 slaves with clean byte alignment - Software-friendly (each peripheral has “round” base address)

Alternatives: - Smaller regions (4KB, 16KB) → more slaves, more complex decode - Larger regions (256KB, 1MB) → fewer slaves, wasted address space

Why Round-Robin?

Advantages: - Fair: Prevents starvation - Simple: Minimal logic (counter + comparator) - Predictable: Software can reason about worst-case latency - No configuration: Works out-of-box

Alternatives: - Fixed priority → Can starve low-priority masters - Weighted fair queuing → More complex, requires configuration - Lottery scheduling → Unpredictable, harder to verify

---

Troubleshooting

Issue: Wrong Slave Selected

Check: 1. BASE_ADDR parameter correct? 2. Address within expected range? 3. Calculated slave_index matches expectation?

Debug:

```
offset = PADDR - BASE_ADDR
slave_index = offset >> 16
expected_slave = slave_index  // Should match actual PSEL
```

Issue: Master Starved

Check: 1. Other masters continuously accessing same slave? 2. Arbitration timeout monitoring enabled? 3. Round-robin priority rotating correctly?

Expected Behavior: - Each master should get grant within M transactions - No master should wait indefinitely

Issue: Back-to-Back Transactions Stalling

Check: 1. Grant persistence working? 2. Slave asserting PREADY correctly? 3. No unintended pipeline bubbles?

Expected Behavior: - Consecutive transactions from same master should flow without gaps - Only arbitration conflicts should introduce wait states

---

Next Steps

- See Architecture for top-level design overview
- See PRD.md for complete specification
- See CLAUDE.md for integration guidance
- See README.md for quick start guide

Version: 1.0 **Last Updated:** 2025-10-25 **Maintained By:** RTL Design Sherpa Project

APB Crossbar RTL Generator

Component: APB Crossbar (MxN Interconnect) **Version:** 1.0 **Status:** Production Ready

---

Overview

The APB Crossbar uses **parametric RTL generation** to create custom MxN configurations. While several pre-generated variants exist (1to1, 2to1, 1to4, 2to4, thin), the Python-based generator enables creation of any arbitrary MxN crossbar up to 16x16.

Philosophy: - Pre-generated variants for common use cases (fast integration) - Python generator for custom configurations (flexibility) - Template-based code generation (consistency) - Built from proven components (quality)

Generator Architecture

Two-Level Structure

```
projects/components/apb_xbar/
├── bin/
│   └── generate_xbars.py              ← Convenience wrapper script
│
└── (repo root)/bin/rtl_generators/amba/
    └── apb_xbar_generator.py          ← Core generator engine
```

Why Two Files? 1. **Core Generator** (`apb_xbar_generator.py`): Reusable library function - Can be imported by other scripts - Contains `generate_apb_xbar()` function - Located in central `bin/rtl_generators/` directory

2. **Convenience Wrapper** (`generate_xbars.py`): Project-specific script
   - Easy command-line interface
   - Pre-configured for standard variants
   - Located in project `bin/` directory

Quick Start

Generate Standard Variants (1:1, 2:1, 1:4, 2:4)

```
cd projects/components/apb_xbar/bin/
python generate_xbars.py
```

Output:

```
Generating 1-to-1 crossbar...
  ✓ apb_xbar_1to1.sv
Generating 2-to-1 crossbar...
  ✓ apb_xbar_2to1.sv
Generating 1-to-4 crossbar...
```

```
  ✓ apb_xbar_1to4.sv
Generating 2-to-4 crossbar...
  ✓ apb_xbar_2to4.sv

✓ Generated 4 crossbar variants
```

Generate Custom Variant

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

Output:

```
Generating 3-to-6 crossbar...
✓ Generated apb_xbar_3to6.sv
```

Generate with Custom Base Address

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

Output:

```
Generating 4-to-8 crossbar...
✓ Generated apb_xbar_4to8.sv

Address Map:
  Slave 0: [0x80000000, 0x8000FFFF]
  Slave 1: [0x80010000, 0x8001FFFF]
  Slave 2: [0x80020000, 0x8002FFFF]
  ...
  Slave 7: [0x80070000, 0x8007FFFF]
```

Command-Line Interface

Full Options

```
python generate_xbars.py [OPTIONS]
```

Options:

Option	Short	Type	Default	Description
`--masters`	`-m`	int	None	Number of master interfaces (1-16)
`--slaves`	`-s`	int	None	Number of slave interfaces (1-16)
`--base-addr`	`-b`	hex	0x10000000	Base address for slave address map

Option	Short	Type	Default	Description
`--help`	`-h`	-	-	Show help message

Rules: - If neither `--masters` nor `--slaves` specified → Generate all standard variants - If only one specified → Error (must specify both) - If both specified → Generate custom variant

Examples

Example 1: Generate All Standard Variants

```
python generate_xbars.py
```

Example 2: Generate 3x6 Crossbar

```
python generate_xbars.py --masters 3 --slaves 6
```

Example 3: Generate 5x10 Crossbar with Custom Base

```
python generate_xbars.py -m 5 -s 10 -b 0xA0000000
```

Example 4: Show Help

```
python generate_xbars.py --help
```

Generated Code Structure

Template Components

The generator creates a complete SystemVerilog module with:

