# ECE 699: Lecture 6

# AXI Interfacing

# Using DMA & AXI4-Stream

# Required Reading

## The ZYNQ Book

- *Chapter 19: AXI Interfacing*
- *Chapter 2.3: Processing System – Programmable Logic Interfaces*
- *Chapter 9.3: Buses*
- *Chapter 10.1: Interfacing and Signals*

# Required Reading

**ARM AMBA AXI Protocol v1.0: Specification**

- ***Chapter 1: Introduction***
- ***Chapter 2: Signal Descriptions***
- ***Chapter 3: Channel Handshake***
- ***Chapter 4: Addressing Options***
- ***Chapter 9: Data Buses***

# Recommended Reading

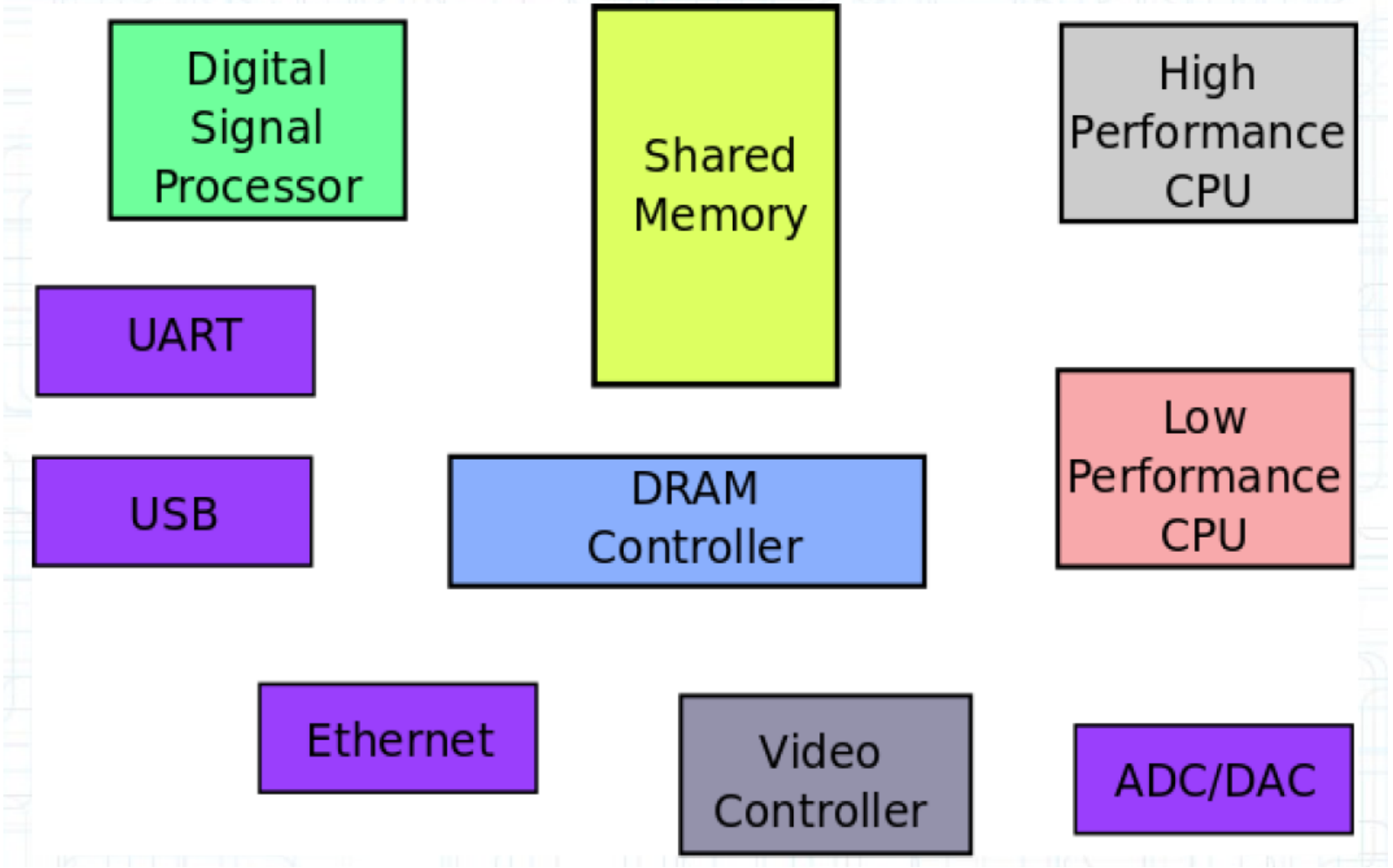
**P. Schaumont, A Practical Introduction to Hardware/Software Codesign, 2nd Ed.**

- ***Chapter 10: On-Chip Buses***

**M.S. Sadri, ZYNQ Training (presentations and videos)**

- ***Lesson 1 : What is AXI?***
- ***Lesson 2 : What is an AXI Interconnect?***
- ***Lesson 3 : AXI Stream Interface***

# Components of Today's Systems-on-Chip



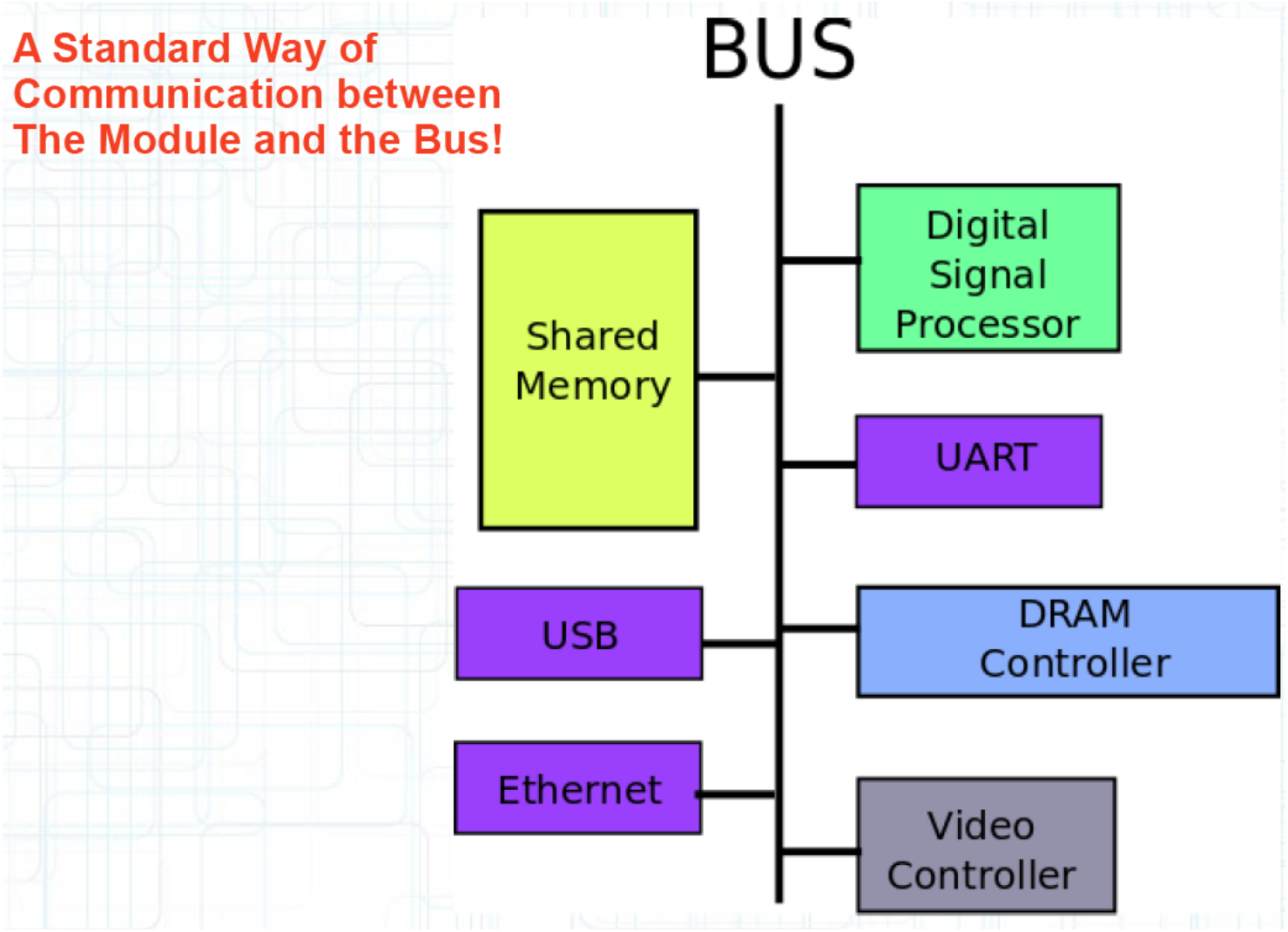

Source: M.S. Sadri, Zynq Training

# Connectivity Requirements

- A standard
  - All units talk based on that standard
  - All units can talk easily to each other
- Maintenance
  - Design is easily maintained/updated, debugged
- Re-use
  - Units can be easily re-used in other desigs

Source: M.S. Sadri, Zynq Training

# SoC Buses




Source: M.S. Sadri, Zynq Training

# Solution Adopted in ZYNQ

**Advanced Microcontroller Bus Architecture (AMBA):**
an open-standard, on-chip interconnect specification for the connection and management of functional blocks in system-on-a-chip (SoC) designs.
First version introduced by ARM in 1996.

**AMBA Advanced eXtensible Interface 4 (AXI4):**
the fourth generation of AMBA interface defined in the AMBA 4 specification, targeted at high performance, high clock frequency systems.
Introduced by ARM in 2010.

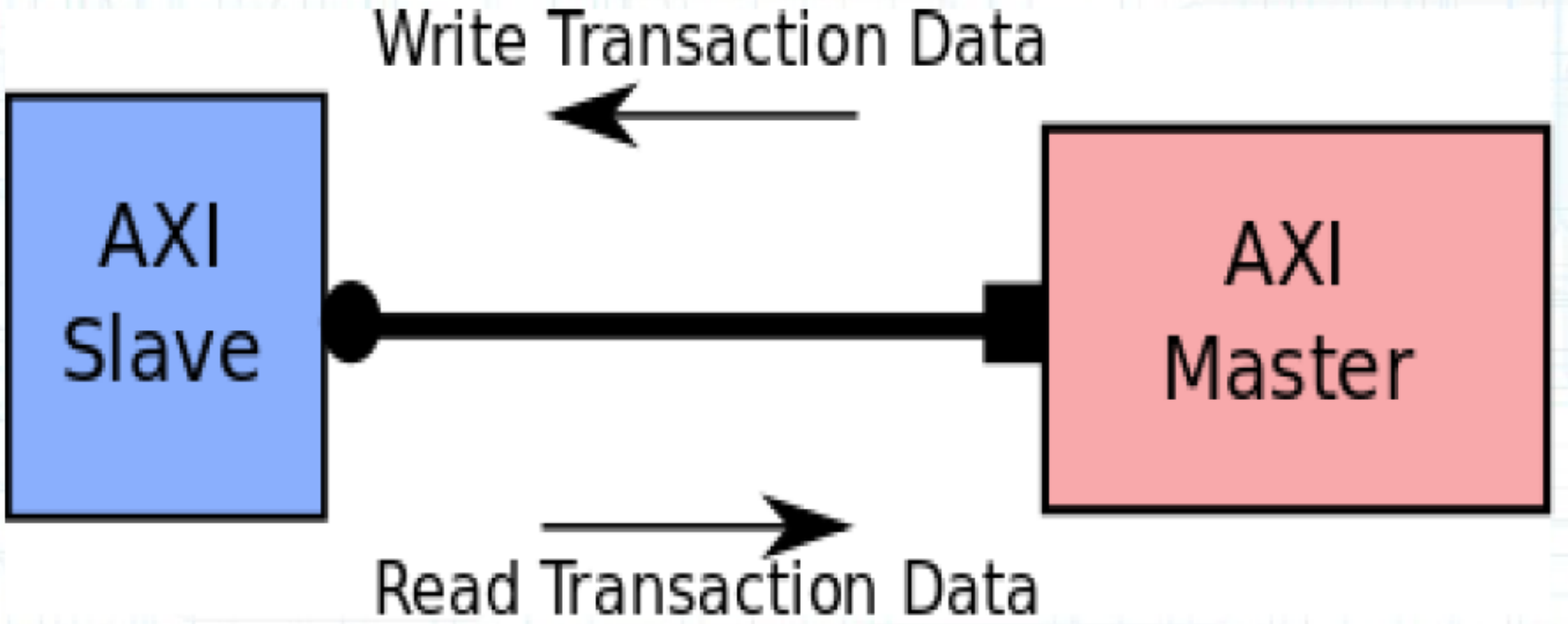
Source: M.S. Sadri, Zynq Training

# Basic Concepts

- Transaction :
  - Transfer of data from one point in the hardware to another point
- Master : Initiates the transaction
- Slave : Responds to the initiated transaction

