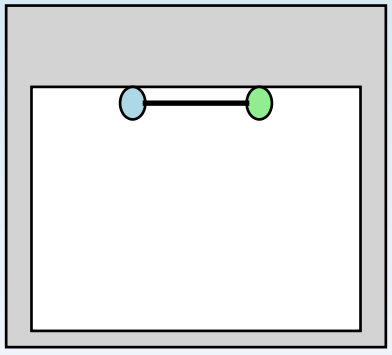
AMBA Bus Matrix GUI Complete User Guide

Step-by-Step GUI Workflow



Version 1.0.0 - Complete Edition
July 2025

Table of Contents

GUI Overview

What the GUI Does:

- Visual design of AMBA bus interconnects
- Drag-and-drop master and slave configuration
- Real-time validation and error checking
- Automatic RTL generation (Verilog)
- Complete UVM VIP environment generation
- Project save/load functionality

Complete GUI Workflow:

1. Launch GUI → 2. Create/Load Project → 3. Add Masters

4. Add Slaves → 5. Make Connections → 6. Configure Parameters

7. Validate Design → 8. Generate RTL → 9. Generate VIP

Key Benefits:

✓ No command-line knowledge required
✓ Visual validation prevents errors
✓ Generates production-ready code
✓ Complete verification environment

Launching the GUI

Method 1: Shell Script (Recommended)

Step 1: Open terminal in project directory

Step 2: Navigate to GUI folder:

```
cd axi4_vip/gui
```

Step 3: Run launch script:

```
./launch_gui.sh
```

Method 2: Direct Python Launch

Step 1: Ensure Python 3.6+ is installed:

```
python3 --version
```

Step 2: Install dependencies (if needed):

```
pip3 install -r requirements.txt
```

Step 3: Launch GUI directly:

```
python3 src/bus_matrix_gui.py
```

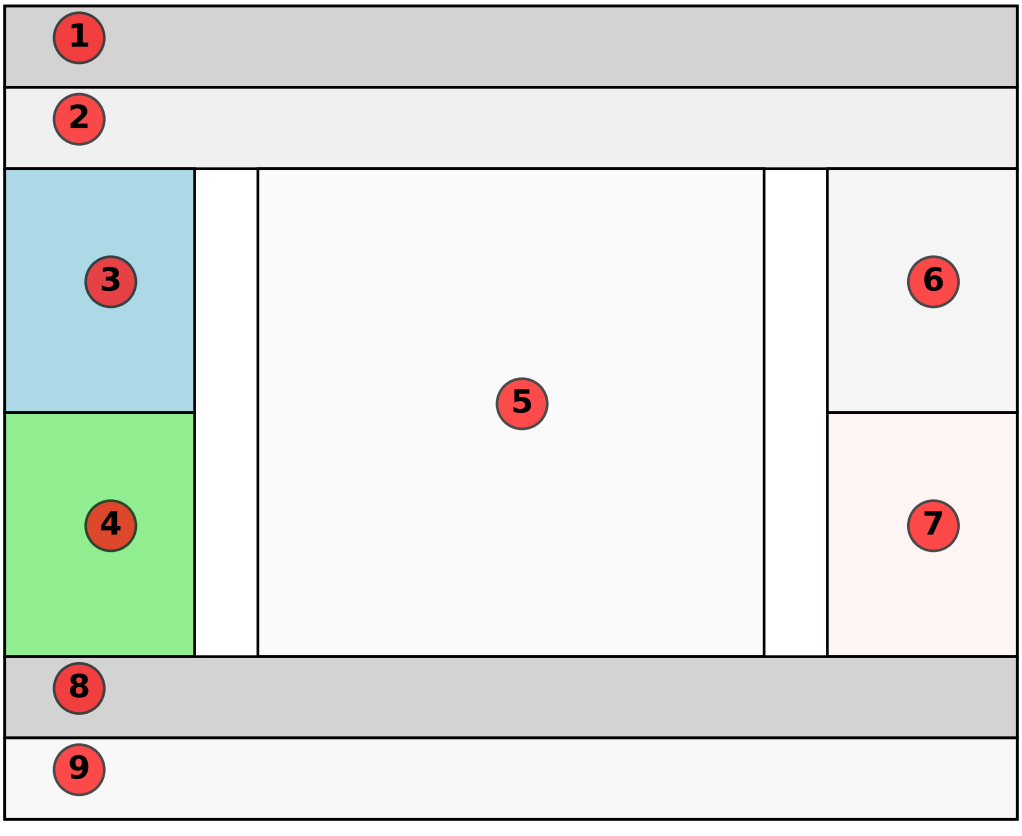
Common Launch Issues:

Problem: 'tkinter not found'
Solution: sudo apt-get install python3-tk

Problem: 'Permission denied'
Solution: chmod +x launch_gui.sh

Problem: GUI window doesn't appear
Solution: Check DISPLAY variable: echo $DISPLAY

GUI Interface Tour



Interface Elements:

1. **Menu Bar** *File, Edit, View, Tools, Help menus*
2. **Toolbar** *Quick access: New, Open, Save, Generate*
3. **Master Panel** *Add/configure master components*
4. **Slave Panel** *Add/configure slave components*
5. **Design Canvas** *Visual design area with grid*
6. **Properties Panel** *Configure selected components*
7. **Connection Panel** *View/edit bus connections*
8. **Status Bar** *Validation messages and progress*
9. **Output Panel** *Generation logs and messages*

Setting Up a New Project

Creating Your First Bus Matrix Project

Step 1: Start New Project

- Click File → New Project (or Ctrl+N)
- Choose project template or start blank
- Set project name and location

Step 2: Choose Bus Type

- Select AXI4 (recommended for new designs)
- Or AXI3 for legacy compatibility
- Set global parameters (data width, address width)

Step 3: Configure Global Settings

- Data Width: 32, 64, 128, 256, 512 bits
- Address Width: 32 or 64 bits (typically 32)
- Clock/Reset naming conventions

Step 4: Save Project

- File → Save Project As...
- Choose .json format for compatibility
- Remember location for future use

Quick Tips:

- Start with AXI4 64-bit data width for most designs
- Use descriptive project names (e.g., 'cpu_gpu_ddr_system')
- Save frequently - GUI auto-saves every 5 minutes
- Templates are available in File→ Templates menu

Adding Masters to Your Design

Step-by-Step: Adding a Master Component

Step 1: Click "Add Master" Button

- Located in left panel or toolbar
- Alternative: Right-click canvas → Add Master

Step 2: Master Icon Appears on Canvas

- Default name: "Master_0", "Master_1", etc.
- Can be dragged to different position

Step 3: Configure Master Properties

- Double-click master icon OR select and use properties panel
- **Configuration Dialog Opens:**
  - Name: Descriptive name (e.g., 'CPU_0', 'DMA_Controller')
  - ID Width: 4-8 bits (affects outstanding transactions)
  - Priority: 0-15 (higher = higher priority in arbitration)
  - QoS Enable: Yes for latency-sensitive masters
  - Security: Secure/Non-secure for TrustZone systems

Step 4: Apply Configuration

- Click "Apply" or "OK" to save settings
- Master icon updates with new name

Common Master Types & Settings:

- CPU: ID=4-6 bits, Priority=High, QoS=Yes, Security=Secure
- DMA: ID=4-8 bits, Priority=Medium, QoS=Yes, Security=Match data
- GPU: ID=6-8 bits, Priority=High, QoS=Yes, Security=Non-secure
- Debug: ID=2-4 bits, Priority=Low, QoS=No, Security=Secure

Adding Slaves to Your Design

Step-by-Step: Adding a Slave Component

Step 1: Click "Add Slave" Button

- Located in right panel or toolbar
- Alternative: Right-click canvas → Add Slave

Step 2: Configure Slave Properties (CRITICAL)