1. **Header with Address Map**

```
// 3-to-6 APB crossbar with address decoding and arbitration
// 3 masters to 6 slaves using apb_slave and apb_master modules
//
// Address Map (same for all masters):
//   Slave 0: [0x10000000, 0x1000FFFF]
//   Slave 1: [0x10010000, 0x1001FFFF]
//   Slave 2: [0x10020000, 0x1002FFFF]
//   Slave 3: [0x10030000, 0x1003FFFF]
//   Slave 4: [0x10040000, 0x1004FFFF]
//   Slave 5: [0x10050000, 0x1005FFFF]
```

2. **Module Declaration with Parameters**

```
module apb_xbar_3to6 #(
    parameter int ADDR_WIDTH = 32,
    parameter int DATA_WIDTH = 32,
```

```
    parameter int STRB_WIDTH = DATA_WIDTH / 8,
    parameter logic [ADDR_WIDTH-1:0] BASE_ADDR = 32'h10000000
) (
    // Clock and Reset
    input  logic                  pclk,
    input  logic                  presetn,

    // Master 0-2 APB interfaces
    // Slave 0-5 APB interfaces
);
```

3. **Internal Signal Declarations**

```
    // Command/Response interfaces for master 0-2 apb_slave
    logic m0_cmd_valid, m0_cmd_ready, m0_cmd_pwrite;
    logic [ADDR_WIDTH-1:0] m0_cmd_paddr;
    // ... (full cmd/rsp bus for each master)

    // Command/Response interfaces for slave 0-5 apb_master
    logic s0_cmd_valid, s0_cmd_ready, s0_cmd_pwrite;
    logic [ADDR_WIDTH-1:0] s0_cmd_paddr;
    // ... (full cmd/rsp bus for each slave)
```

4. **Master-Side apb_slave Instantiations**

```
    // Master 0 apb_slave
    apb_slave #(
        .ADDR_WIDTH(ADDR_WIDTH),
        .DATA_WIDTH(DATA_WIDTH)
    ) u_m0_slave (
        .pclk(pclk),
        .presetn(presetn),
        // APB interface
        .apb_PSEL(m0_apb_PSEL),
        .apb_PENABLE(m0_apb_PENABLE),
        // ... (full APB connections)
        // cmd/rsp interface
        .cmd_valid(m0_cmd_valid),
        .cmd_ready(m0_cmd_ready),
        // ... (full cmd/rsp connections)
    );

    // Master 1, 2 apb_slave instances...
```

5. **Address Decode Logic**

```
    // Address decode for each master
    logic [M-1:0][N-1:0] m_to_s_req;  // Request matrix

    // Master 0 address decode
    always_comb begin
```

```
        m_to_s_req[0] = '0;
        if (m0_cmd_valid) begin
            logic [ADDR_WIDTH-1:0] offset = m0_cmd_paddr - BASE_ADDR;
            logic [3:0] slave_idx = offset[19:16];
            case (slave_idx)
                4'd0: m_to_s_req[0][0] = 1'b1;
                4'd1: m_to_s_req[0][1] = 1'b1;
                // ... (case for each slave)
            endcase
        end
    end

    // Master 1, 2 address decode...
```

6. **Per-Slave Arbitration Logic**

```
    // Arbitration for each slave
    logic [N-1:0][M-1:0] s_grant;  // Grant matrix
    logic [N-1:0][$clog2(M)-1:0] s_arb_priority;  // Round-robin
priority

    // Slave 0 arbiter (round-robin among M masters)
    always_ff @(posedge pclk or negedge presetn) begin
        if (!presetn) begin
            s_arb_priority[0] <= '0;
            s_grant[0] <= '0;
        end else begin
            // Round-robin arbitration logic
            // Priority rotation after each grant
        end
    end

    // Slave 1-5 arbiters...
```

7. **Response Routing Logic**

```
    // Route responses back to requesting masters
    always_comb begin
        // Track which master initiated each transaction
        // Route PRDATA/PSLVERR back to correct master
    end
```

8. **Slave-Side apb_master Instantiations**

```
    // Slave 0 apb_master
    apb_master #(
        .ADDR_WIDTH(ADDR_WIDTH),
        .DATA_WIDTH(DATA_WIDTH)
    ) u_s0_master (
        .pclk(pclk),
        .presetn(presetn),
        // cmd/rsp interface
```

```
        .cmd_valid(s0_cmd_valid),
        .cmd_ready(s0_cmd_ready),
        // ... (full cmd/rsp connections)
        // APB interface
        .apb_PSEL(s0_apb_PSEL),
        .apb_PENABLE(s0_apb_PENABLE),
        // ... (full APB connections)
    );

    // Slave 1-5 apb_master instances...
```

9. **Module End**

```
endmodule : apb_xbar_3to6
```

Generator Algorithm

Step-by-Step Process

1. Validate Parameters

```
if M < 1 or M > 16:
    raise ValueError(f"Number of masters must be 1-16, got {M}")
if N < 1 or N > 16:
    raise ValueError(f"Number of slaves must be 1-16, got {N}")
```

2. Calculate Derived Parameters

```
slave_addr_bits = max(1, math.ceil(math.log2(N)))  # Bits for slave
index
strb_width = data_width // 8                        # Byte strobe
width
module_name = f"apb_xbar_{M}to{N}"                  # Module name
```

3. Generate Address Map Documentation

```
for s in range(N):
    addr_offset = s * 0x10000  # 64KB per slave
    addr_start = base_addr + addr_offset
    addr_end = addr_start + 0xFFFF
    # Generate comment: Slave S: [START, END]
```