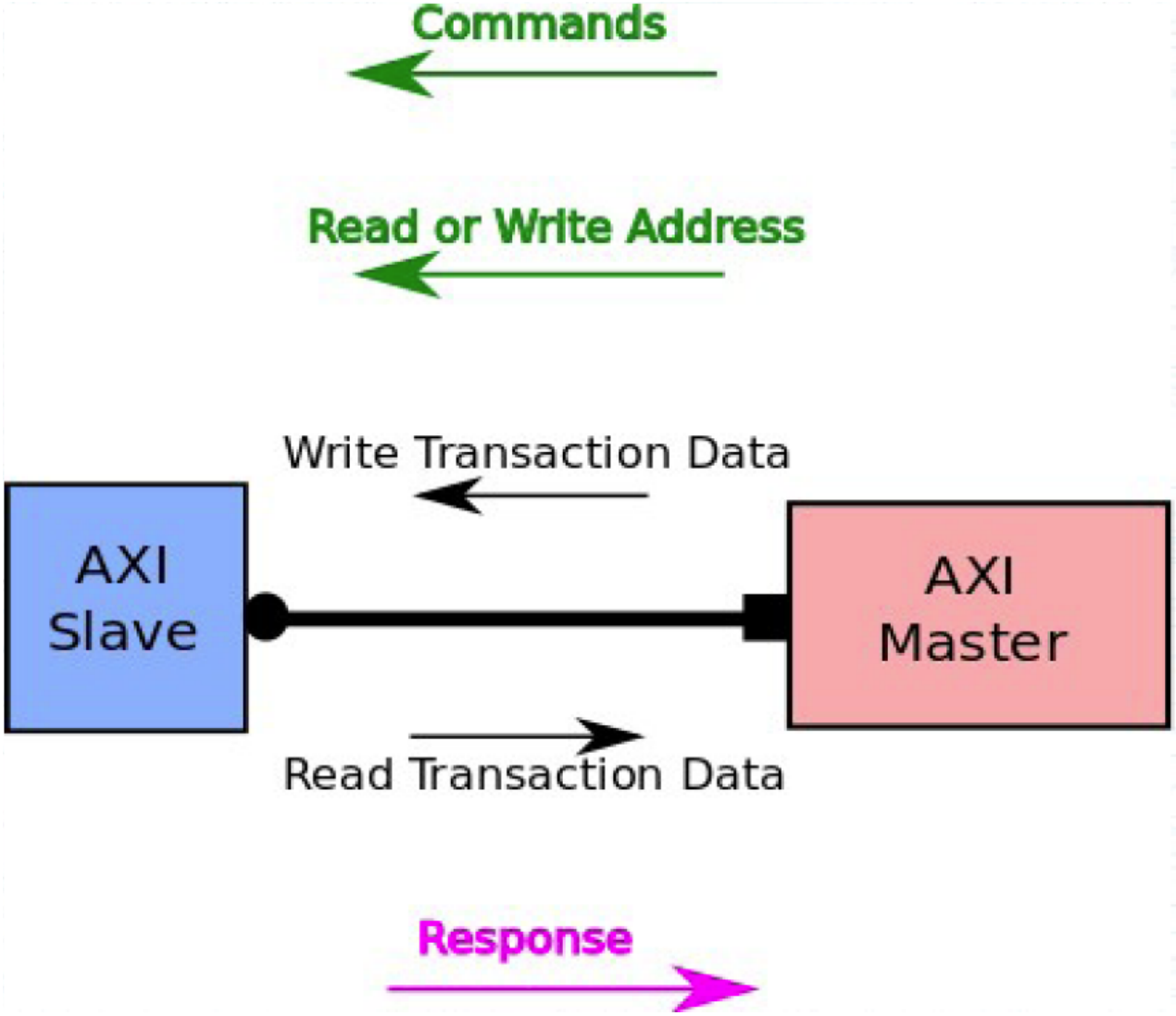
Source: M.S. Sadri, Zynq Training

# Communication Between AXI Master and AXI Slave



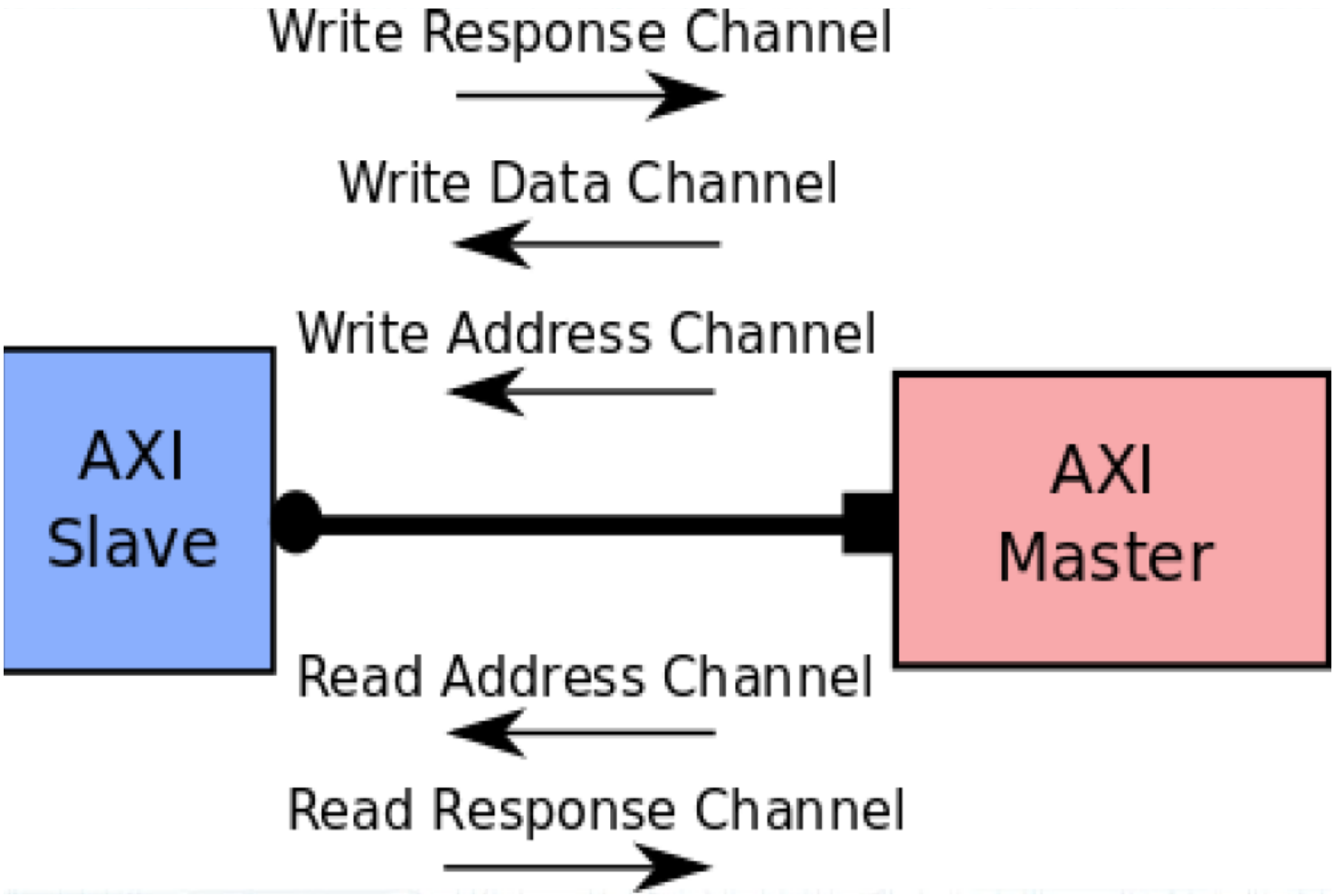

Source: M.S. Sadri, Zynq Training

# Additional Information Exchanged Between AXI Master and AXI Slave



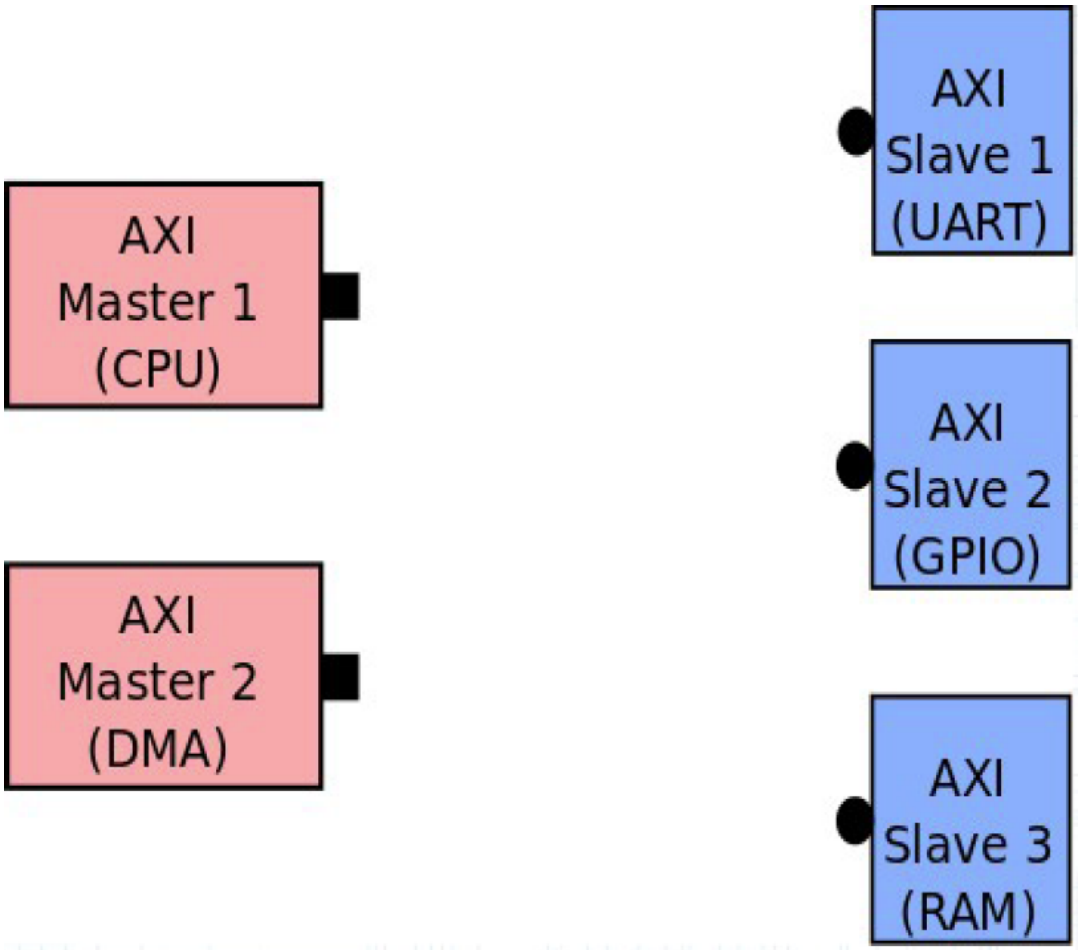

Source: M.S. Sadri, Zynq Training

# Five Channels of AXI Interface



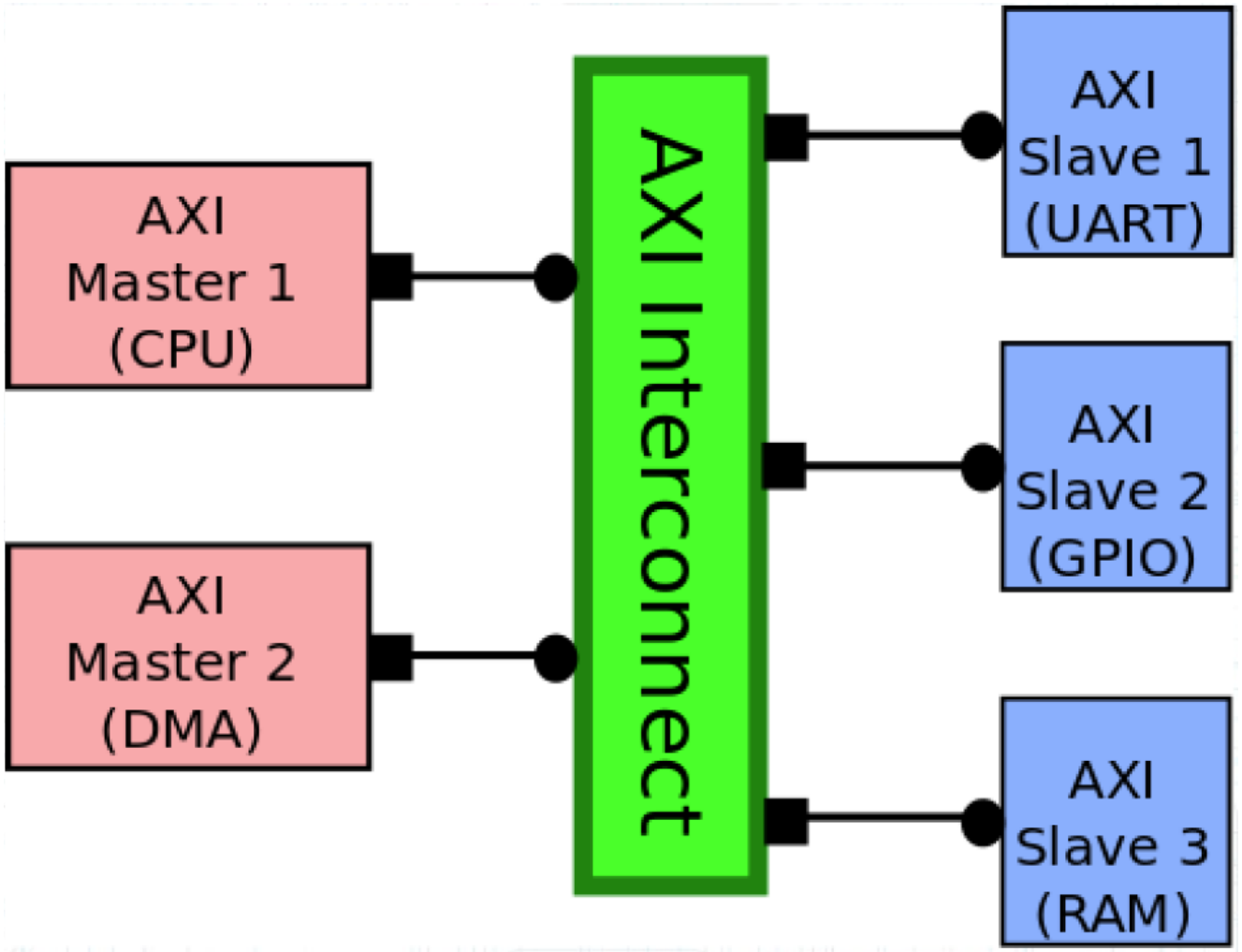

Source: M.S. Sadri, Zynq Training

# Connecting Masters and Slaves

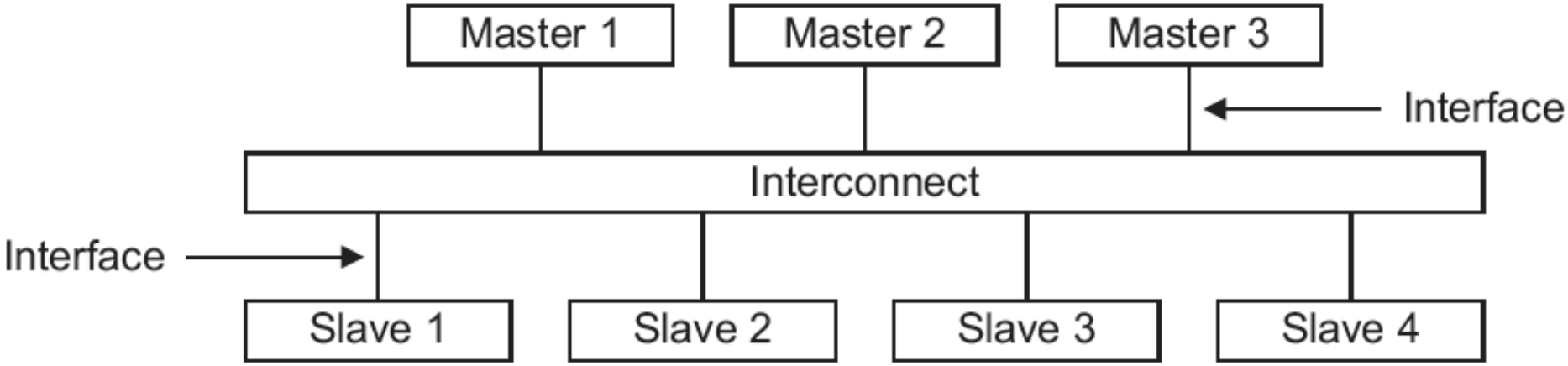



Source: M.S. Sadri, Zynq Training

# AXI Interconnect



# Interconnect vs. Interface



# AXI Interfaces and Interconnects

**Interface**

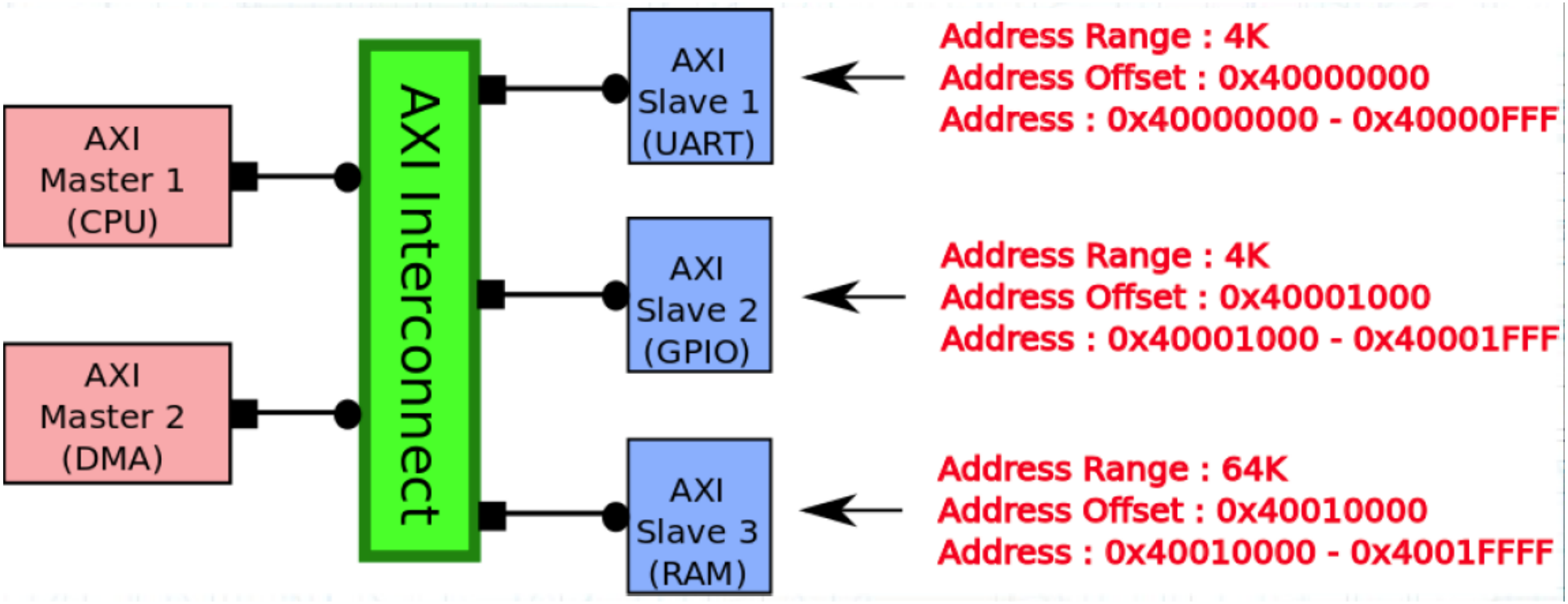
A point-to-point connection for passing data, addresses, and hand-shaking signals between master and slave clients within the system