- Slave configuration dialog appears automatically
- **MUST configure address mapping:**
  - Name: Descriptive name (e.g., 'DDR_Memory', 'UART_0')
  - Base Address: Starting address in hex (e.g., 0x00000000)
  - Size: Memory/peripheral size (e.g., 1GB, 4KB, 1MB)
  - Memory Type: Memory or Peripheral
  - Security: Secure/Non-secure access requirements
  - Read Latency: Response delay in clock cycles
  - Write Latency: Response delay in clock cycles

⚠ **IMPORTANT: No overlapping addresses allowed!**

GUI will show error if addresses overlap

Example Address Map:

```
DDR Memory:    Base=0x00000000, Size=1GB   (0x00000000-0x3FFFFFFF)
SRAM:          Base=0x40000000, Size=256MB (0x40000000-0x4FFFFFFF)
Peripherals:   Base=0x50000000, Size=256MB (0x50000000-0x5FFFFFFF)
Debug ROM:     Base=0xF0000000, Size=64KB  (0xF0000000-0xF000FFFF)
```

Quick Tips:

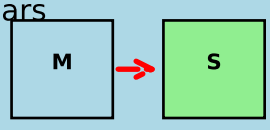
- Use power-of-2 sizes when possible
- Align base addresses to size boundaries
- Leave gaps for future expansion

Making Bus Connections

Connecting Masters to Slaves

Method 1: Drag and Drop (Recommended)

1. Click and hold on master output port (right side of master)
2. Drag mouse to slave input port (left side of slave)
3. Release mouse - connection line appears
4. Connection is automatically validated



Method 2: Connection Matrix

1. Click 'View' → 'Connection Matrix' (or Ctrl+M)
2. Matrix shows all masters (rows) vs slaves (columns)
3. Click checkboxes to enable/disable connections
4. Click 'Apply' to update visual design

Connection Rules & Validation:

✓ Each master can connect to multiple slaves
✓ Each slave can be accessed by multiple masters
✓ GUI prevents invalid connections automatically
✗ Cannot connect master to master
✗ Cannot connect slave to slave
⚠ Unconnected masters/slaves show warnings

Visual Connection Indicators:

- Solid Blue Line: Valid connection
- Dashed Red Line: Invalid/problematic connection
- Green Dots: Connection endpoints
- No line: Components not connected

Configuration Parameters

Global System Configuration

Access: Edit → Global Parameters

- Bus Type: AXI4, AXI3, AHB (affects generated RTL)
- Data Width: 32, 64, 128, 256, 512 bits (system-wide)
- Address Width: 32 or 64 bits (memory addressing range)
- Clock Domain: Single or multiple clock domains
- Reset Style: Synchronous or asynchronous reset

Interconnect Configuration

Access: Select interconnect → Properties panel

- Arbitration: Round-robin, Priority-based, QoS-aware
- Pipeline Stages: 0-3 stages (affects timing/area)
- Outstanding Transactions: Per-master limits
- Timeout Detection: Enable/disable transaction timeouts
- Error Handling: SLVERR, DECERR response generation

Advanced Features (Optional)

- QoS (Quality of Service): 4-bit priority levels
- Security: TrustZone secure/non-secure regions
- Region Support: 4-bit region identifiers
- User Signals: Custom user-defined signals
- Cache Hints: ARCACHE/AWCACHE signal generation

Configuration Tips:

- Start with defaults, modify only what you need
- Higher data widths = better performance but more area
- QoS only needed for mixed-criticality systems
- Pipeline stages help timing closure in large designs

RTL Generation Process

Generating RTL from Your Design

STEP 0: Pre-Generation Validation (REQUIRED)

- Click "Tools"→ "Validate Design" (or Ctrl+V)
- Fix any errors shown in red before proceeding

STEP 1: Access RTL Generation

- Click "Generate" → "Generate RTL" (or Ctrl+G)
- Alternative: Click RTL button in toolbar
- RTL Generation dialog opens

STEP 2: Configure RTL Options

- Output Directory: Choose where to save files
- File Naming: Prefix for generated modules
- Target: Synthesis (default) or Simulation
- Include Testbench: Generate basic testbench