4. Generate Module Declaration - Parameters: ADDR_WIDTH, DATA_WIDTH, STRB_WIDTH, BASE_ADDR - Ports: Clock, reset - Master interfaces: M × full APB interface - Slave interfaces: N × full APB interface

5. Generate Internal Signals - Master cmd/rsp buses: M sets - Slave cmd/rsp buses: N sets - Request matrix: [M][N] - Grant matrix: [N][M] - Priority counters: [N]

6. Instantiate Master-Side Components

```
for m in range(M):
    # Generate apb_slave instance
    # Connect APB interface to external master m
    # Connect cmd/rsp to internal buses
```

7. Generate Address Decode Logic

```
for m in range(M):
    # For each master, decode address to slave index
    # offset = paddr - BASE_ADDR
    # slave_idx = offset[19:16]  # Upper 4 bits
    # Set m_to_s_req[m][slave_idx] = 1
```

8. Generate Arbitration Logic

```
for s in range(N):
    # For each slave, arbitrate among M masters
    # Round-robin: priority rotates after each grant
    # Grant persistence: hold until transaction completes
```

9. Generate Response Routing

```
# Track transaction source (which master)
# Route responses back to originating master
```

10. Instantiate Slave-Side Components

```
for s in range(N):
    # Generate apb_master instance
    # Connect cmd/rsp from internal buses
    # Connect APB interface to external slave s
```

11. Write Generated Code

```
with open(output_file, 'w') as f:
    f.write(generated_code)
```

Code Generation Patterns

Loop-Based Generation

Master Interfaces:

```
for m in range(M):
    code += f"    // Master {m} APB interface\n"
    code += f"    input  logic m{m}_apb_PSEL,\n"
    code += f"    input  logic m{m}_apb_PENABLE,\n"
    # ... (all APB signals for master m)
```

Slave Interfaces:

```
for s in range(N):
    code += f"    // Slave {s} APB interface\n"
    code += f"    output logic s{s}_apb_PSEL,\n"
    code += f"    output logic s{s}_apb_PENABLE,\n"
    # ... (all APB signals for slave s)
```

String Formatting

Hex Values:

```
code += f"    parameter logic [ADDR_WIDTH-1:0] BASE_ADDR =
{addr_width}'h{base_addr:08X}\n"
# Example: 32'h10000000
```

Address Map Comments:

```
addr_start = base_addr + (s * 0x10000)
code += f"//   Slave {s}: [0x{addr_start:08X}, 0x{addr_start +
0xFFFF:08X}]\n"
# Example: Slave 2: [0x10020000, 0x1002FFFF]
```

Customization Examples

Example 1: Large SoC with 10 Peripherals

Scenario: CPU + DMA need to access 10 peripherals

Command:

```
python generate_xbars.py --masters 2 --slaves 10
```

Generated File: `apb_xbar_2to10.sv`

Address Map:

```
Slave 0:  0x10000000 - 0x1000FFFF (UART0)
Slave 1:  0x10010000 - 0x1001FFFF (UART1)
Slave 2:  0x10020000 - 0x1002FFFF (GPIO)
Slave 3:  0x10030000 - 0x1003FFFF (I2C)
Slave 4:  0x10040000 - 0x1004FFFF (SPI)
Slave 5:  0x10050000 - 0x1005FFFF (Timer0)
```

```
Slave 6:  0x10060000 - 0x1006FFFF (Timer1)
Slave 7:  0x10070000 - 0x1007FFFF (PWM)
Slave 8:  0x10080000 - 0x1008FFFF (ADC)
Slave 9:  0x10090000 - 0x1009FFFF (Watchdog)
```

LOC: ~2500 lines (estimated)

Example 2: Multi-CPU System

Scenario: 4 CPUs accessing shared peripheral bus with 4 slaves

Command:

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

Generated File: `apb_xbar_4to4.sv`

Benefits: - Fair arbitration among 4 CPUs - Each slave independently arbitrated - Maximum throughput when CPUs access different slaves

LOC: ~1600 lines (estimated)

Example 3: Memory-Mapped I/O Region

Scenario: Single master accessing I/O region at 0xA0000000

Command:

```
python generate_xbars.py --masters 1 --slaves 8 --base-addr 0xA0000000
```

Generated File: `apb_xbar_1to8.sv`

Address Map:

```
Slave 0:  0xA0000000 - 0xA000FFFF
Slave 1:  0xA0010000 - 0xA001FFFF
...
Slave 7:  0xA0070000 - 0xA007FFFF
```

Using Generated Modules

Integration Example

```systemverilog
// Generated file: apb_xbar_3to6.sv

module my_soc (
    input  logic clk,
    input  logic rst_n,
    // ... other interfaces
```

```
);

    // Instantiate 3x6 crossbar
    apb_xbar_3to6 #(
        .ADDR_WIDTH(32),
        .DATA_WIDTH(32),
        .BASE_ADDR(32'h10000000)  // Can override at instantiation
    ) u_periph_xbar (
        .pclk(clk),
        .presetn(rst_n),

        // Connect 3 masters (CPU, DMA, Debug)
        .m0_apb_PSEL(cpu_apb_psel),
        .m0_apb_PENABLE(cpu_apb_penable),
        // ... (full CPU APB interface)

        .m1_apb_PSEL(dma_apb_psel),
        .m1_apb_PENABLE(dma_apb_penable),
        // ... (full DMA APB interface)

        .m2_apb_PSEL(debug_apb_psel),
        .m2_apb_PENABLE(debug_apb_penable),
        // ... (full Debug APB interface)

        // Connect 6 slaves (UART, GPIO, Timer, SPI, I2C, ADC)
        .s0_apb_PSEL(uart_psel),
        .s0_apb_PENABLE(uart_penable),
        // ... (full UART APB interface)

        .s1_apb_PSEL(gpio_psel),
        // ... (remaining slaves)
    );

endmodule : my_soc
```