**Interconnect**

A switch which manages and directs traffic between attached AXI interfaces

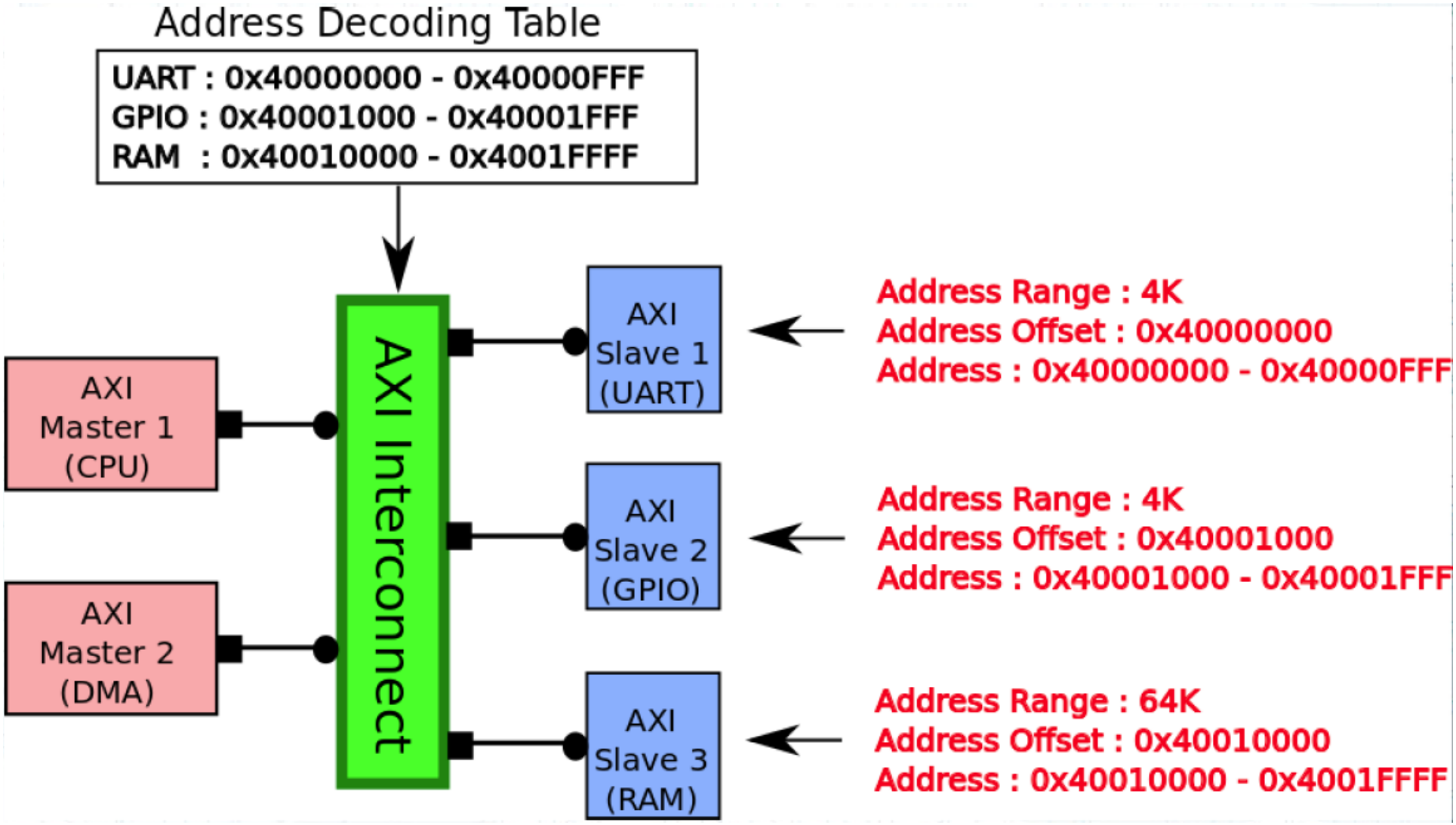
Source: The Zynq Book

# Addressing of Slaves



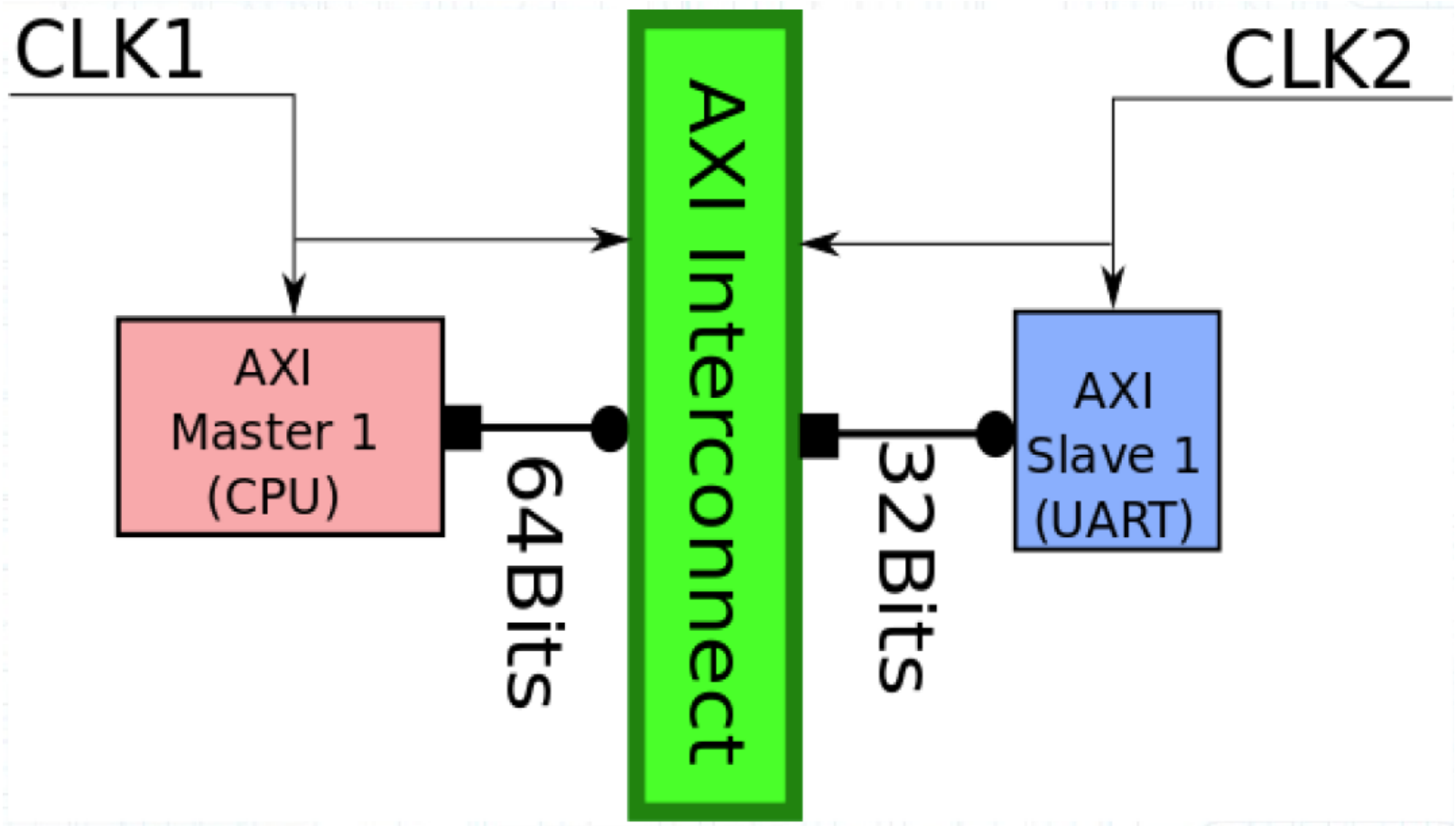

Source: M.S. Sadri, Zynq Training

# AXI Interconnect Address Decoding



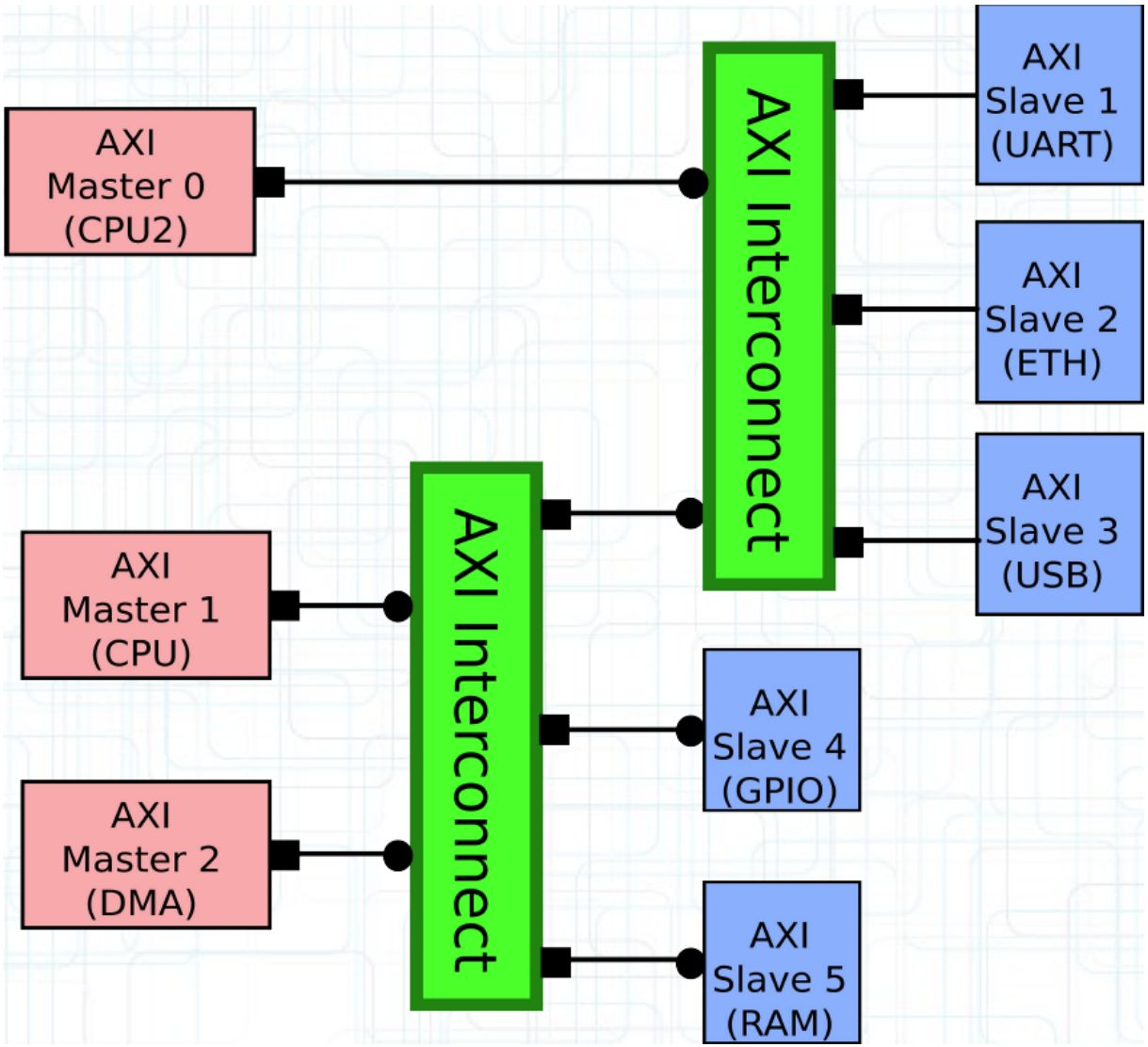

Source: M.S. Sadri, Zynq Training

# Clock Domain and Width Conversion



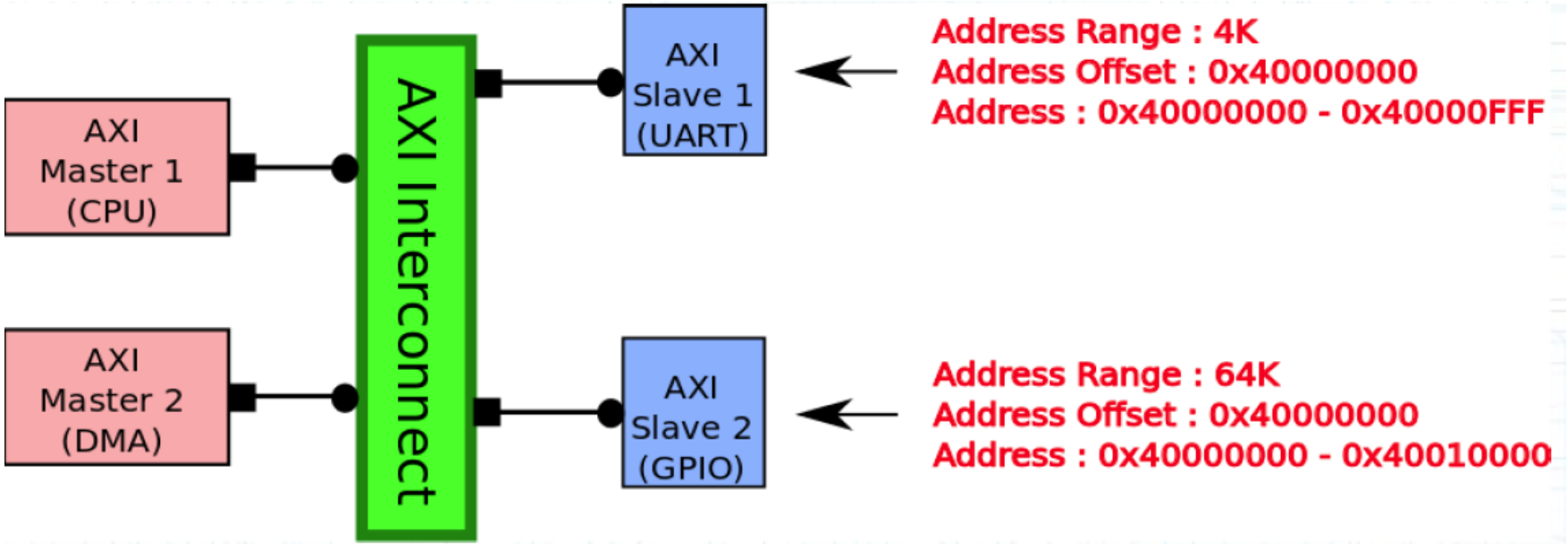

Source: M.S. Sadri, Zynq Training

# Hierarchical AXI Interconnects



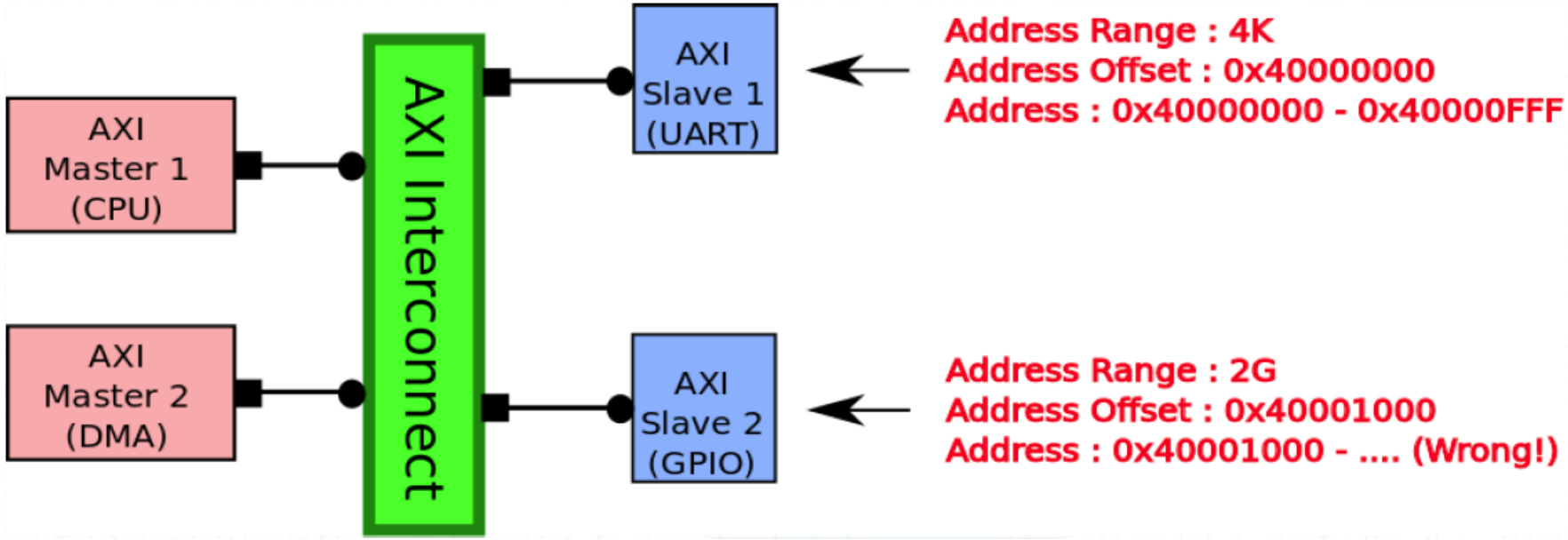

Source: M.S. Sadri, Zynq Training

# Simple Address Definition Rules
# No Overlaps

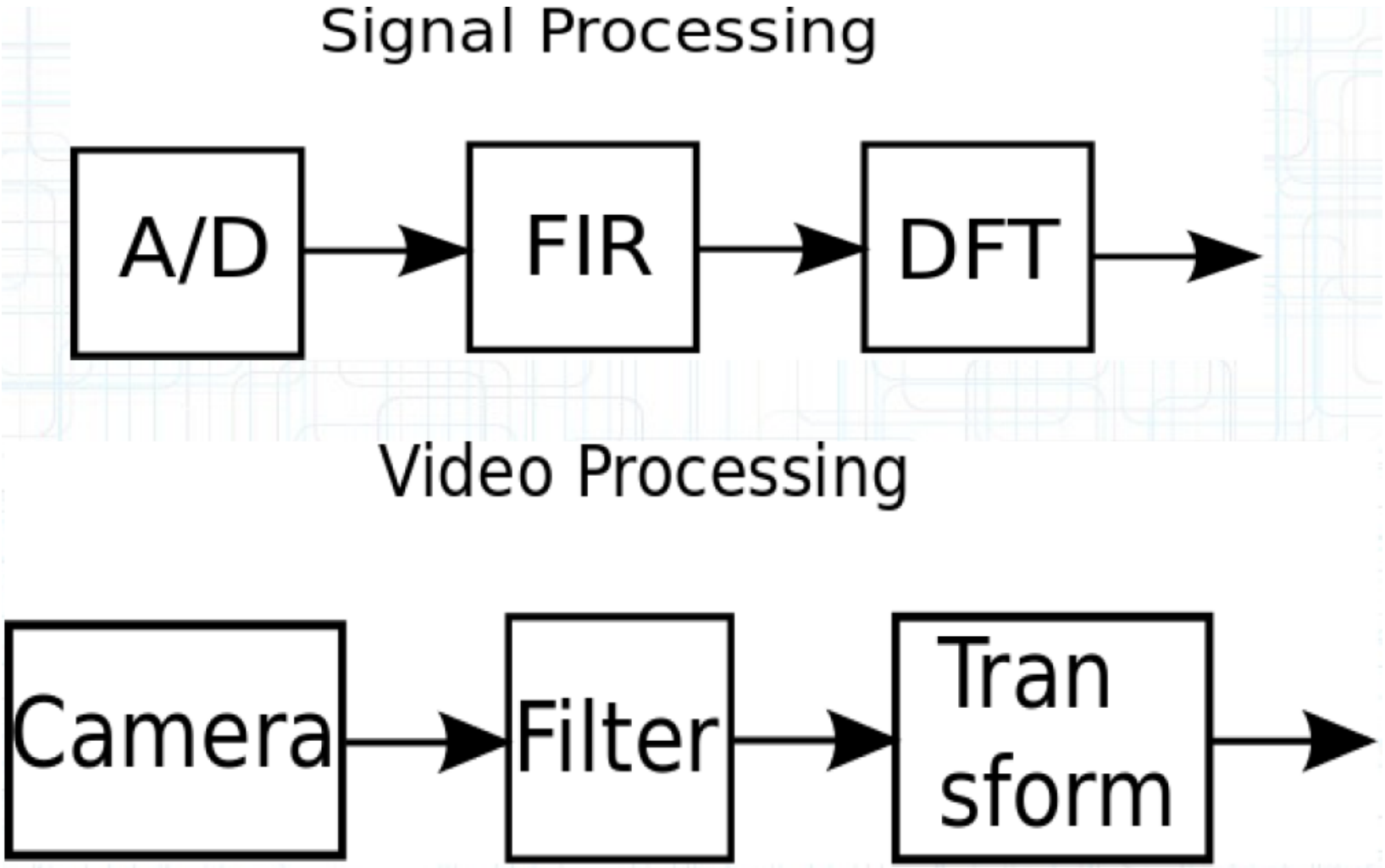



Source: M.S. Sadri, Zynq Training

# Simple Address Definition Rules
## Address Alignment

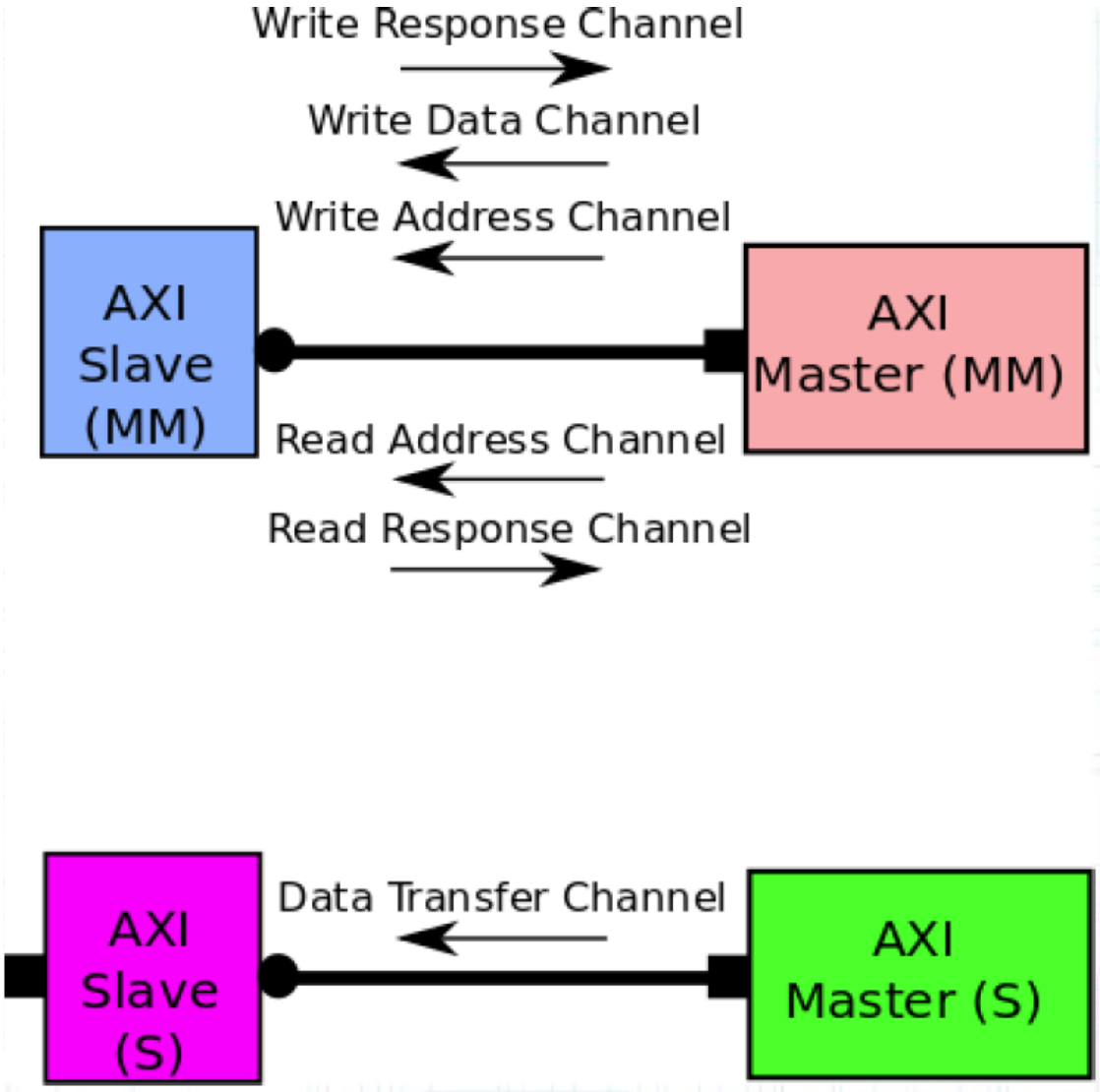


# Point-to-Point Data Flows



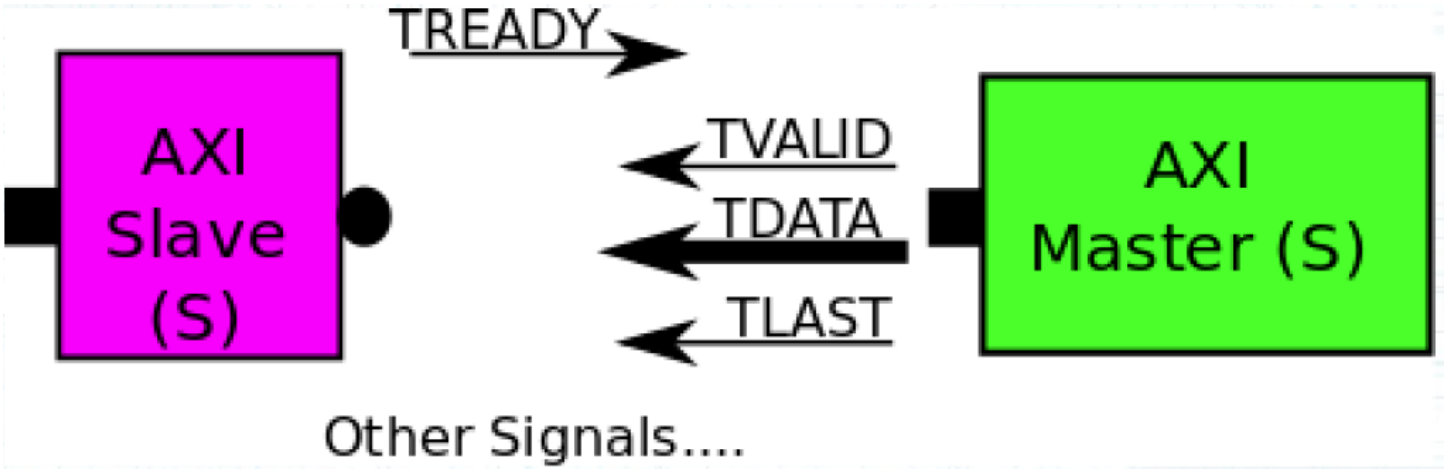

Source: M.S. Sadri, Zynq Training

# AXI Memory-Mapped vs. AXI Stream




Source: M.S. Sadri, Zynq Training

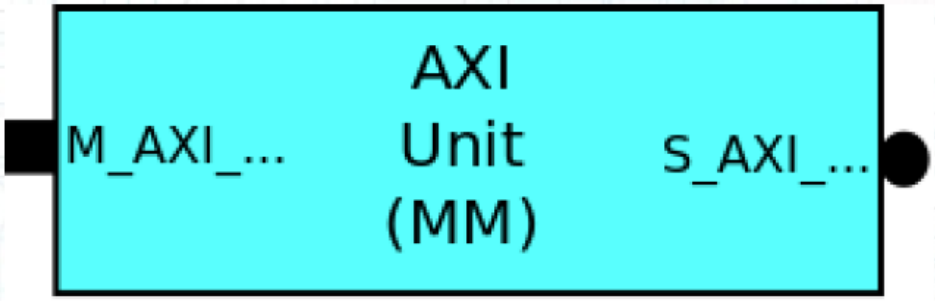
# Selected AXI Stream Ports



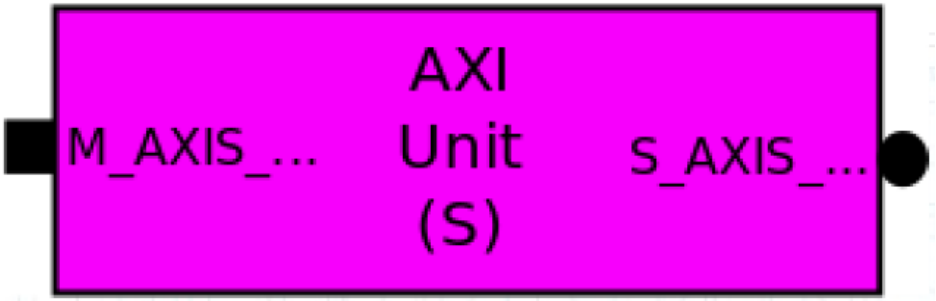

Source: M.S. Sadri, Zynq Training

# AXI Port Naming Conventions

- Memory Mapped



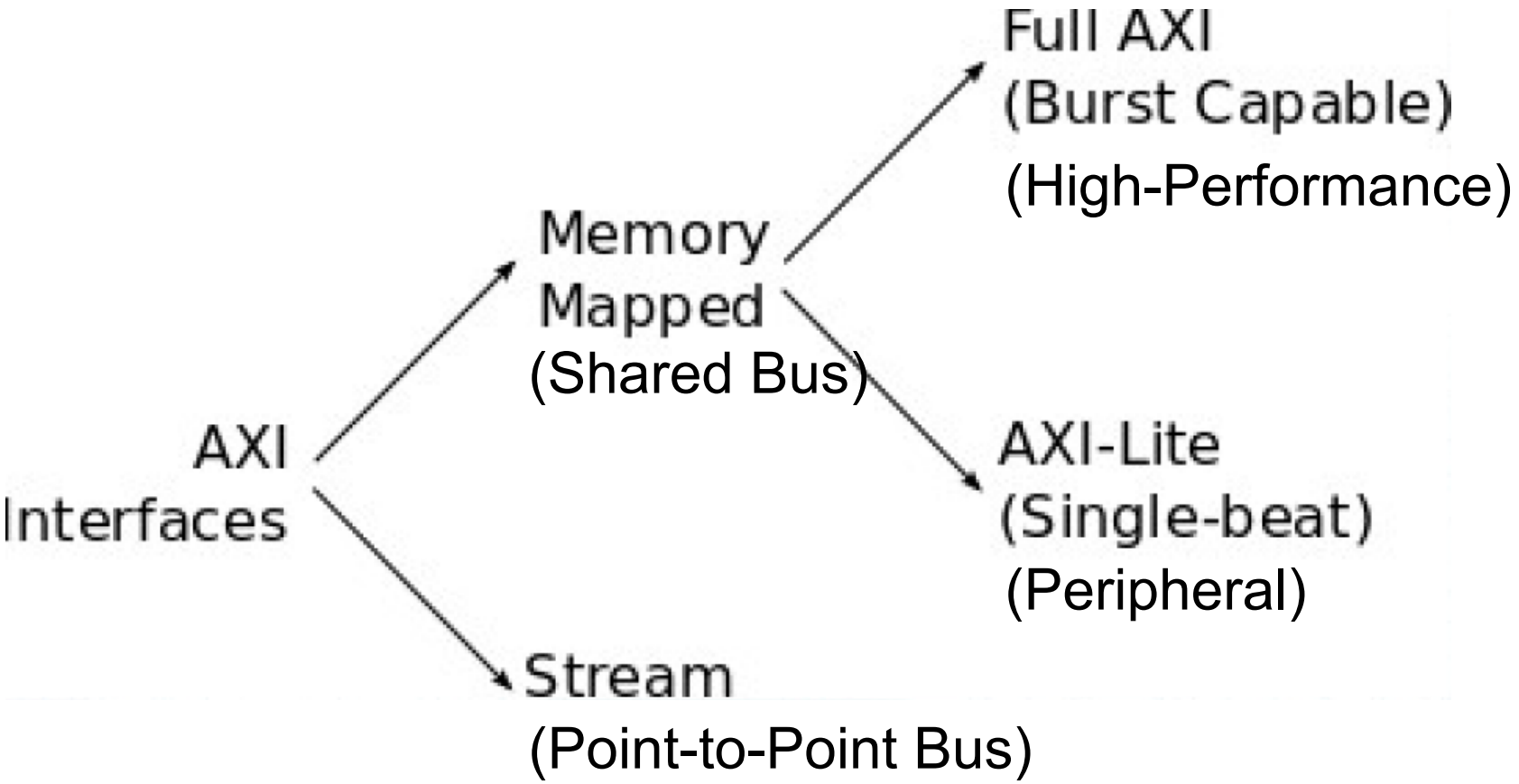

- Stream




- Signal names:
  - e.g. S_AXI_awid, M_AXI_arready ….