STEP 3: Generate RTL Files

- Click "Generate" button
- Progress bar shows generation status
- Output panel shows generated files
- Success message appears when complete

STEP 4: Verify Output

- Check output directory for .v files
- Review generation log for warnings
- Optionally run syntax check
- Files ready for synthesis/simulation

Understanding RTL Output Files

Generated RTL File Structure

Generated Files (Example for 2 masters, 3 slaves):

output_rtl/

```
├── axi4_interconnect_m2s3.v      # Top-level interconnect
├── axi4_address_decoder.v        # Address decoding logic
├── axi4_arbiter.v               # Arbitration logic
├── axi4_router.v                # Transaction routing
├── tb_axi4_interconnect.v       # Basic testbench
└── run_synthesis.tcl            # Synthesis script
```

File Descriptions:

- axi4_interconnect_m2s3.v: Main module to instantiate in your design
- axi4_address_decoder.v: Decodes addresses to route to correct slave
- axi4_arbiter.v: Handles multiple masters accessing same slave
- axi4_router.v: Routes transactions between masters and slaves
- tb_axi4_interconnect.v: Basic testbench for functional verification

How to Use Generated RTL:

1. Copy generated .v files to your project directory
2. Add files to your synthesis/simulation tool
3. Instantiate axi4_interconnect_m2s3 in your top module:

```
axi4_interconnect_m2s3 u_interconnect (
    .clk(clk),
    .rst_n(rst_n),
    // Master 0 interface
    .m0_axi_awvalid(cpu_awvalid),
    .m0_axi_awready(cpu_awready),
    // ... other signals
);
```

4. Connect your masters and slaves to the interface ports

RTL Integration Steps

Integrating Generated RTL into Your Design

Step 1: Prepare Your Environment

- • Create project in your synthesis tool (Vivado, Quartus, etc.)
- • Set up directory structure for RTL files
- • Ensure proper tool versions and settings

Step 2: Add Generated Files

- • Copy all .v files from output directory
- • Add files to project (maintaining hierarchy)
- • Set axi4_interconnect_m2s3.v as top-level if standalone

Step 3: Create Wrapper Module

- • Create your system top-level module
- • Instantiate interconnect with proper port connections
- • Add clock/reset generation logic

Step 4: Connect Masters and Slaves

- • Connect CPU, DMA, GPU to master ports
- • Connect memories, peripherals to slave ports
- • Ensure signal widths match interconnect interface

Step 5: Verify and Synthesize

- • Run syntax check on all files
- • Simulate basic functionality
- • Run synthesis and check for errors
- • Analyze resource utilization and timing

Common Integration Issues:

- Port width mismatch: Check master/slave interface widths
- Clock domain crossing: Add proper synchronizers
- Reset timing: Ensure reset is properly distributed
- Synthesis warnings: Review and fix constraint violations

VIP Generation Process

Generating UVM VIP from Your Design

What is VIP (Verification Intellectual Property)?

- Complete UVM-based verification environment
- Includes agents, sequences, tests, and scoreboards

STEP 1: Access VIP Generation

- Ensure RTL is generated first (VIP needs RTL files)
- Click "Generate" → "Generate VIP" (or Ctrl+Shift+G)
- VIP Generation dialog opens

STEP 2: Configure VIP Options

- Output Directory: Choose VIP output location
- Test Suite: Basic, Comprehensive, or Custom
- Simulator: VCS, Questa, Xcelium, or Generic
- Coverage: Enable functional and code coverage

STEP 3: Generate VIP Environment

- Click "Generate VIP" button
- Progress shows: Agents → Sequences → Tests→ Scripts
- Generation log shows detailed progress
- Success message when complete

STEP 4: Verify VIP Output

- Check vip_output/ directory structure
- Review generated test list
- Compilation scripts created for target simulator
- VIP ready for simulation