Parameter Override

```
// Use different base address at instantiation
apb_xbar_3to6 #(
    .BASE_ADDR(32'h80000000)  // Override default 0x10000000
) u_mmio_xbar (
    // ... connections
);
// Slaves now at: 0x80000000, 0x80010000, 0x80020000, ...
```

Generator Limitations and Future Work

Current Limitations

1. **Fixed 64KB Per Slave**
   - Cannot customize individual slave region sizes
   - Workaround: Use multiple crossbars with different BASE_ADDR
2. **Max 16x16 Configuration**
   - Practical limit due to arbitration complexity
   - Workaround: Hierarchical crossbars
3. **Single Address Map**
   - All masters see same slave address map
   - Workaround: Address translation before crossbar
4. **No Priority Levels**
   - Round-robin only, no fixed priority or weighted arbitration
   - Workaround: External priority arbitration before crossbar

Future Enhancements

1. Variable Slave Sizes

```
# Potential future API
generate_apb_xbar(
    num_masters=2,
    slave_sizes=[4096, 65536, 1024, 16384],  # Custom sizes per slave
    base_addr=0x10000000
)
```

2. Arbitration Policies

```
# Potential future API
generate_apb_xbar(
    num_masters=4,
    num_slaves=4,
    arbitration="weighted",  # "round-robin", "fixed-priority",
"weighted"
    weights=[4, 2, 1, 1]     # Master 0 gets 4x priority
)
```

3. Address Translation

```
# Potential future API
generate_apb_xbar(
    num_masters=2,
    num_slaves=4,
    master_views=[
```

```
        {"base": 0x00000000, "remap": True},   # Master 0 sees 0x0
        {"base": 0x10000000, "remap": False}   # Master 1 sees
0x1000_0000
    ]
)
```

Testing Generated Code

Verification Flow

1. Generate Crossbar

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

2. Lint Generated RTL

```
cd ../../rtl/
verilator --lint-only apb_xbar_2to4.sv
```

3. Run Functional Tests

```
cd ../../dv/tests/
pytest test_apb_xbar_2to4.py -v
```

4. View Waveforms (If Test Fails)

```
gtkwave waves.vcd
```

Recommended Test Cases

1. **Single Master Access** - Verify basic routing
2. **Multi-Master Contention** - Verify arbitration
3. **Back-to-Back Transactions** - Verify zero-bubble throughput
4. **Address Decode Boundary** - Verify slave selection at boundaries
5. **Error Response Propagation** - Verify PSLVERR routing

Advanced Usage

Using as Python Library

```
# Import generator function
from bin.rtl_generators.amba.apb_xbar_generator import
generate_apb_xbar

# Generate crossbar programmatically
code = generate_apb_xbar(
    num_masters=3,
```

```
    num_slaves=8,
    base_addr=0xA0000000,
    addr_width=32,
    data_width=32,
    output_file="my_custom_xbar.sv"
)

# Generated code is returned as string
print(f"Generated {len(code)} bytes of SystemVerilog code")

# Can also write to file manually
with open("custom_output.sv", "w") as f:
    f.write(code)
```

Batch Generation

```
# Generate multiple variants in one script
variants = [
    (1, 4, 0x10000000),
    (2, 4, 0x20000000),
    (3, 6, 0x30000000),
    (4, 8, 0x40000000),
]

for masters, slaves, base_addr in variants:
    code = generate_apb_xbar(
        num_masters=masters,
        num_slaves=slaves,
        base_addr=base_addr,
        output_file=f"xbar_{masters}to{slaves}_0x{base_addr:08X}.sv"
    )
    print(f"✓ Generated
xbar_{masters}to{slaves}_0x{base_addr:08X}.sv")
```

Troubleshooting

Issue: “ModuleNotFoundError: No module named ‘amba.apb_xbar_generator’”

Cause: Python can’t find the core generator module

Solution:

```
# Verify generator exists
ls -la bin/apb_xbar_generator.py

# Run from bin directory
cd projects/components/apb_xbar/bin/
python generate_xbars.py
```

Issue: Generated Code Won't Lint

Cause: Generator bug or unsupported configuration

Solution: 1. Check parameter ranges (1-16 for M and N) 2. Verify base address alignment (should be aligned to 64KB × N) 3. Report bug with command that caused issue

Issue: Generated Code Too Large

Cause: Very large MxN configuration (e.g., 16x16)

Expected Behavior: - 1x1: ~200 LOC - 2x4: ~1000 LOC - 16x16: ~25,000 LOC (practical limit)

Solution: - Consider hierarchical approach (multiple smaller crossbars) - Or use different interconnect topology (bus, tree, etc.)