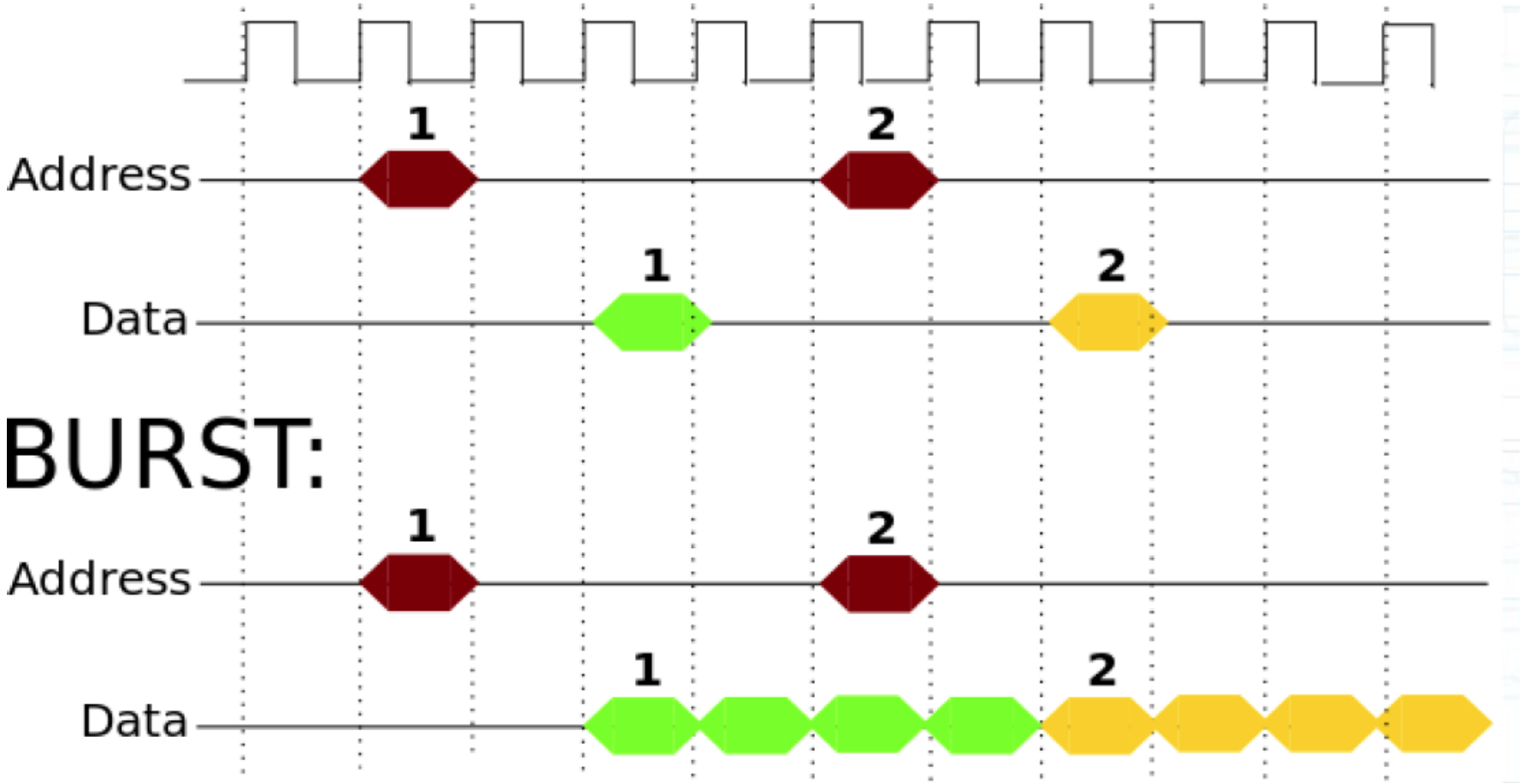
Source: M.S. Sadri, Zynq Training

# AXI Interfaces




Source: M.S. Sadri, Zynq Training

# Concept of a Burst




Source: M.S. Sadri, Zynq Training

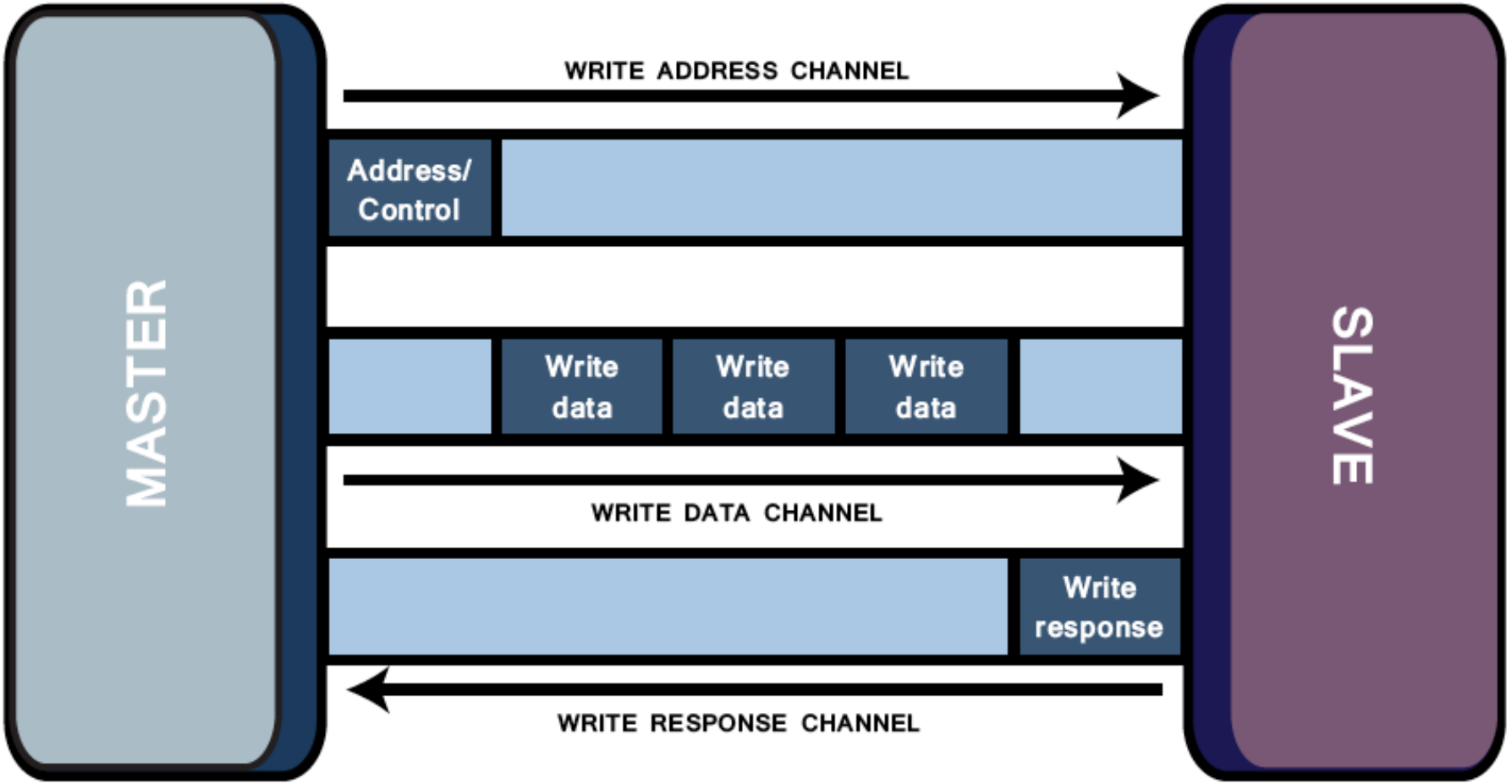
# Competing System-on-Chip Bus Standards

| Bus | Developed by | High-Performance Shared Bus | Peripheral Shared Bus | Point-to-Point Bus |
|---|---|---|---|---|
| AMBA v3 | ARM | AHB | APB | |
| AMBA v4 | ARM | AXI4 | AXI4-Lite | AXI4-Stream |
| Coreconnect | IBM | PLB | OPB | |
| Wishbone | SiliCore Corp. | Crossbar Topology | Shared Topology | Point to Point Topology |
| Avalon | Altera | Avalon-MM | Avalon-MM | Avalon-ST |

AMBA: Advanced Microcontroller Bus Architecture
AXI: Advanced eXtensible Interface
AHB: AMBA High-speed Bus
APB: AMBA Peripheral Bus
PLB: Processor Local Bus
OPB: On-chip Peripheral Bus
MM: Memory Mapped
ST: Streaming

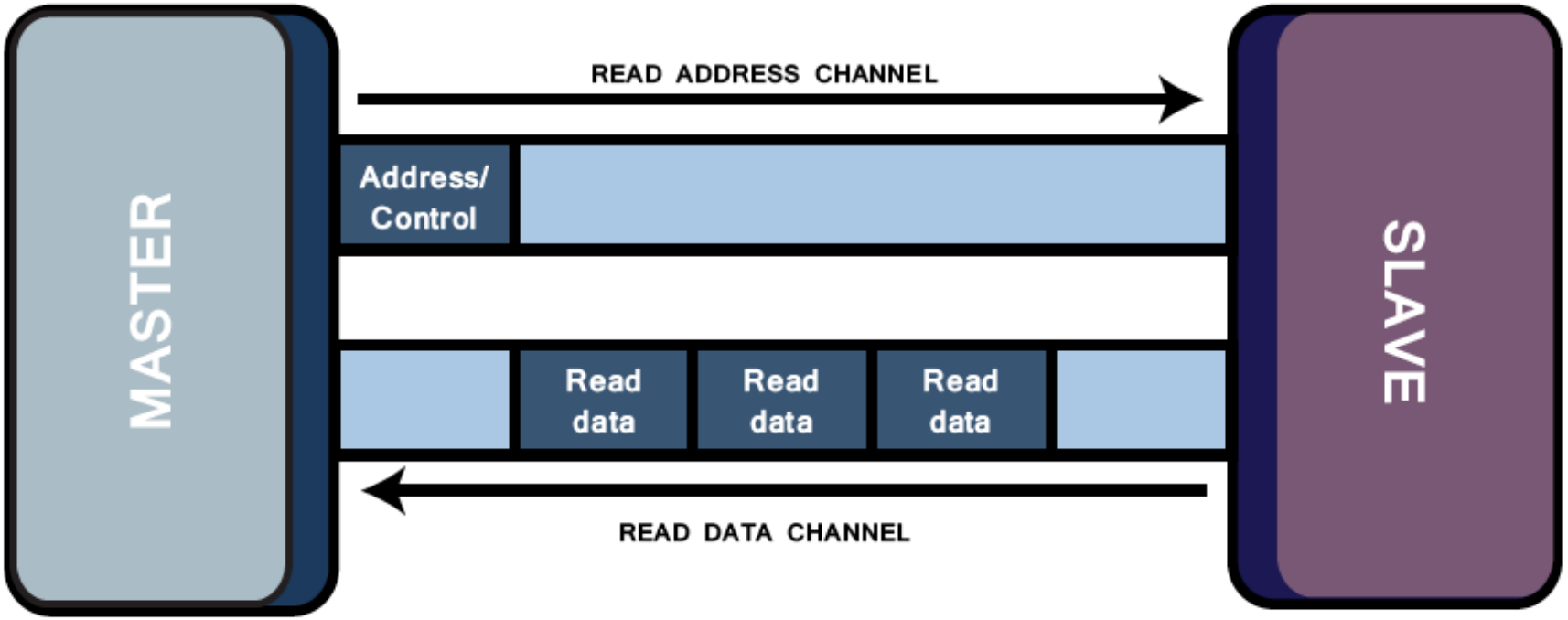
Source: A Practical Introduction to Hardware/Software Codesign

# AXI4 Write




Source: The Zynq Book

# AXI4 Read




Source: The Zynq Book

# AXI4 Interface

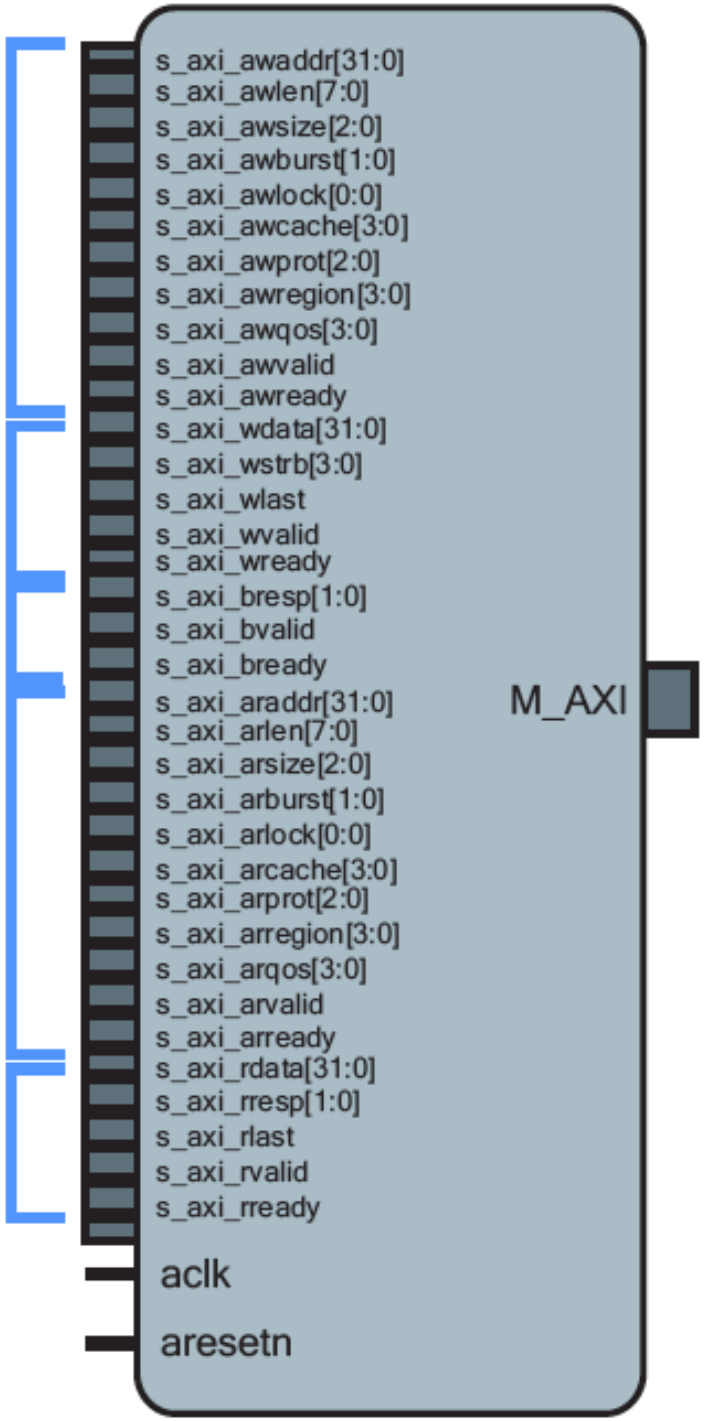
Write Address Channel

Write Data Channel

Write Response Channel

Read Address Channel

Read Data Channel



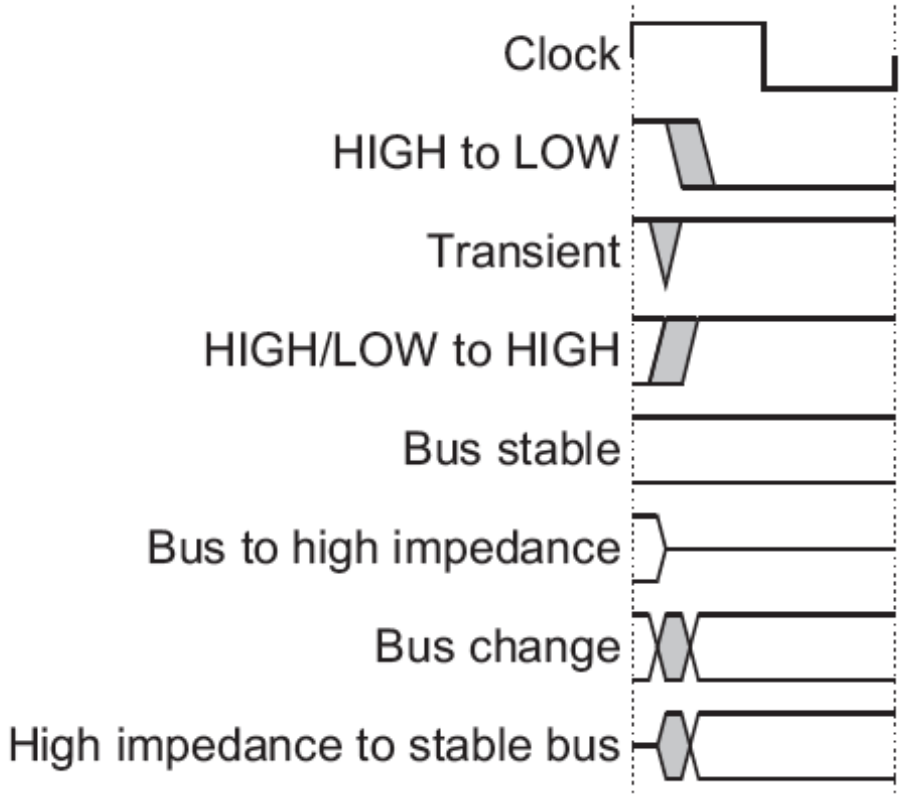

Source: The Zynq Book

# Prefixes of Ports from Particular Channels

- **Write Address Channel** — the signals contained within this channel are named in the format *s_axi_aw...*
- **Write Data Channel** — the signals contained within this channel are named in the format *s_axi_w...*
- **Write Response Channel** — the signals contained within this channel are named in the format *s_axi_b...*
- **Read Address Channel** — the signals contained within this channel are named in the format *s_axi_ar...*
- **Read Data Channel** — the signals contained within this channel are named in the format *s_axi_r...*