Understanding VIP Output

Generated VIP Directory Structure

VIP Output Structure:

```
vip_output/
├── env/                    # UVM Environment
│   ├── axi4_env.sv         # Top environment class
│   ├── axi4_scoreboard.sv  # Transaction checking
│   └── axi4_coverage.sv    # Functional coverage
├── agents/                 # Master/Slave agents
│   ├── master_agent/       # Master BFM and driver
│   └── slave_agent/        # Slave BFM and driver
├── sequences/              # Test sequences
│   ├── basic_sequences.sv  # Basic read/write
│   ├── burst_sequences.sv  # Burst transfers
│   └── error_sequences.sv  # Error injection
├── tests/                  # UVM tests
│   ├── base_test.sv        # Base test class
│   ├── basic_test.sv       # Basic functionality
│   └── stress_test.sv      # Stress testing
├── rtl_wrapper/            # RTL integration
│   └── dut_wrapper.sv      # DUT instantiation
└── sim/                    # Simulation scripts
    ├── Makefile            # Build automation
    └── run_sim.sh          # Simulation runner
```

Key VIP Components:

- Environment (env/): Top-level UVM environment coordinating all agents
- Agents (agents/): Master and slave BFMs with drivers and monitors
- Sequences (sequences/): Reusable transaction sequences for testing
- Tests (tests/): Complete test scenarios with pass/fail criteria
- RTL Wrapper: Integration layer connecting VIP to generated RTL
- Simulation Scripts: Automated compilation and execution scripts

How to Use Generated VIP:

1. Navigate to vip_output/sim/ directory
2. Run: make compile (compiles all VIP and RTL)
3. Run: make sim TEST=basic_test (runs specific test)
4. View results in logs/ directory
5. Open waveforms for debugging: verdi -ssf waves.fsdb

Running VIP Simulations

Step-by-Step VIP Simulation Guide

Step 1: Setup Environment

```
cd vip_output/sim
source /path/to/simulator/setup.sh  # VCS, Questa, etc.
export UVM_HOME=/path/to/uvm/library
```

Step 2: Compile VIP and RTL

```
make compile                        # Compile everything
# OR compile manually:
vcs -sverilog -ntb_opts uvm -f compile.f
```

Step 3: Run Basic Test

```
make sim TEST=basic_test            # Run basic test
# OR run manually:
./simv +UVM_TESTNAME=basic_test +UVM_VERBOSITY=LOW
```

Step 4: View Results

```
cat logs/basic_test.log             # View test log
verdi -ssf waves/basic_test.fsdb # View waveforms
firefox coverage_report/index.html  # View coverage
```

Available Test Suite:

- basic_test: Simple read/write operations
- burst_test: Various burst types and sizes
- stress_test: High-traffic scenarios
- error_test: Error injection and recovery
- coverage_test: Maximum coverage collection

Debugging Tips:

- Use +UVM_VERBOSITY=HIGH for detailed logs
- Enable waveform dumping: +fsdb+all

Complete Workflow: CPU + GPU + DDR System

Real Example: Building a CPU+GPU+DDR System

System Goal: 2 Masters (CPU, GPU) → 3 Slaves (DDR, SRAM, Peripherals)

1. Launch GUI: ./launch_gui.sh
2. New Project: File → New Project → 'cpu_gpu_system'
3. Add CPU Master:
- Click 'Add Master' → Name: 'CPU_0'
- ID Width: 4, Priority: 2, QoS: Yes, Security: Secure

4. Add GPU Master:
- Click 'Add Master' → Name: 'GPU_0'
- ID Width: 6, Priority: 1, QoS: Yes, Security: Non-secure

5. Add DDR Slave:
- Click 'Add Slave' → Name: 'DDR_Memory'
- Base: 0x00000000, Size: 2GB, Type: Memory

6. Add SRAM Slave:
- Click 'Add Slave' → Name: 'SRAM_Cache'
- Base: 0x80000000, Size: 256MB, Type: Memory

7. Add Peripheral Slave:
- Click 'Add Slave' → Name: 'Peripherals'
- Base: 0xA0000000, Size: 256MB, Type: Peripheral

8. Make Connections:
- Drag CPU_0 → DDR_Memory, SRAM_Cache, Peripherals
- Drag GPU_0 → DDR_Memory, SRAM_Cache (no peripherals)