References

- Architecture Overview
- Address Decode and Arbitration
- PRD.md - Complete specification
- CLAUDE.md - Integration guidance
- README.md - Quick start guide

Version: 1.0 **Last Updated:** 2025-10-25 **Maintained By:** RTL Design Sherpa Project

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

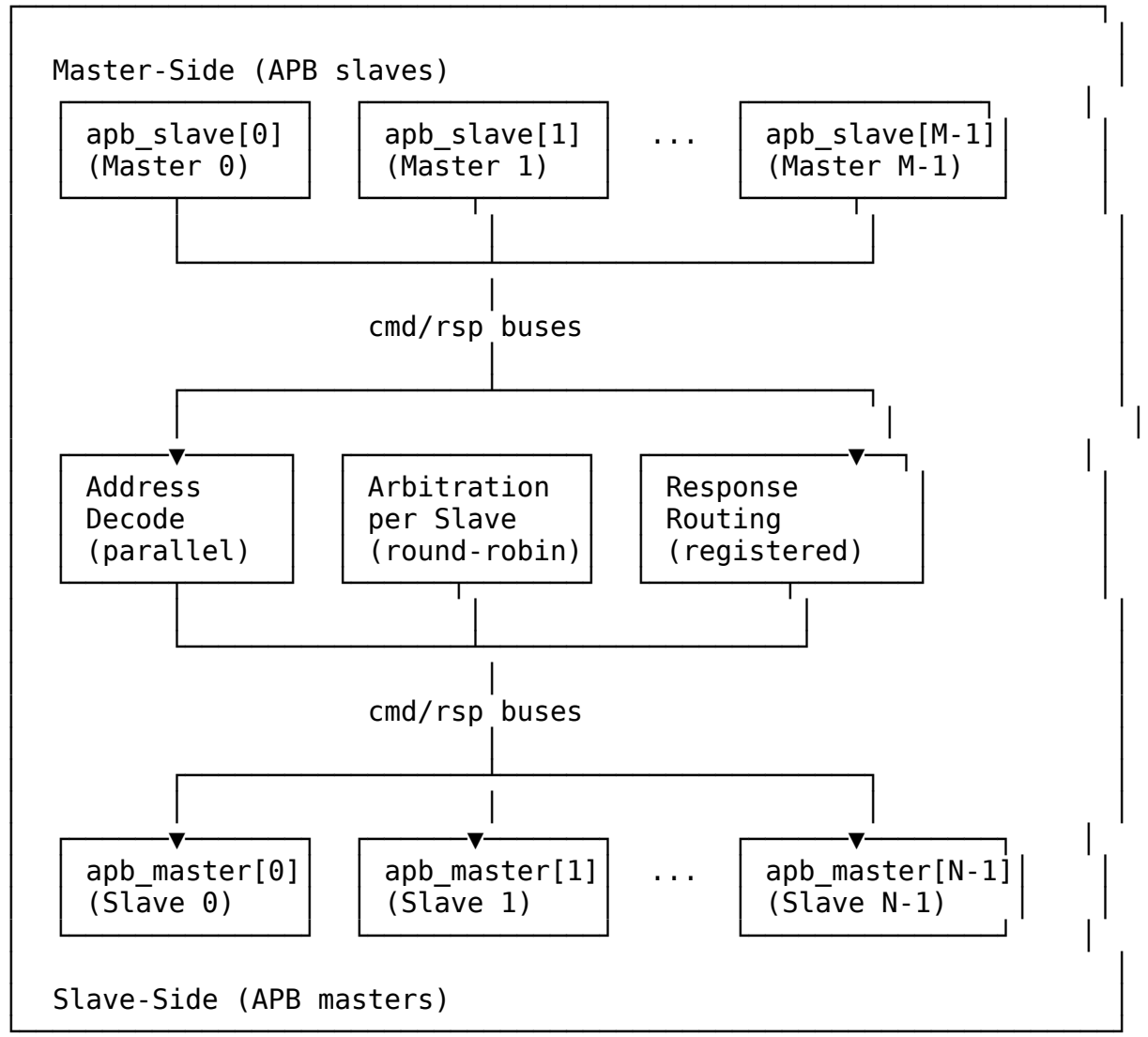
2.2 Design Trade-offs

Feature	Choice	Rationale
Arbitration	Round-robin per slave	Fair, simple, predictable