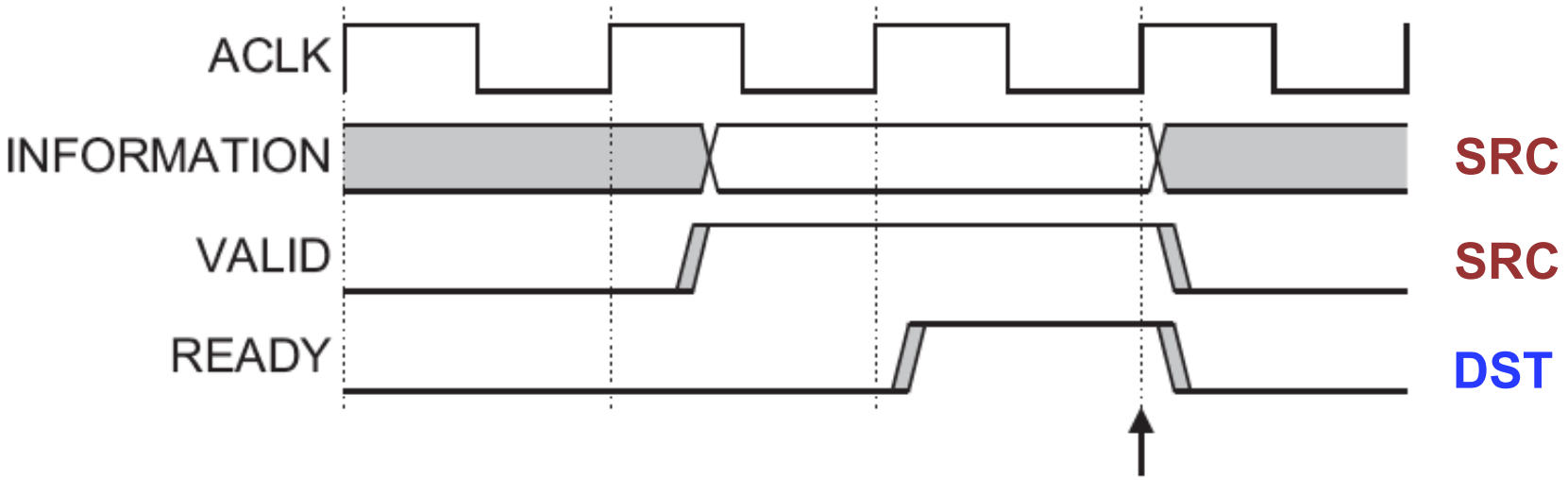
Source: The Zynq Book

# Timing Diagram Conventions



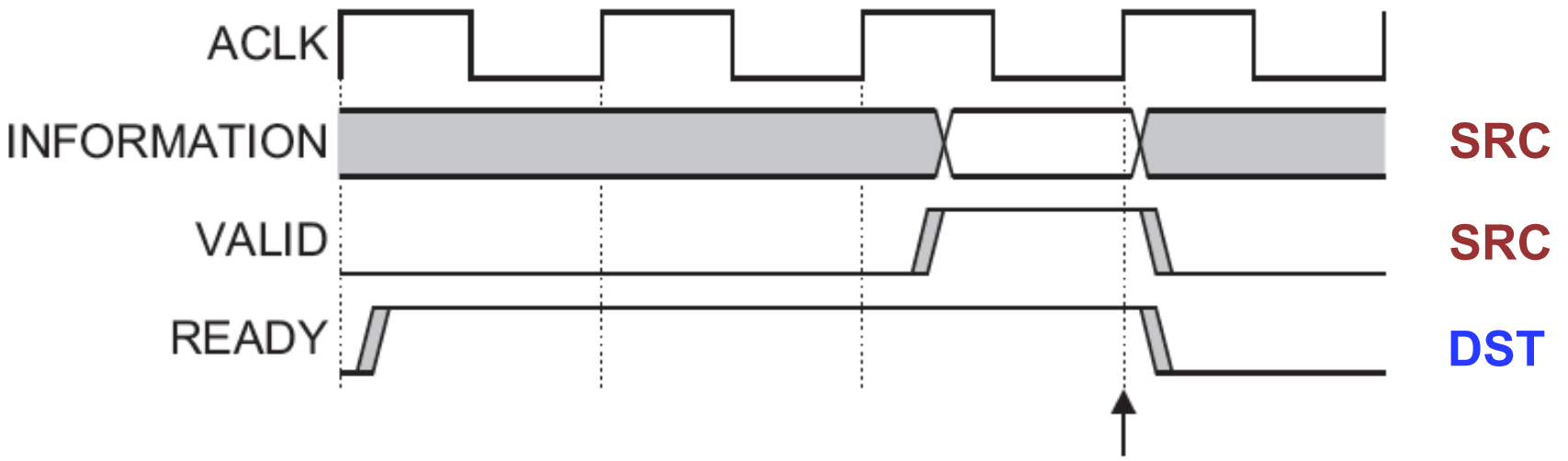

Source: ARM AMBA AXI Protocol v1.0: Specification

# VALID before READY Handshake



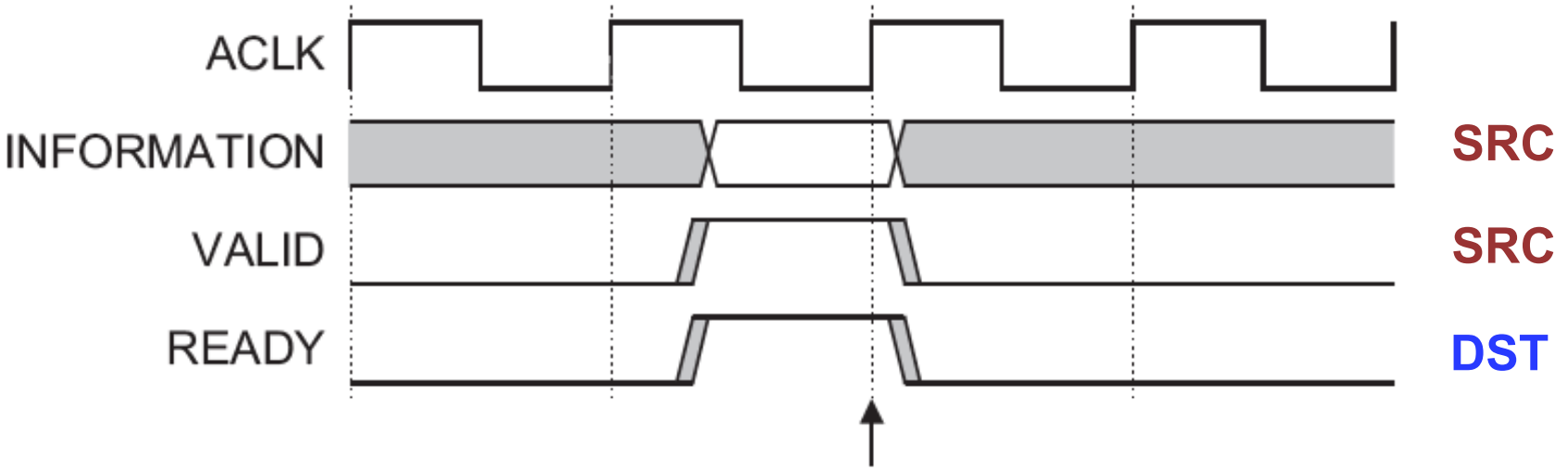

Source: ARM AMBA AXI Protocol v1.0: Specification

# READY before VALID Handshake



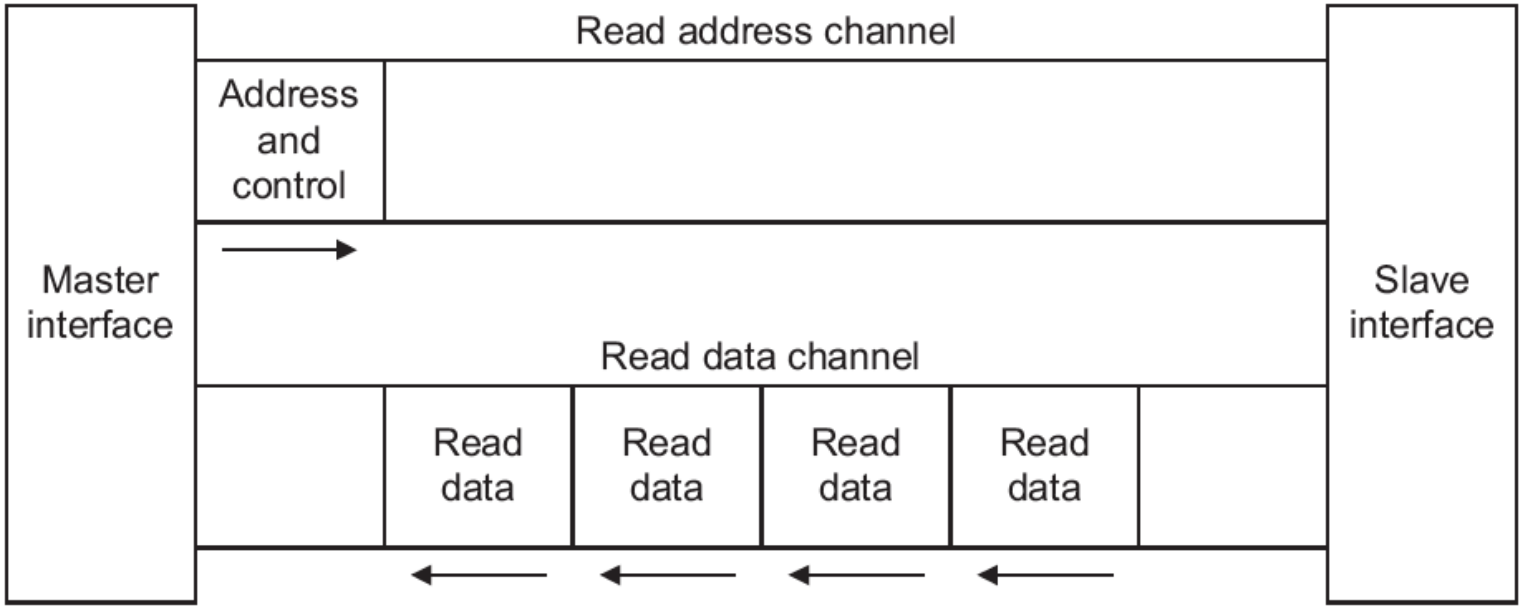

Source: ARM AMBA AXI Protocol v1.0: Specification

# VALID with READY Handshake



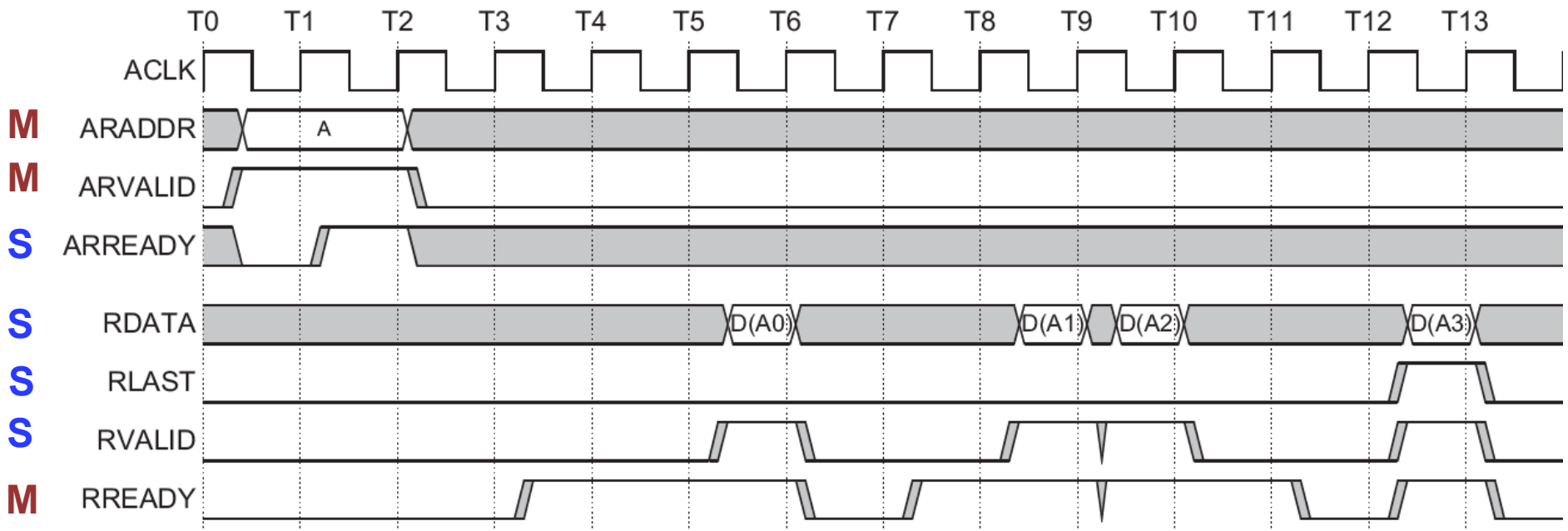

Source: ARM AMBA AXI Protocol v1.0: Specification

# Channel Architecture of Reads



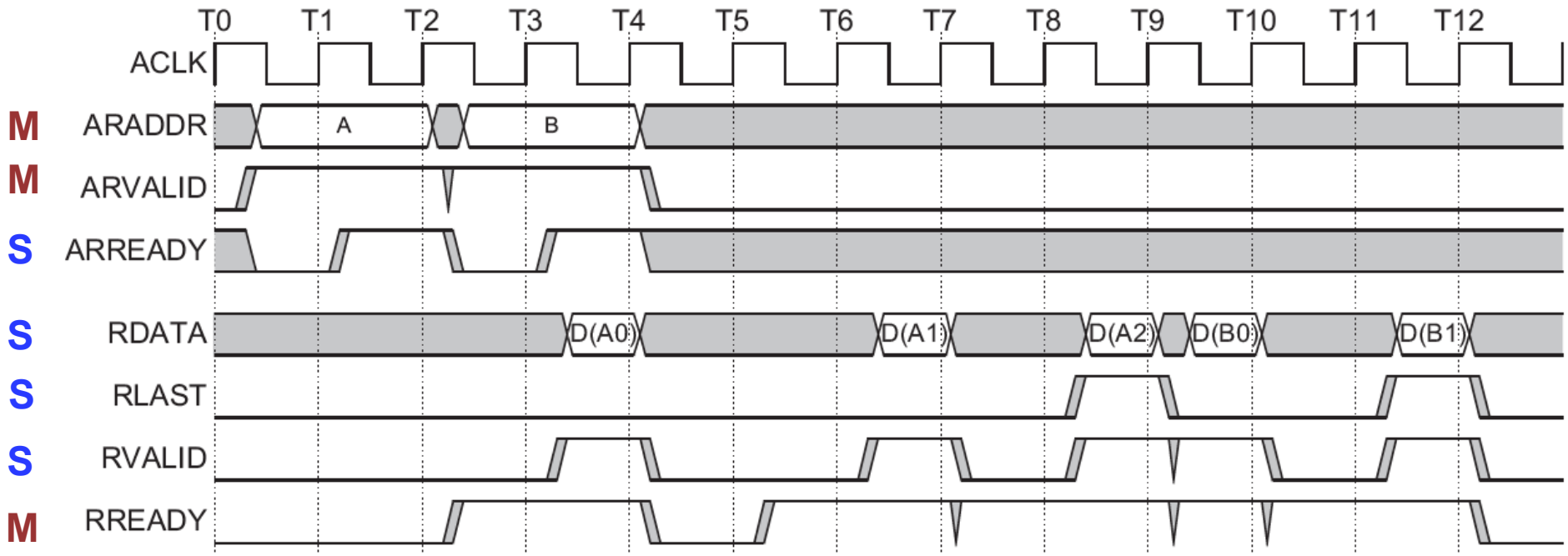

Source: ARM AMBA AXI Protocol v1.0: Specification

# Read Burst



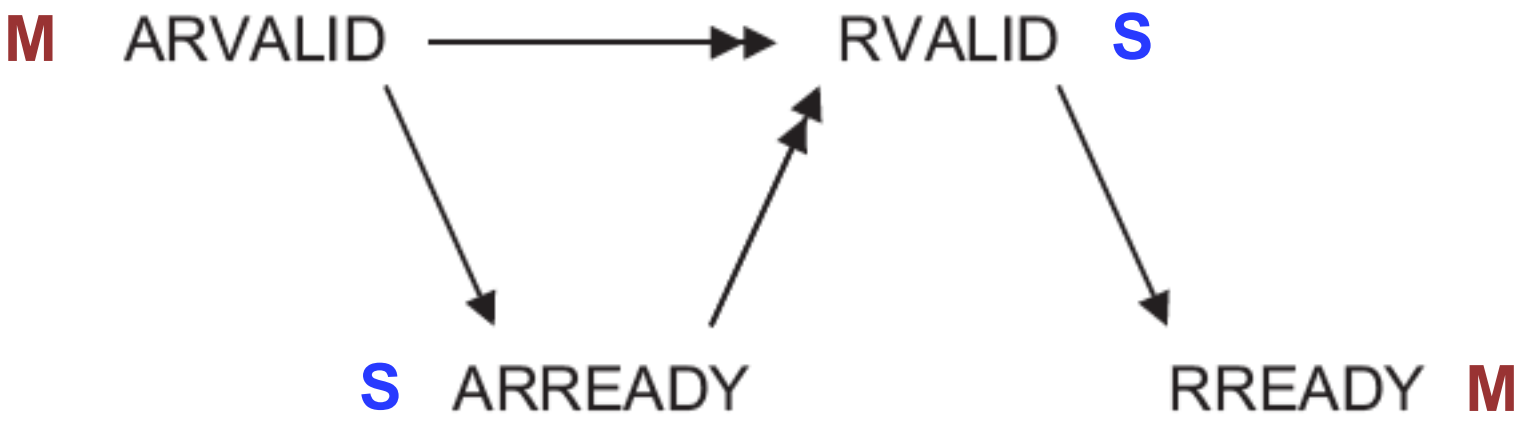

Source: ARM AMBA AXI Protocol v1.0: Specification

# Overlapping Read Bursts

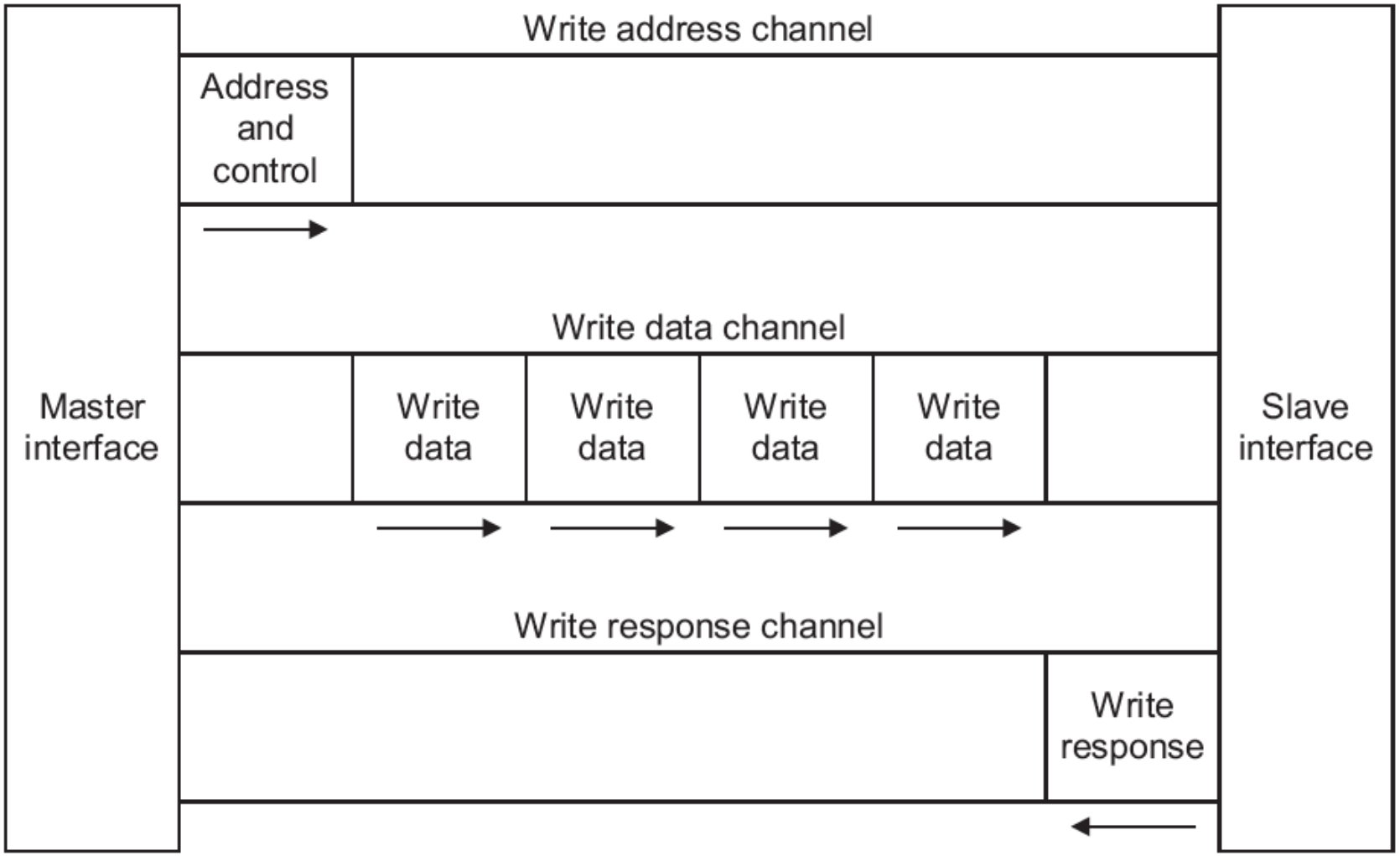



Source: ARM AMBA AXI Protocol v1.0: Specification

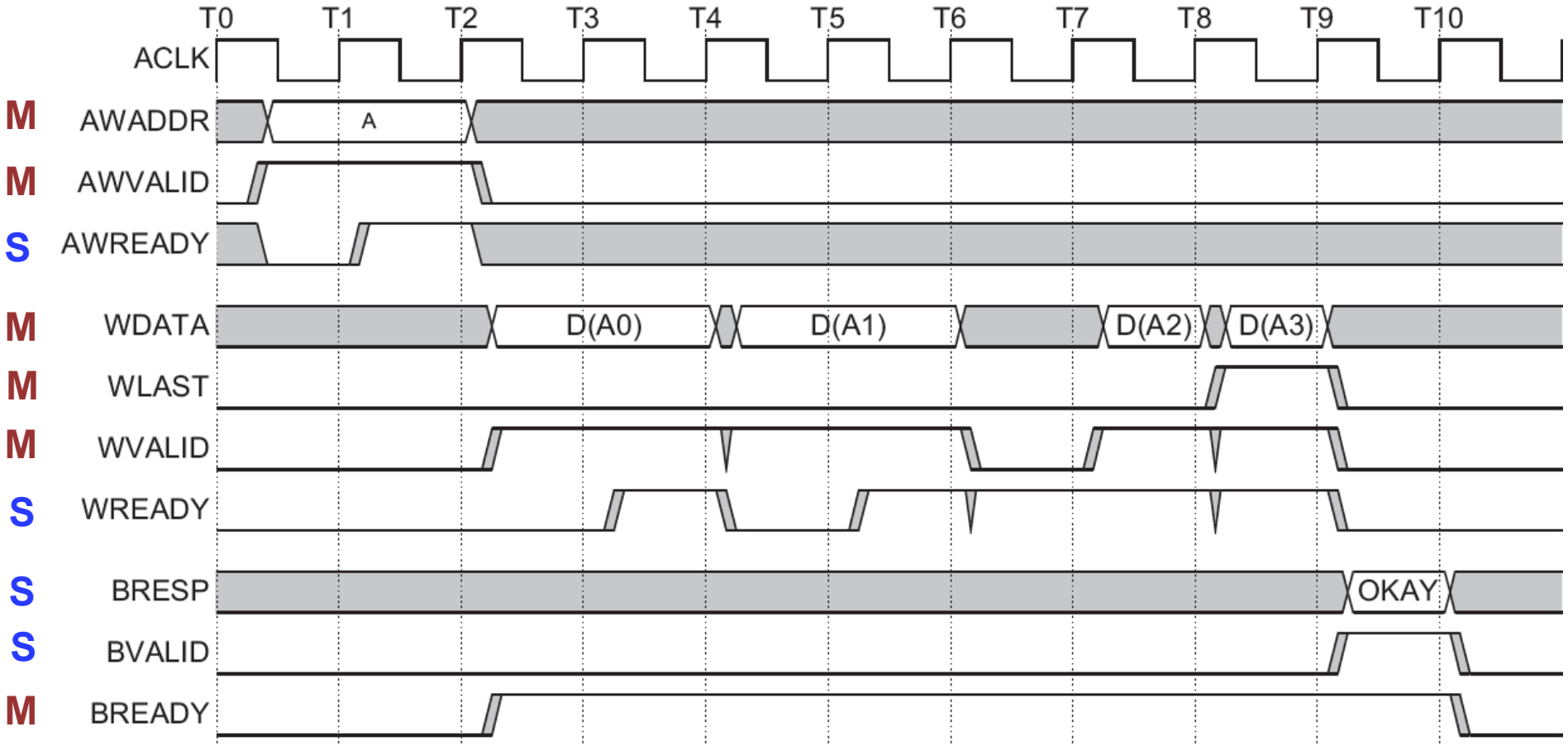
# Read Transaction
# Handshake Dependencies