9. Validate: Tools→ Validate Design (fix any errors)
10. Generate RTL: Generate → Generate RTL → output_rtl/
11. Generate VIP: Generate → Generate VIP → vip_output/
12. Test VIP: cd vip_output/sim && make sim TEST=basic_test

Expected Results:

✓ RTL: axi4_interconnect_m2s3.v + support files (~5 files)
✓ VIP: Complete UVM environment (~50+ files)
✓ Simulation: All tests pass with 100% functional coverage
✓ Ready for: Synthesis, FPGA implementation, ASIC flow

Advanced GUI Features

Project Templates

Pre-configured system templates

Usage: File → New From Template → Select template

Batch Generation

Generate multiple configurations

Usage: Tools → Batch Generation → Configure variants

Address Map Viewer

Visual address space layout

Usage: View → Address Map → Interactive viewer

Performance Analysis

Bandwidth and latency analysis

Usage: Tools → Performance Analysis → Run analysis

Design Rule Check

Comprehensive design validation

Usage: Tools → Design Rule Check → Full validation

Export/Import

Configuration file management

Usage: File → Export/Import → Various formats

Keyboard Shortcuts:

```
Ctrl+N: New Project          Ctrl+G: Generate RTL
Ctrl+O: Open Project         Ctrl+Shift+G: Generate VIP
Ctrl+S: Save Project         Ctrl+V: Validate Design
Ctrl+M: Connection Matrix    F5: Refresh Canvas
Delete: Remove Selected      Ctrl+Z: Undo
```

GUI Troubleshooting

Problem: GUI won't launch

Symptoms:
- Command not found
- Permission denied

Solutions:
- chmod +x launch_gui.sh
- sudo apt-get install python3-tk
- Check Python 3.6+ is installed

Problem: Can't add masters/slaves

Symptoms:
- Buttons greyed out
- No response to clicks

Solutions:
- Create new project first
- Check project is not read-only
- Restart GUI and try again

Problem: Address overlap errors

Symptoms:
- Red error messages
- Can't generate RTL

Solutions:
- Check slave address ranges
- Use Address Map viewer
- Adjust base addresses or sizes

Problem: RTL generation fails

Symptoms:
- Generation stops
- Error in output panel

Solutions:
- Run validation first (Ctrl+V)
- Fix all validation errors
- Check output directory permissions

Problem: VIP generation fails

Symptoms:
- Missing VIP files
- Compilation errors

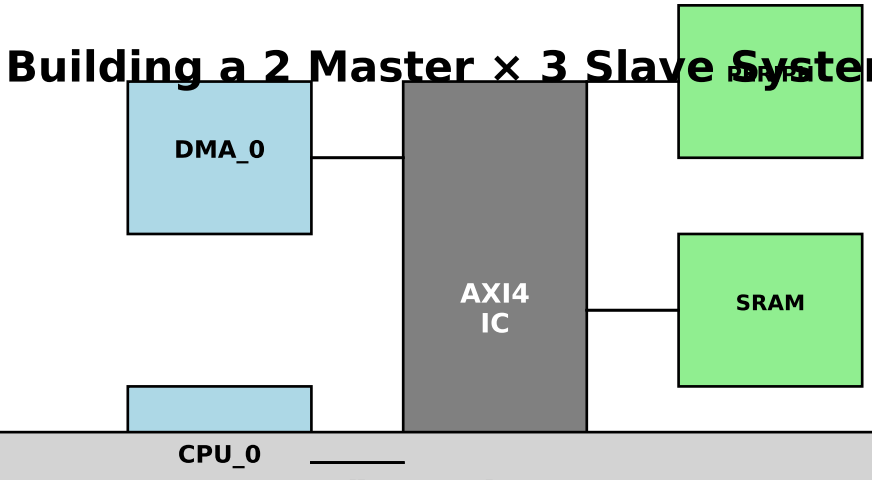
Solutions:
- Generate RTL first
- Check RTL files exist
- Verify simulator setup

Getting Additional Help:

- Check Help → User Manual for detailed documentation
- Enable debug logging: Help → Enable Debug Logging

Example: 2×3 System Design

Building a 2 Master × 3 Slave System



Component Configuration:

Masters:

```
CPU_0:  ID=4, Priority=2, QoS=Yes, Security=Secure
DMA_0:  ID=6, Priority=1, QoS=Yes, Security=Match
```

Slaves:

```
DDR:    Base=0x00000000, Size=1GB,   Type=Memory
SRAM:   Base=0x40000000, Size=256MB, Type=Memory
PERIPH: Base=0x50000000, Size=256MB, Type=Peripheral
```

GUI Steps to Build This System:

1. New Project → Name: '2x3_system'
2. Add Master → Name: 'CPU_0', configure as above
3. Add Master → Name: 'DMA_0', configure as above
4. Add Slave → Name: 'DDR', Base: 0x00000000, Size: 1GB
5. Add Slave → Name: 'SRAM', Base: 0x40000000, Size: 256MB
6. Add Slave → Name: 'PERIPH', Base: 0x50000000, Size: 256MB
7. Connect: CPU_0 → All slaves, DMA_0 → DDR+SRAM only
8. Validate → Generate RTL → Generate VIP
9. Result: axi4_interconnect_m2s3.v + complete VIP

Example: Complex Multi-Master System

Advanced: 4 Master × 6 Slave System

System: Multi-core CPU + GPU + AI Accelerator + Debug

Use Case: High-performance SoC with AI capabilities

Features: QoS, Security zones, Performance monitoring

Masters:

- CPU_Cluster: 4-bit ID, High priority, Secure
- GPU_Engine: 6-bit ID, High priority, Non-secure
- AI_Accel: 8-bit ID, Medium priority, Non-secure
- Debug_Port: 2-bit ID, Low priority, Secure

Slaves:

- DDR_0: 0x00000000, 2GB, Main memory
- DDR_1: 0x80000000, 2GB, Graphics memory
- SRAM: 0x40000000, 512MB, Fast cache
- ROM: 0xF0000000, 64MB, Boot code
- AI_MEM: 0x60000000, 1GB, AI model storage
- PERIPH: 0x50000000, 256MB, System peripherals

Advanced Configuration Settings:

- QoS Arbitration: Weighted round-robin with latency bounds
- Security: TrustZone with secure/non-secure partitioning
- Performance: Outstanding transaction limits per master
- Monitoring: Transaction counters and bandwidth meters
- Error Handling: Comprehensive timeout and error injection

Build Considerations:

- Design time: ~30 minutes in GUI vs hours manually
- Generated RTL: ~15 files, ~5000 lines of optimized Verilog
- VIP complexity: 80+ files, comprehensive test suite
- Validation: 20+ test scenarios, full coverage analysis
- Integration: Ready for synthesis and FPGA implementation

Common Design Patterns

CPU + Memory Pattern

1 Master → 1-2 Slaves (DDR + optional cache)

Use Case: Simple microcontroller systems

`Config: Master: CPU, ID=4 | Slaves: DDR, SRAM`

Multi-Core Pattern

2-4 Masters → Shared memory hierarchy

Use Case: Multi-core processors, parallel computing

`Config: Masters: CPU0-3, ID=4-6 | Slaves: L3, DDR, Peripherals`

SoC Multimedia Pattern

CPU + GPU + Specialized accelerators

Use Case: Graphics, video processing, mobile SoCs

`Config: Masters: CPU, GPU, DSP | Slaves: DDR, SRAM, Peripherals`

AI/ML Accelerator Pattern

CPU + AI engine + High-bandwidth memory

Use Case: Machine learning inference/training

`Config: Masters: CPU, AI, DMA | Slaves: HBM, DDR, Model cache`

IoT Edge Pattern

Low-power CPU + Sensor interfaces

Use Case: IoT devices, sensor networks

`Config: Master: MCU | Slaves: Flash, SRAM, Peripheral hub`

Design Guidelines:

- Start simple, add complexity gradually
- Use templates for common patterns