Feature	Choice	Rationale
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



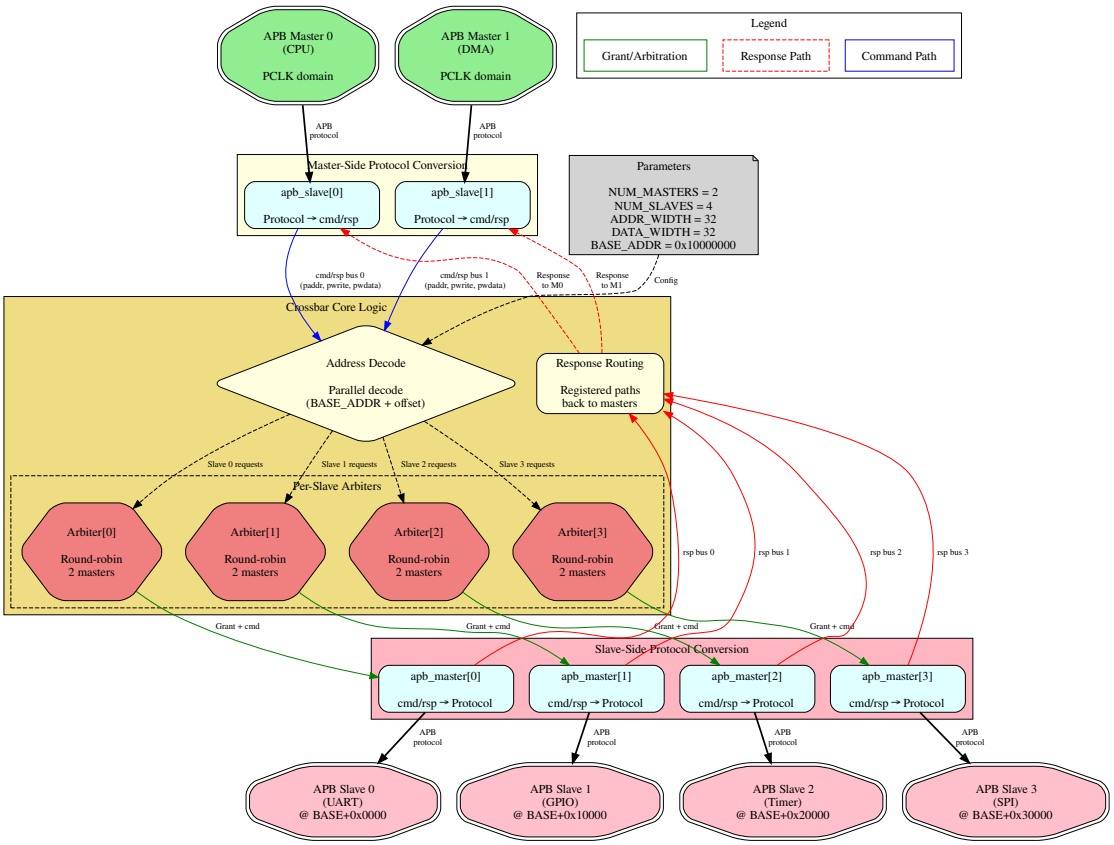


Figure: APB Crossbar top-level architecture showing 2 masters connected to 4 slaves. Source: docs/apb_xbar_spec/assets/graphviz/apb_xbar_architecture.gv | SVG

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)
```

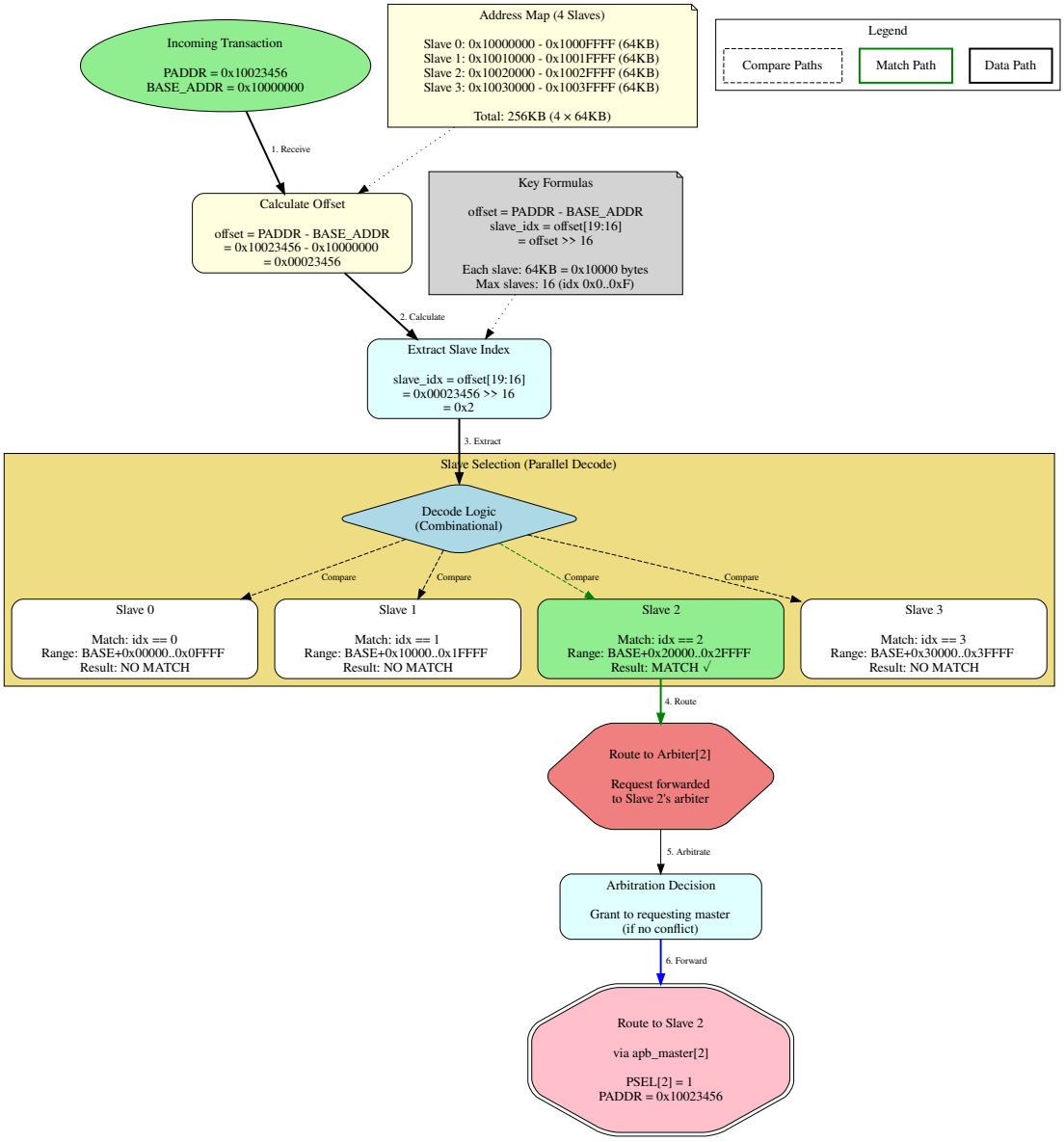


Figure: Address decode flow showing how address 0x10023456 routes to Slave 2.
Source: docs/apb_xbar_spec/assets/graphviz/address_decode_flow.gv | SVG

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
```