# Channel Architecture of Writes

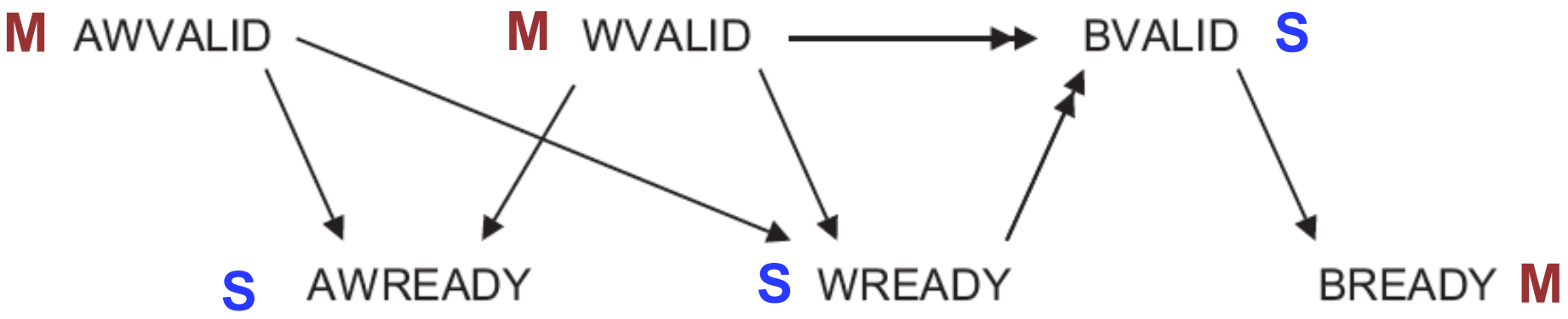


# Write Burst




Source: ARM AMBA AXI Protocol v1.0: Specification

# Write Transaction Handshake Dependencies

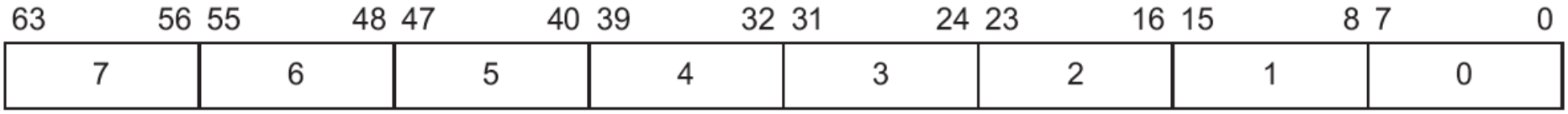



Source: ARM AMBA AXI Protocol v1.0: Specification

| ARBURST[1:0] AWBURST[1:0] | Burst type | Description | Access |
|---|---|---|---|
| b00 | FIXED | Fixed-address burst | FIFO-type |
| b01 | INCR | Incrementing-address burst | Normal sequential memory |
| b10 | WRAP | Incrementing-address burst that wraps to a lower address at the wrap boundary | Cache line |
| b11 | Reserved | - | - |

Source: ARM AMBA AXI Protocol v1.0: Specification

# Role of Write Strobe
# WSTRB




WSTRB[n] corresponds to

WDATA[8*n+7 downto 8*n]

Source: ARM AMBA AXI Protocol v1.0: Specification

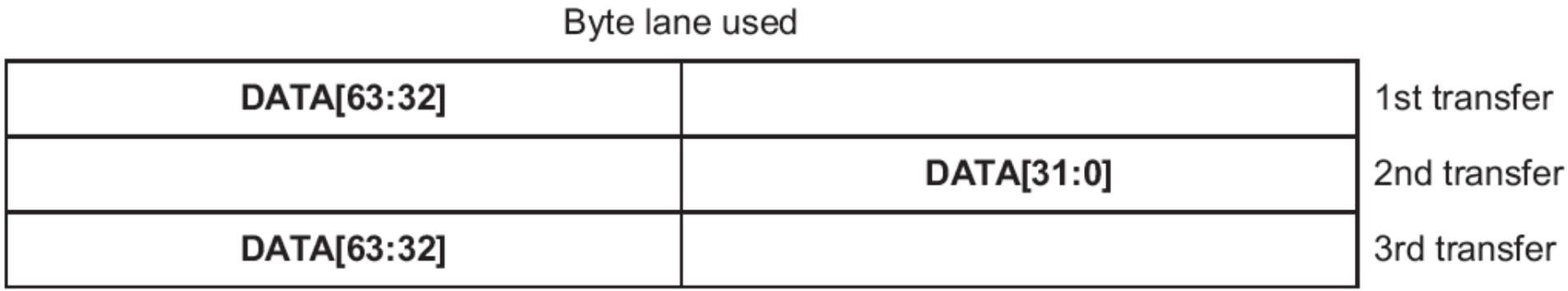
# Narrow Transfer Example with 8-bit Transfers

Byte lane used

| | | | | |
|---|---|---|---|---|
| | | | **DATA[7:0]** | 1st transfer |
| | | **DATA[15:8]** | | 2nd transfer |
| | **DATA[23:16]** | | | 3rd transfer |
| **DATA[31:24]** | | | | 4th transfer |
| | | | **DATA[7:0]** | 5th transfer |

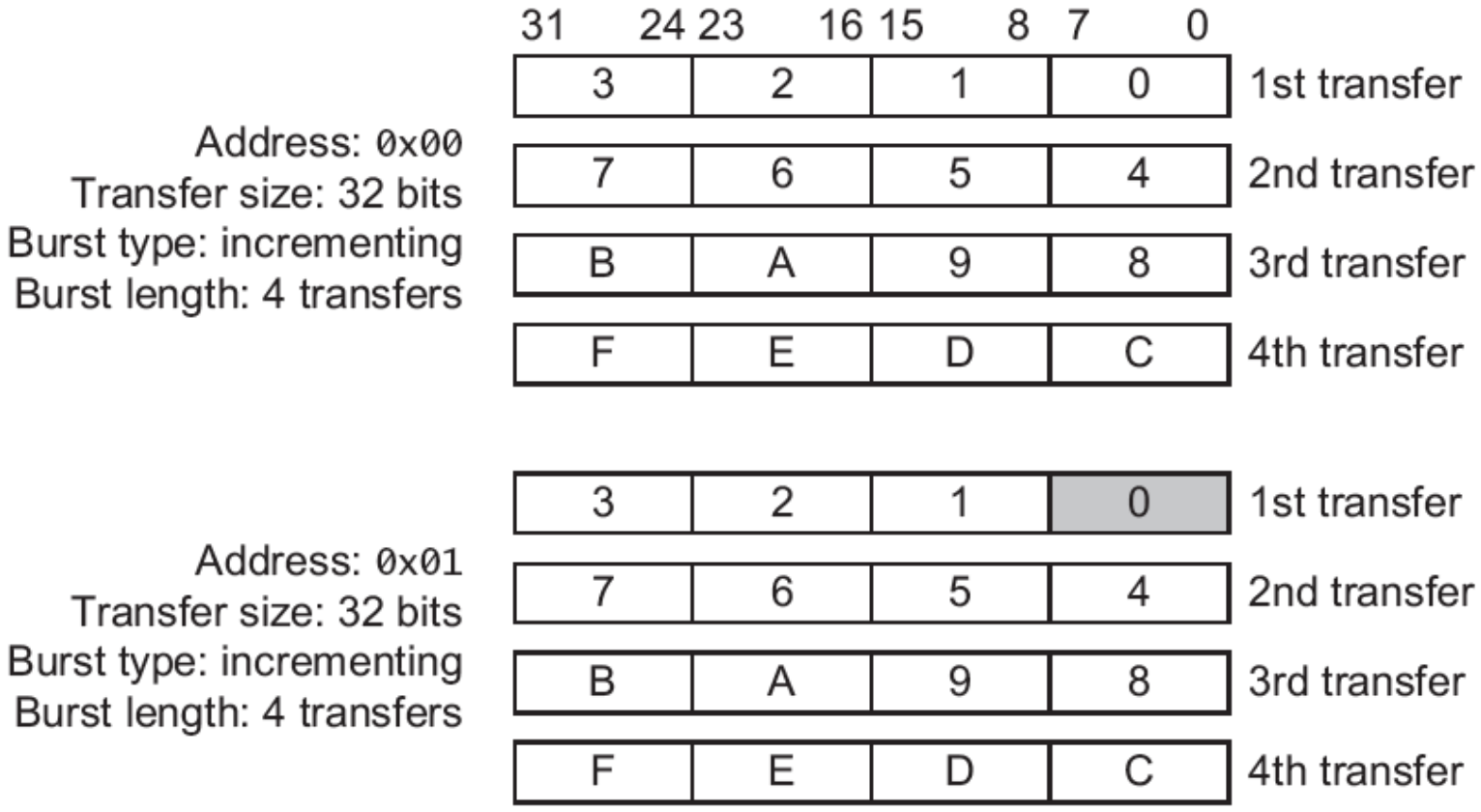
Source: ARM AMBA AXI Protocol v1.0: Specification

# Narrow Transfer Example with 32-bit Transfers



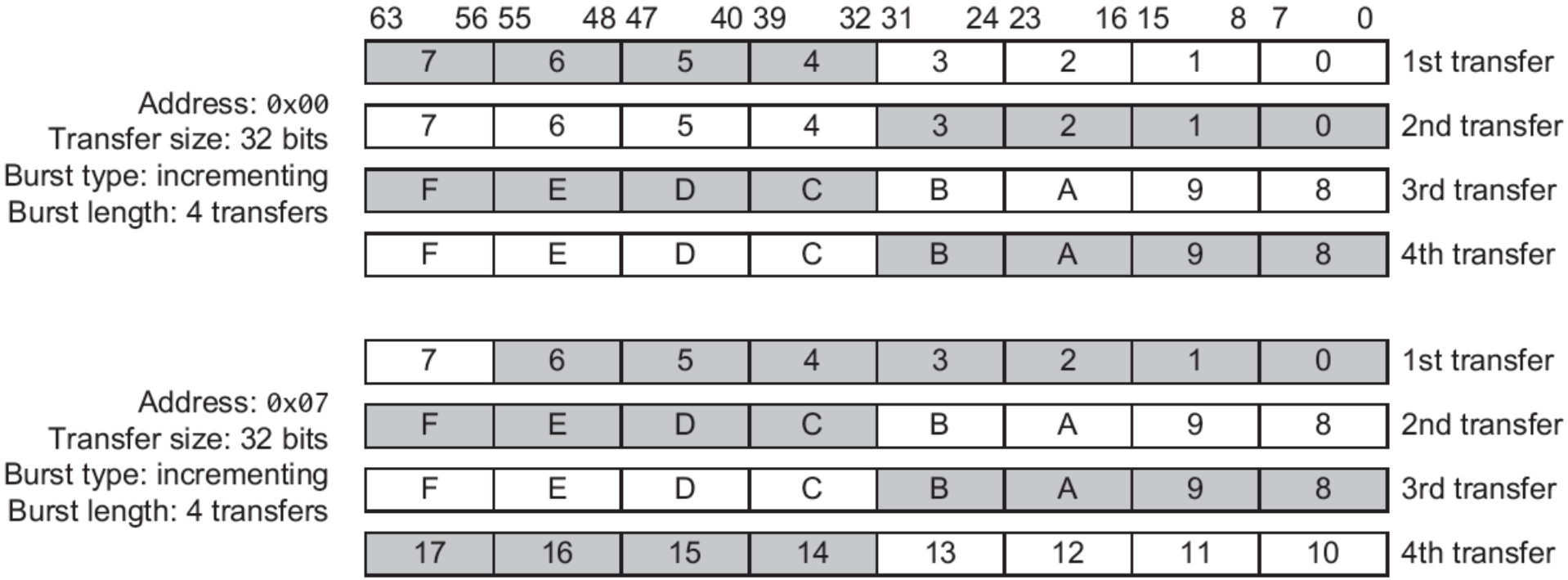

Source: ARM AMBA AXI Protocol v1.0: Specification

# Aligned and Unaligned Word Transfers on a 32-bit Bus




Source: ARM AMBA AXI Protocol v1.0: Specification

# Aligned and Unaligned Word Transfers on a 64-bit Bus

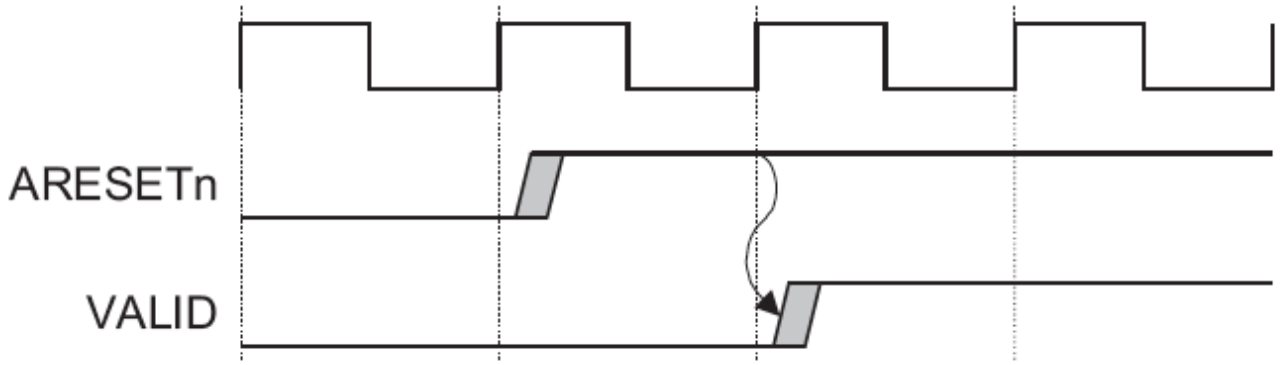



Source: ARM AMBA AXI Protocol v1.0: Specification

# Example of IP Core with AXI Interface

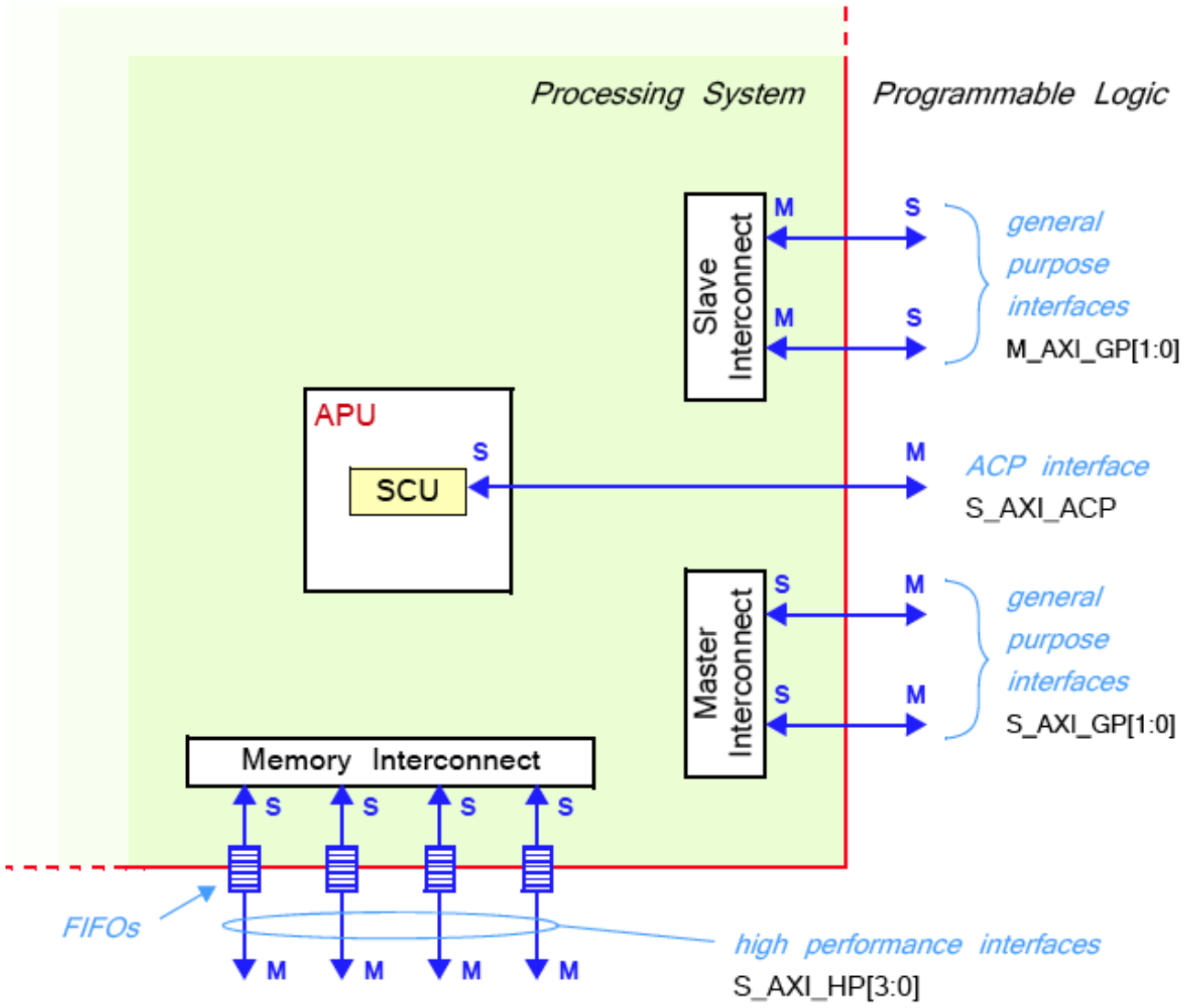

Source: The Zynq Book

# Exit from Reset



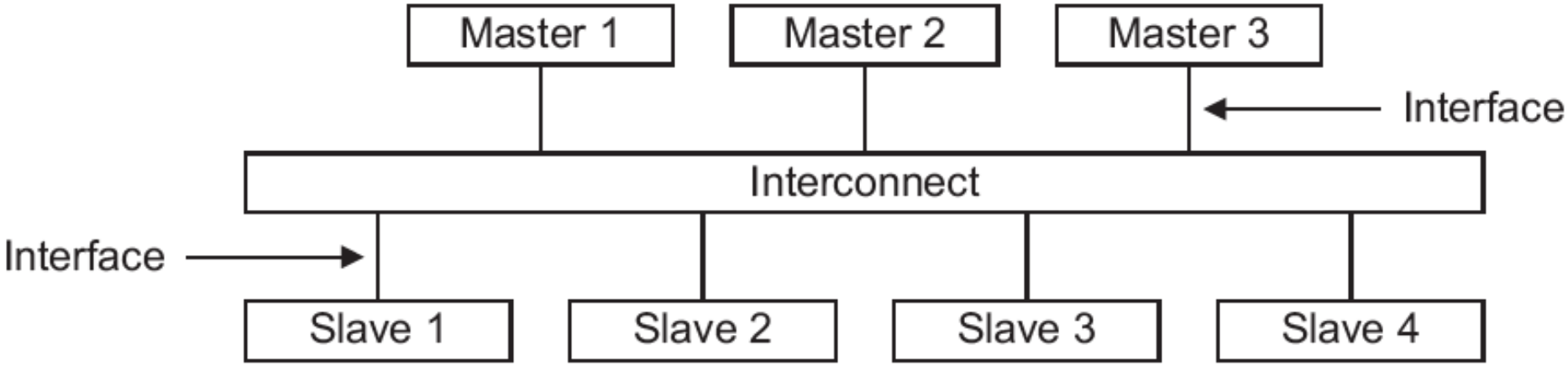

Source: ARM AMBA AXI Protocol v1.0: Specification

# PS-PL Interfaces and Interconnects




Source: The Zynq Book

# Interconnect vs. Interface



# AXI Interfaces and Interconnects

**Interface**

A point-to-point connection for passing data, addresses, and hand-shaking signals between master and slave clients within the system