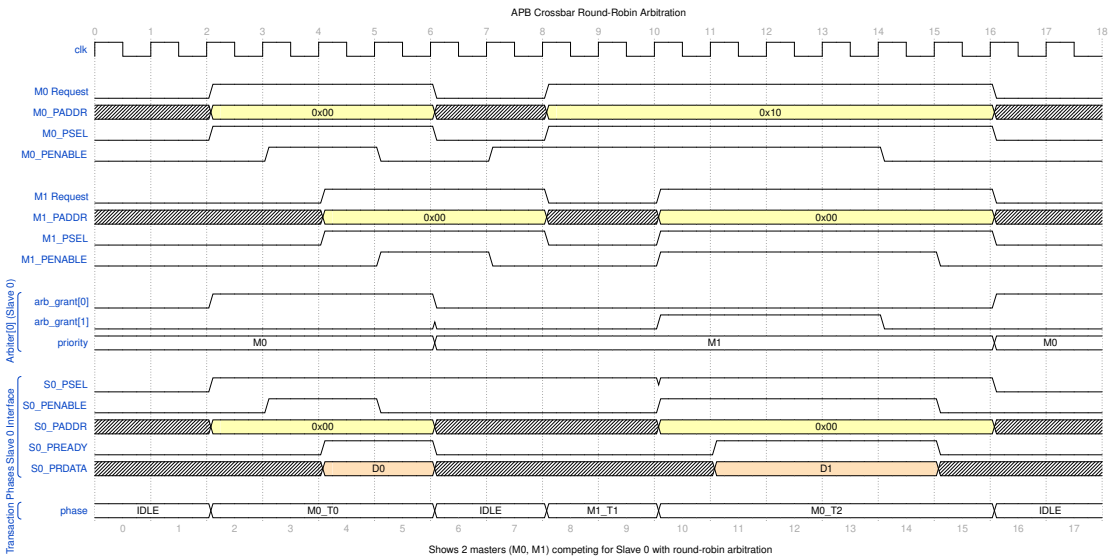


Figure: Round-robin arbitration timing showing 2 masters competing for Slave 0. Master priority rotates after each grant to ensure fair access. Source: docs/apb_xbar_spec/assets/wavedrom/arbitration_round_robin.json

---

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	350+	√ Pass	Full crossbar

Test	Transactions	Status	Coverage
_2to4			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

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:

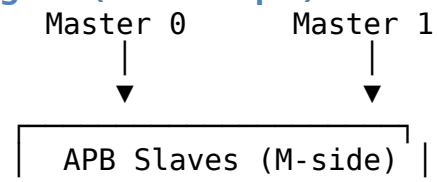
```
# Generate 3x8 crossbar with custom base
python generate_xbars.py --masters 3 --slaves 8 --base-addr 0x80000000

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

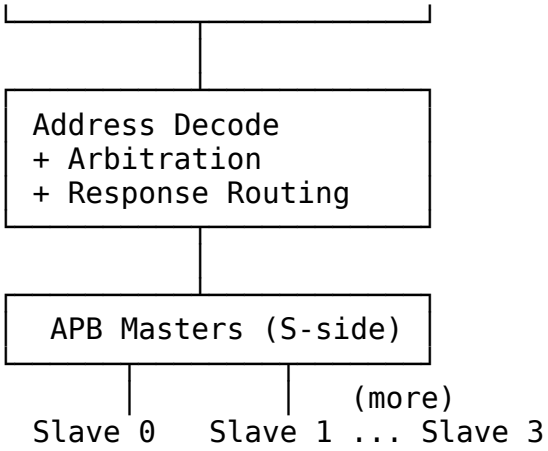
Architecture Quick Reference

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

APB Crossbar Modules

This directory contains APB crossbar modules generated using the proven `apb_slave` and `apb_master` architecture.

Architecture

All crossbars follow a consistent design:

```
Master Side:     apb_slave modules convert APB → cmd/rsp interface
Internal:        Round-robin arbitration + address decoding
Slave Side:      apb_master modules convert cmd/rsp → APB interface
```

Key Features

- **Independent arbitration per slave**: Each slave has its own round-robin arbiter
- **Grant persistence**: Grants held from command acceptance through response completion
- **Address decoding**: Automatic slave selection based on address ranges
- **Proven architecture**: Uses production-tested apb_slave/apb_master modules

Available Modules

Module	Description	Masters	Slaves	Use Case
`apb_xbar_1to1.sv`	Simple passthrough	1	1	Protocol conversion, testing
`apb_xbar_2to1.sv`	Arbitration only	2	1	Multi-master to single slave
`apb_xbar_1to4.sv`	Address decode only	1	4	Single master to multi-slave
`apb_xbar_2to4.sv`	Full crossbar	2	4	Multi-master to multi-slave

Generating Crossbars

Method 1: Convenience Script (Recommended)

Generate all standard variants:

```
cd rtl/amba/apb/xbar/
python generate_xbars.py
```

Generate custom variant:

```
python generate_xbars.py --masters 3 --slaves 6
python generate_xbars.py --masters 4 --slaves 8 --base-addr 0x80000000
```

Method 2: Direct Generator

Use the main generator for more control:

```
cd bin/rtl_generators/amba/
python apb_xbar_generator.py --masters 2 --slaves 4 --
output ../../../rtl/amba/apb/xbar/apb_xbar_2to4.sv
```

Address Map

All crossbars use a uniform address mapping (configurable via `BASE_ADDR` parameter):

```
Slave 0: [BASE_ADDR + 0x0000_0000, BASE_ADDR + 0x0000_FFFF]  (64KB)
Slave 1: [BASE_ADDR + 0x0001_0000, BASE_ADDR + 0x0001_FFFF]  (64KB)
Slave 2: [BASE_ADDR + 0x0002_0000, BASE_ADDR + 0x0002_FFFF]  (64KB)
Slave 3: [BASE_ADDR + 0x0003_0000, BASE_ADDR + 0x0003_FFFF]  (64KB)
...
```

Default BASE_ADDR: `0x1000_0000`

Parameters

All modules support these parameters:

Parameter	Default	Description
`ADDR_WIDTH`	32	Address bus width
`DATA_WIDTH`	32	Data bus width
`STRB_WIDTH`	DATA_WIDTH/8	Strobe width (auto-calculated)
`BASE_ADDR`	0x10000000	Base address for slave address map

Usage Example

```
apb_xbar_2to4 #(
    .ADDR_WIDTH (32),
    .DATA_WIDTH (32),
    .BASE_ADDR  (32'h8000_0000)
) u_xbar (
    .pclk       (apb_clk),
    .presetn    (apb_rst_n),

    // Master 0 interface
    .m0_apb_PSEL    (m0_psel),
    .m0_apb_PENABLE (m0_penable),
    .m0_apb_PADDR   (m0_paddr),
    .m0_apb_PWRITE  (m0_pwrite),
    .m0_apb_PWDATA  (m0_pwdata),
    .m0_apb_PSTRB   (m0_pstrb),
    .m0_apb_PPROT   (m0_pprot),
    .m0_apb_PRDATA  (m0_prdata),
    .m0_apb_PSLVERR (m0_pslverr),
    .m0_apb_PREADY  (m0_pready),

    // Master 1 interface
    .m1_apb_PSEL    (m1_psel),
    // ... (similar connections)

    // Slave 0-3 interfaces
    .s0_apb_PSEL    (s0_psel),
    // ... (similar connections)
);
```

Testing

All generated crossbars have corresponding test files in `val/integ_amba/`:

- `test_apb_xbar_1to1.py` - 100+ transactions, variable delay profiles
- `test_apb_xbar_2to1.py` - 130+ transactions, arbitration stress tests
- `test_apb_xbar_1to4.py` - 200+ transactions, address decode validation
- `test_apb_xbar_2to4.py` - 350+ transactions, full crossbar stress

Run tests:

```
pytest val/integ_amba/test_apb_xbar_2to4.py -v
pytest val/integ_amba/test_apb_xbar_*.py -v  # All variants
```

Design Notes

Arbitration Strategy

- **Round-robin per slave**: Master priority rotates (M0→M1→M0...)
- **Grant persistence**: Once granted, master owns slave until transaction completes
- **Fairness**: No master can starve another master

Address Decoding

- **Parallel decode**: All masters decode addresses simultaneously
- **Registered routing**: Slave selection registered at command acceptance
- **Response routing**: Based on registered slave selection

Timing

- **Zero bubble overhead**: Back-to-back transactions supported
- **Single-cycle arbitration**: New grants issued in same cycle as previous completion
- **Pipelined datapath**: Command and response phases overlap different transactions

Known Limitations

1. **Fixed address map**: 64KB regions per slave
   - Can be changed by modifying generator's `addr_offset` calculation
2. **No slave disable**: All slaves always active
   - Could add enable parameter if needed
3. **No timeout handling**: Assumes slaves always respond
   - Add watchdog if needed for unreliable slaves

Generating Custom Variants

For variants beyond 16x16, modify generator limits in `apb_xbar_generator.py`:

```python
if M < 1 or M > 16:  # Change 16 to desired max
    raise ValueError(f"Number of masters must be 1-16, got {M}")
```

Files

- `generate_xbars.py` - Convenience script for generation
- `apb_xbar_1to1.sv` - 1-to-1 passthrough
- `apb_xbar_2to1.sv` - 2-to-1 with arbitration
- `apb_xbar_1to4.sv` - 1-to-4 with address decode
- `apb_xbar_2to4.sv` - 2-to-4 full crossbar
- `README.md` - This file

References

- APB Specification: ARM IHI 0024C (AMBA APB Protocol v2.0)
- Generator: `projects/components/apb_xbar/bin/apb_xbar_generator.py`
- Base modules: `rtl/amba/apb/apb_slave.sv`, `rtl/amba/apb/apb_master.sv`
- Tests: `val/integ_amba/test_apb_xbar_*.py`

Generated by RTL Design Sherpa | **Last Updated:** 2025-10-14