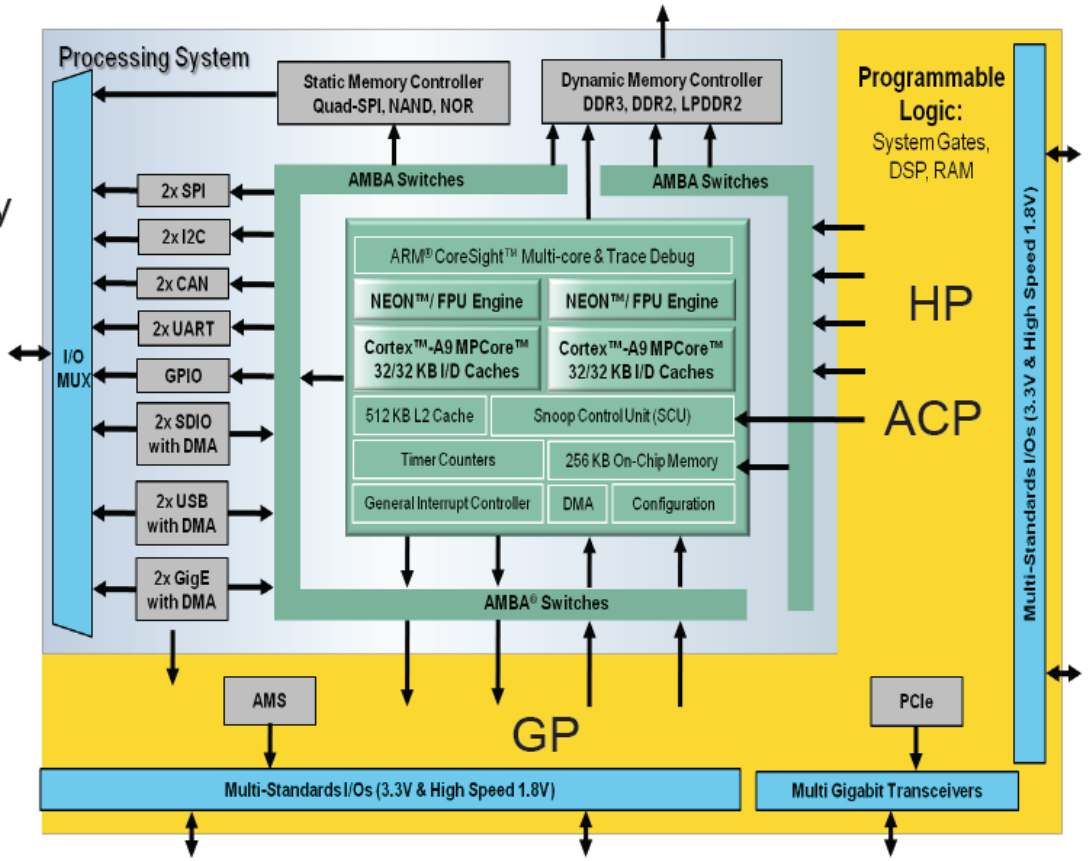
**Interconnect**

A switch which manages and directs traffic between attached AXI interfaces

Source: The Zynq Book

# Zynq AXI PS-PL Interfaces

- **HP**
  - 4 x 64 bit Slave interfaces
    - Optimized for high bandwidth access from PL to external memory
- **GP**
  - 2 x 32 bit Slave interfaces
    - Optimized for access from PL to PS peripherals
  - 2 x 32 bit Master interfaces
    - Optimized for access from processors to PL registers
- **ACP**
  - 1 x 64 bit Slave interface
    - Optimized for access from PL to processor caches




ACP - Accelerator Coherency Port

# AXI Interfaces between PS and PL

| Interface Name | Interface Description | Master | Slave |
|---|---|---|---|
| M_AXI_GP0 | General Purpose (AXI_GP) | PS | PL |
| M_AXI_GP1 | | PS | PL |
| S_AXI_GP0 | General Purpose (AXI_GP) | PL | PS |
| S_AXI_GP1 | | PL | PS |
| S_AXI_ACP | Accelerator Coherency Port (ACP), cache coherent transaction | PL | PS |
| S_AXI_HP0 | High Performance Ports (AXI_HP) with read/write FIFOs. (Note that AXI_HP interfaces are sometimes referred to as AXI Fifo Interfaces, or AFIs). | PL | PS |
| S_AXI_HP1 | | PL | PS |
| S_AXI_HP2 | | PL | PS |
| S_AXI_HP3 | | PL | PS |

Source: The Zynq Book

# General-Purpose Port Summary

- GP ports are designed for maximum flexibility
- Allow register access from PS to PL or PL to PS
- Good for Synchronization
- Prefer ACP or HP port for data transport

# High-Performance Port Summary

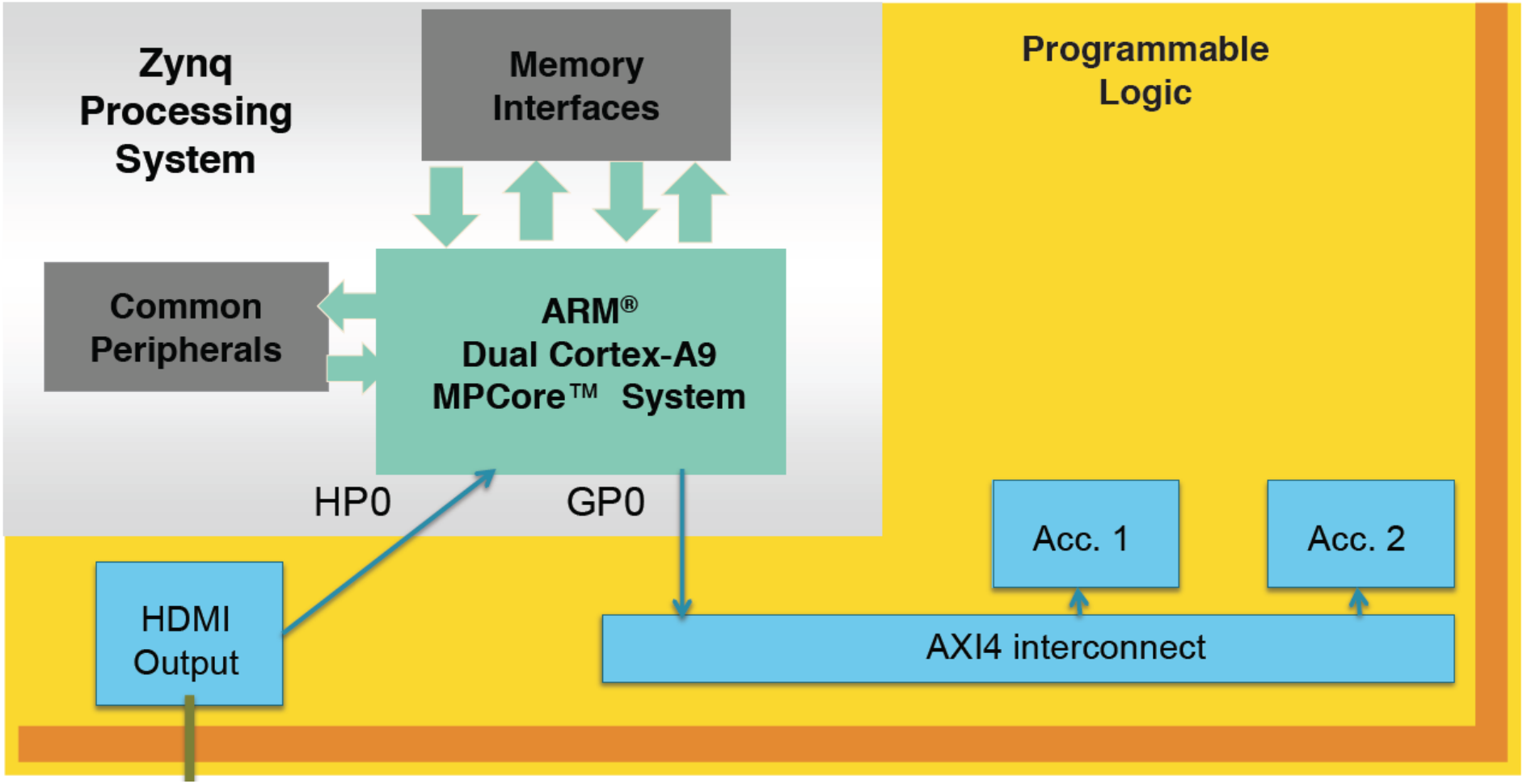
- HP ports are designed for maximum bandwidth access to external memory and On-Chip Memory (OCM)
- When combined can saturate external memory and OCM bandwidth
  - HP Ports : 4 * 64 bits * 150 MHz * 2 = 9.6 GByte/sec
  - external DDR: 1 * 32 bits * 1066 MHz * 2 = 4.3 GByte/sec
  - OCM : 64 bits * 222 MHz * 2 = 3.5 GByte/sec
- Optimized for large burst lengths and many outstanding transactions
- Large data buffers to amortize access latency
- Efficient upsizing/downsizing for 32 bit accesses

# Accelerator Coherency Port (ACP) Summary

- ACP allows limited support for Hardware Coherency
  - Allows a PL accelerator to access cache of the Cortex-A9 processors
  - PL has access through the same path as CPUs including caches, OCM, DDR, and peripherals
  - Access is low latency (assuming data is in processor cache) no switches in path
- ACP does not allow full coherency
  - PL is not notified of changes in processor caches
  - Use write to PL register for synchronization
- ACP is compromise between bandwidth and latency
  - Optimized for cache line length transfers
  - Low latency for L1/L2 hits
  - Minimal buffering to hide external memory latency
  - One shared 64 bit interface, limit of 8 masters

# Accelerator Architecture with Bus Slave

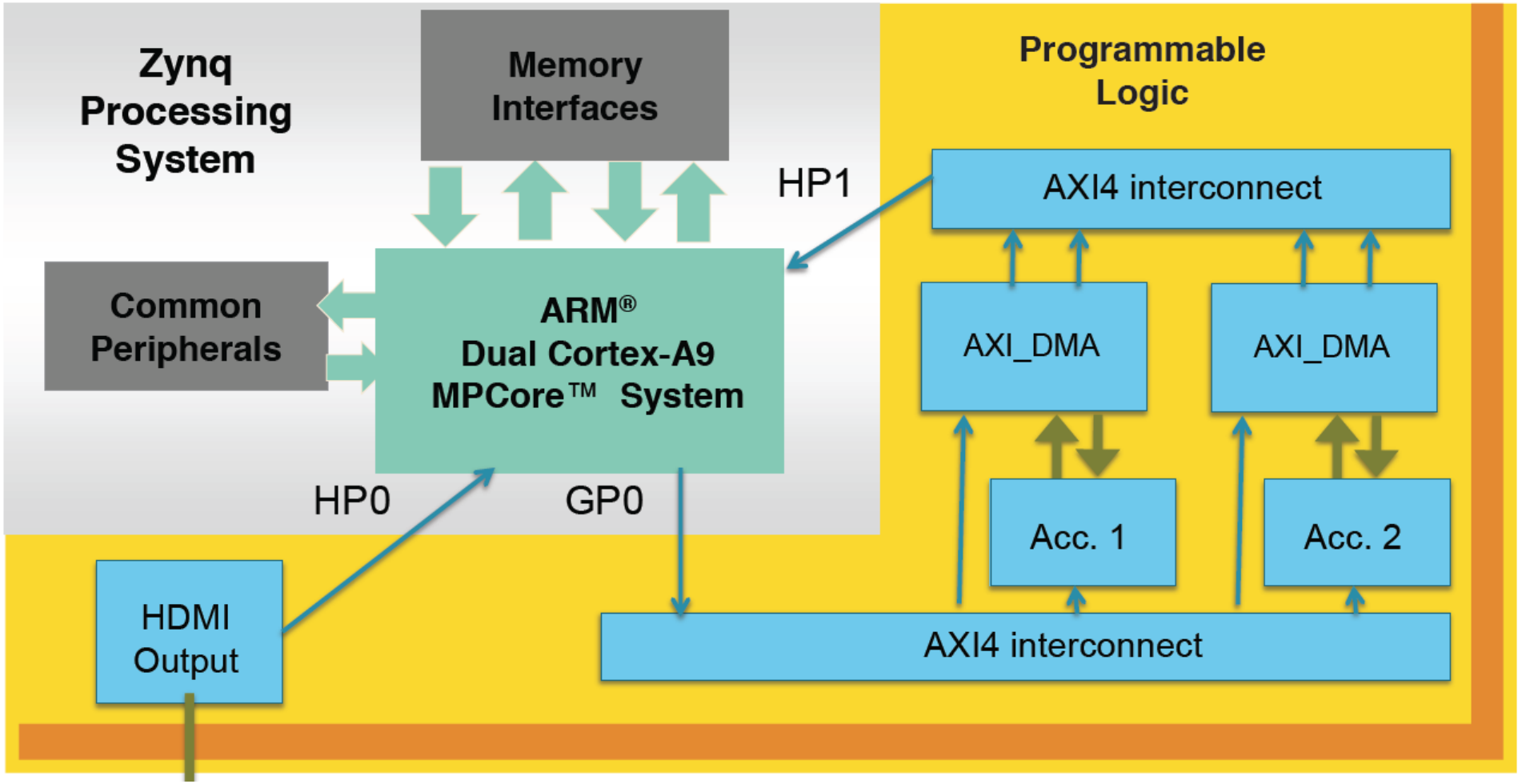
Pro: Simple System Architecture
Con: Limited communication bandwidth




Source: Building Zynq Accelerators with Vivado HLS, FPL 2013 Tutorial

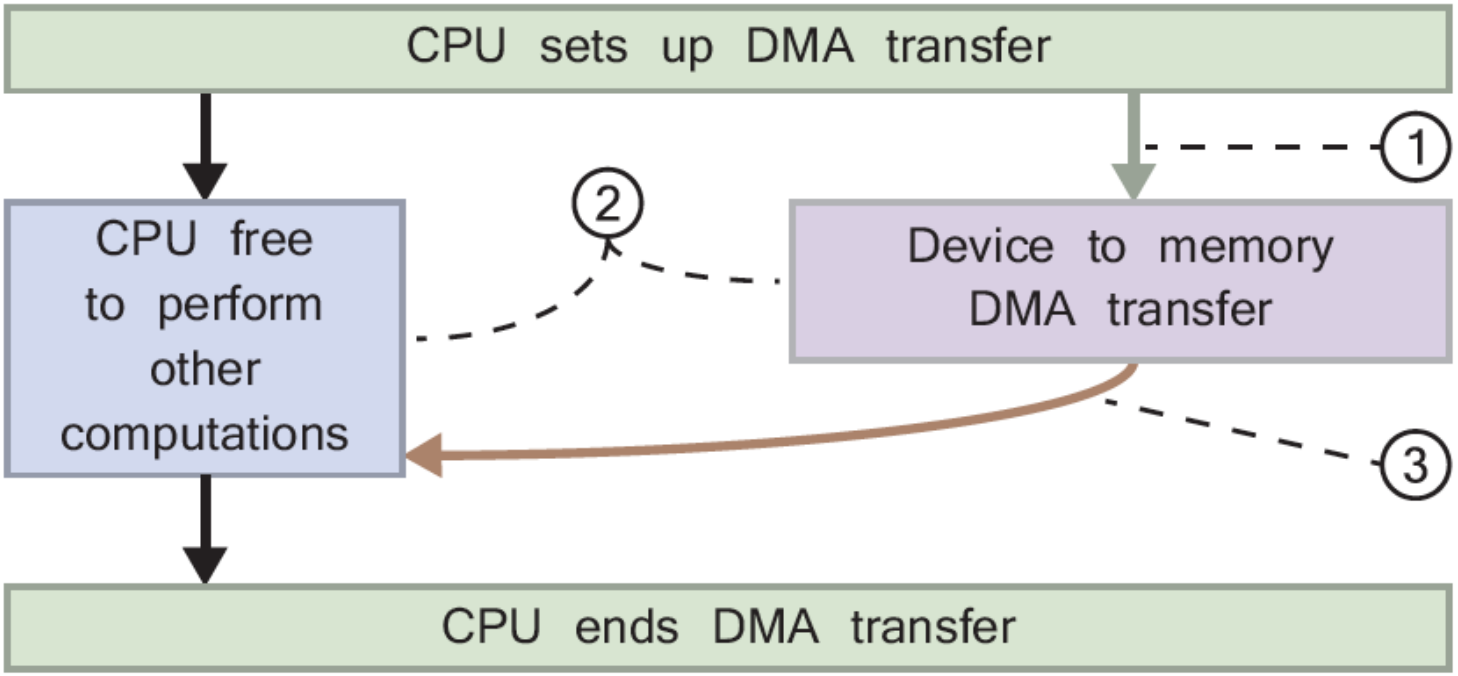
# Accelerator Architecture with DMA

Pro: High Bandwidth Communication
Con: Complicated System Architecture, High Latency




Source: Building Zynq Accelerators with Vivado HLS, FPL 2013 Tutorial

# DMA Memory Transfer Operation




Source: The Zynq Book

# AXI DMA-based Accelerator Communication

**Write to Accelerator**

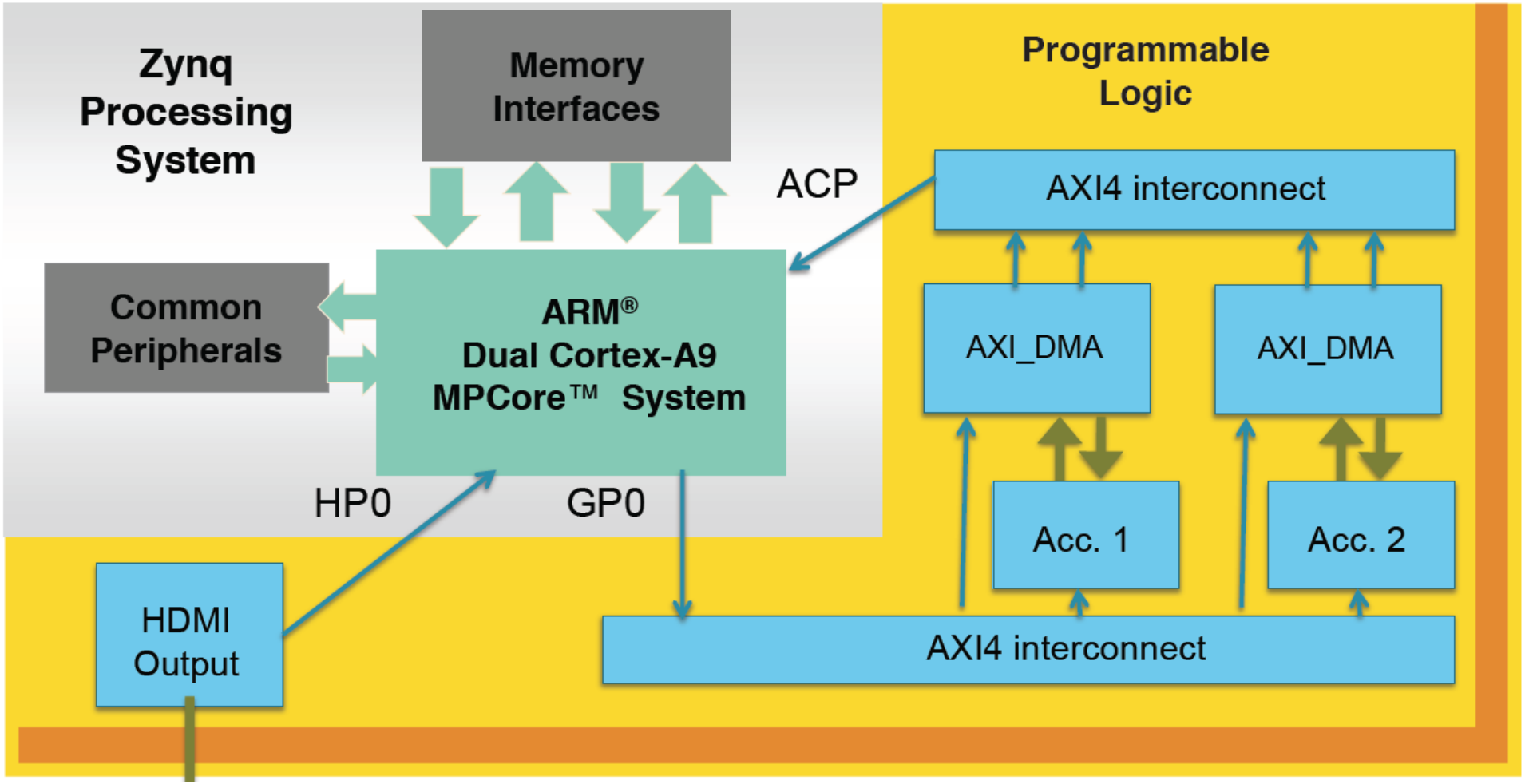
- processor allocates buffer
- processor writes data into buffer
- processor flushes cache for buffer
- processor initiates DMA transfer

**Read from Accelerator**

- processor allocates buffer
- processor initiates DMA transfer
- processor waits for DMA to complete
- processor invalidates cache for buffer
- processor reads data from buffer

# Accelerator Architecture with Coherent DMA

Pro: Low latency, high-bandwidth communication
Con: Complicated system architecture, Limited to data that fits in caches




Source: Building Zynq Accelerators with Vivado HLS, FPL 2013 Tutorial

# Coherent AXI DMA-based Accelerator Communication

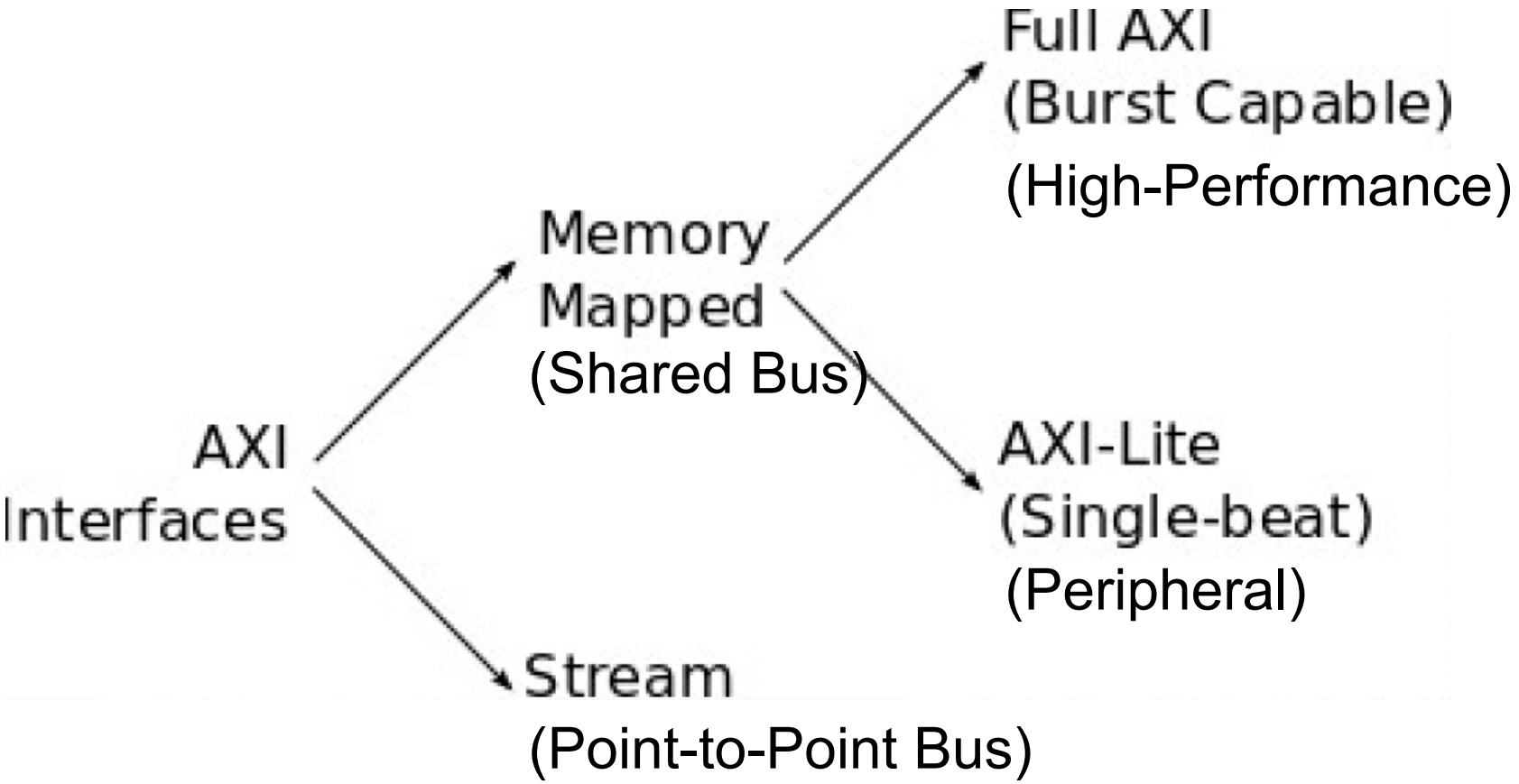
**Write to Accelerator**

- processor allocates buffer
- processor writes data into buffer
- ~~processor flushes cache for buffer~~
- processor initiates DMA transfer

**Read from Accelerator**

- processor allocates buffer
- processor initiates DMA transfer
- processor waits for DMA to complete
- ~~processor invalidates cache for buffer~~
- processor reads data from buffer

# AXI Interfaces




Source: M.S. Sadri, Zynq Training

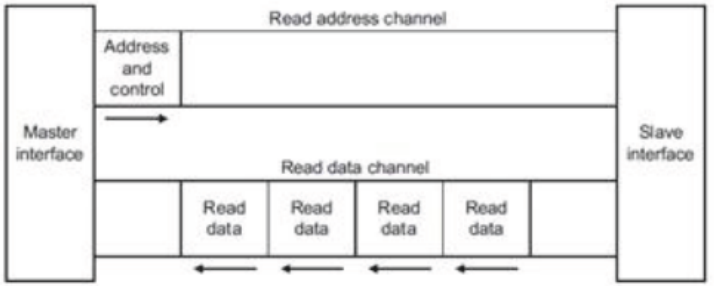
# Features of AXI Interfaces

| Interface | Features |
|---|---|
| MemoryMap/Full | Traditional address/data burst (single address,multiple data) |
| Streaming | Dataonly,burst |
| Lite | Traditional address/data—no burst (single address,multiple data) |

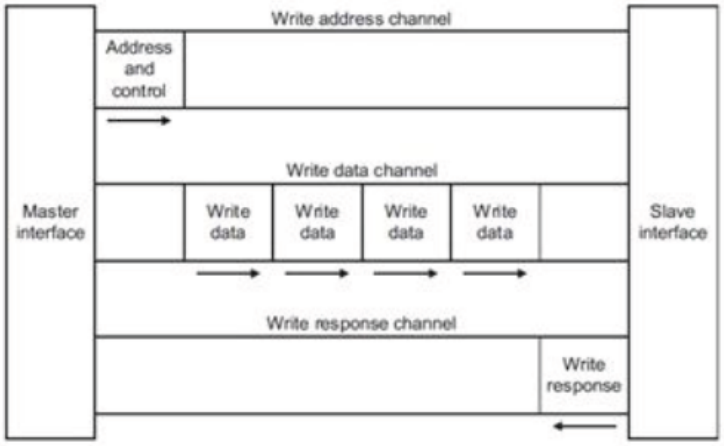
Source: Building Zynq Accelerators with Vivado HLS, FPL 2013 Tutorial

# Summary of AXI Full and AXI Lite Interfaces

- **Memory mapped interfaces consist of 5 streams**
  - Read Address
  - Read Data
  - Write Address
  - Write Data
  - Write Acknowledge
- **Burst length limited to 256**
- **Data width limited to 256 bits for Xilinx IP**
- **AXI Lite is a subset**
  - no bursts
  - 32 bit data width only




AXI4 READ

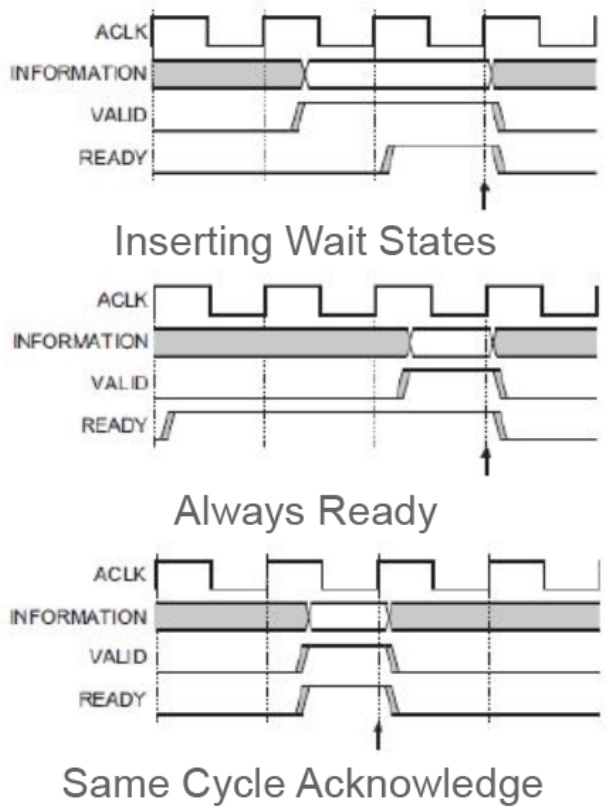



AXI4 Write

Source: Building Zynq Accelerators with Vivado HLS, FPL 2013 Tutorial

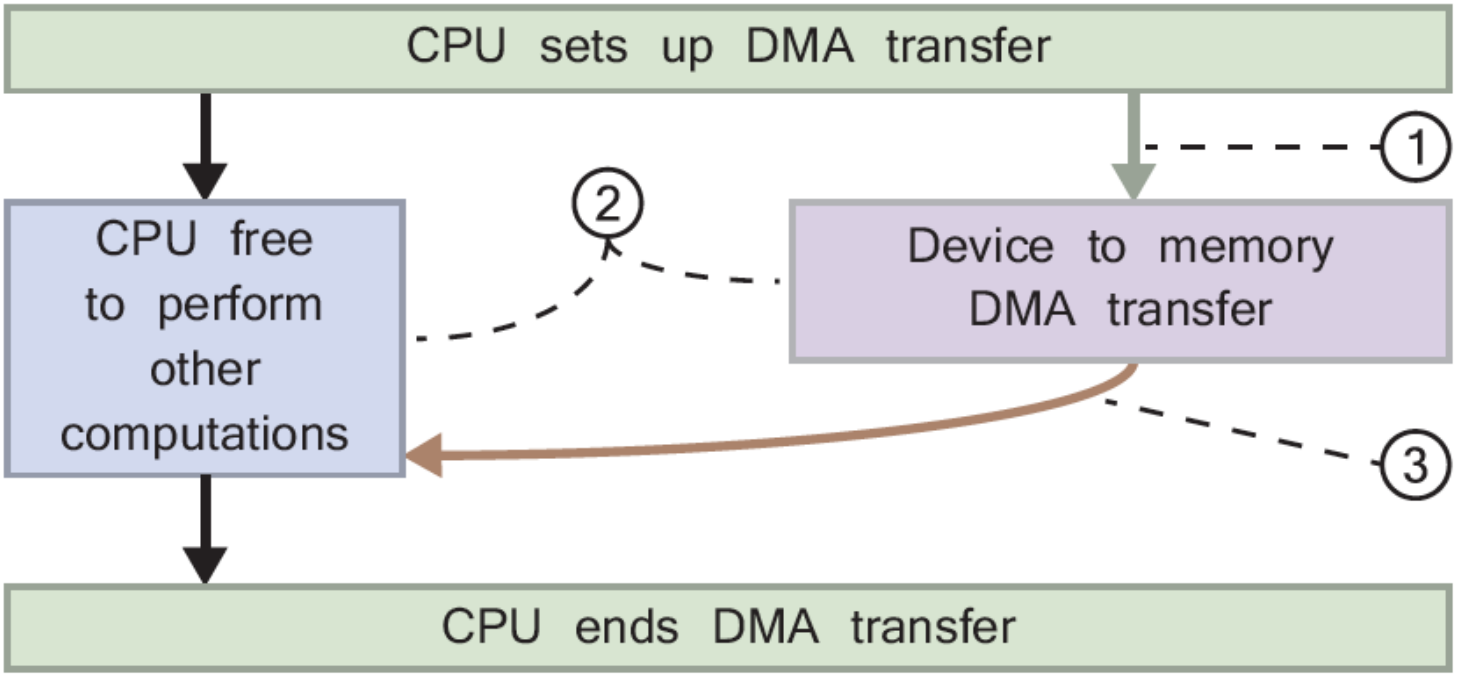
# Summary of AXI Stream Interface

- **AXI Streams are fully handshaked**
  - Data is transferred when source asserts VALID and destination asserts READY
- **'Information' includes DATA and other side channel signals**
  - STRB
  - KEEP
  - LAST
  - ID
  - DEST
  - USER
- **Most of these are optional**



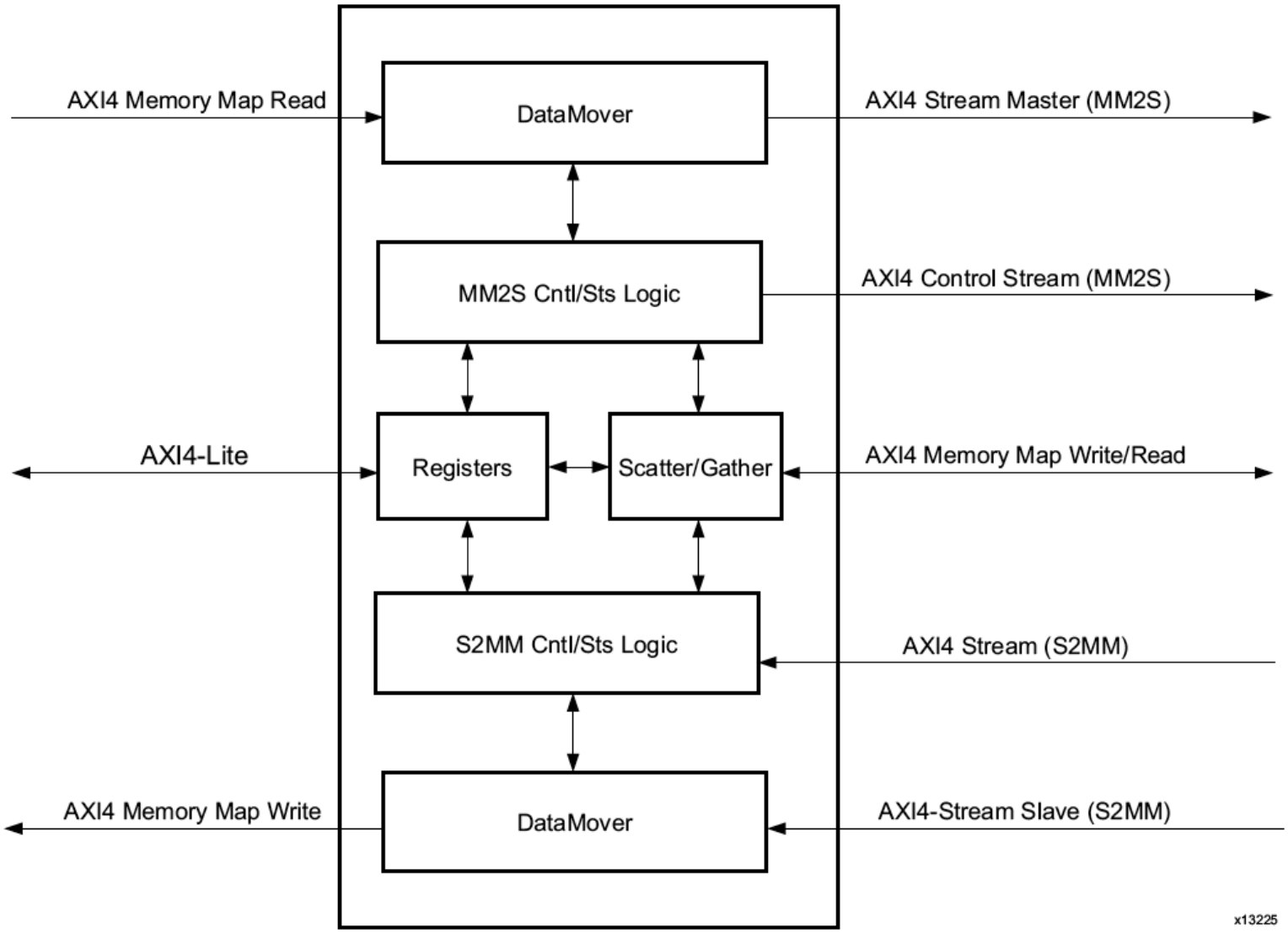

Source: Building Zynq Accelerators with Vivado HLS, FPL 2013 Tutorial

# DMA Memory Transfer Operation



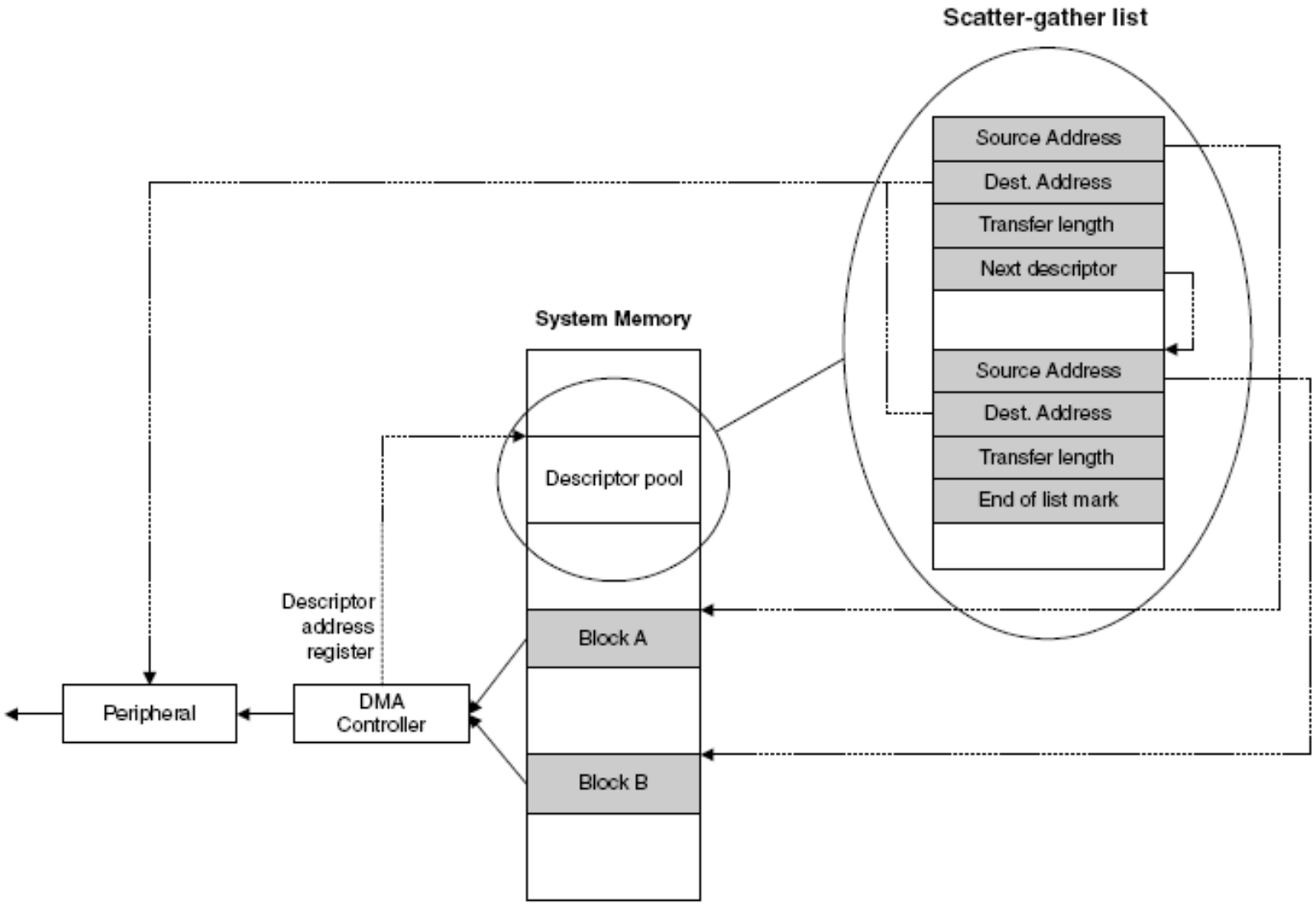

Source: The Zynq Book

# Block Diagram of AXI DMA Core



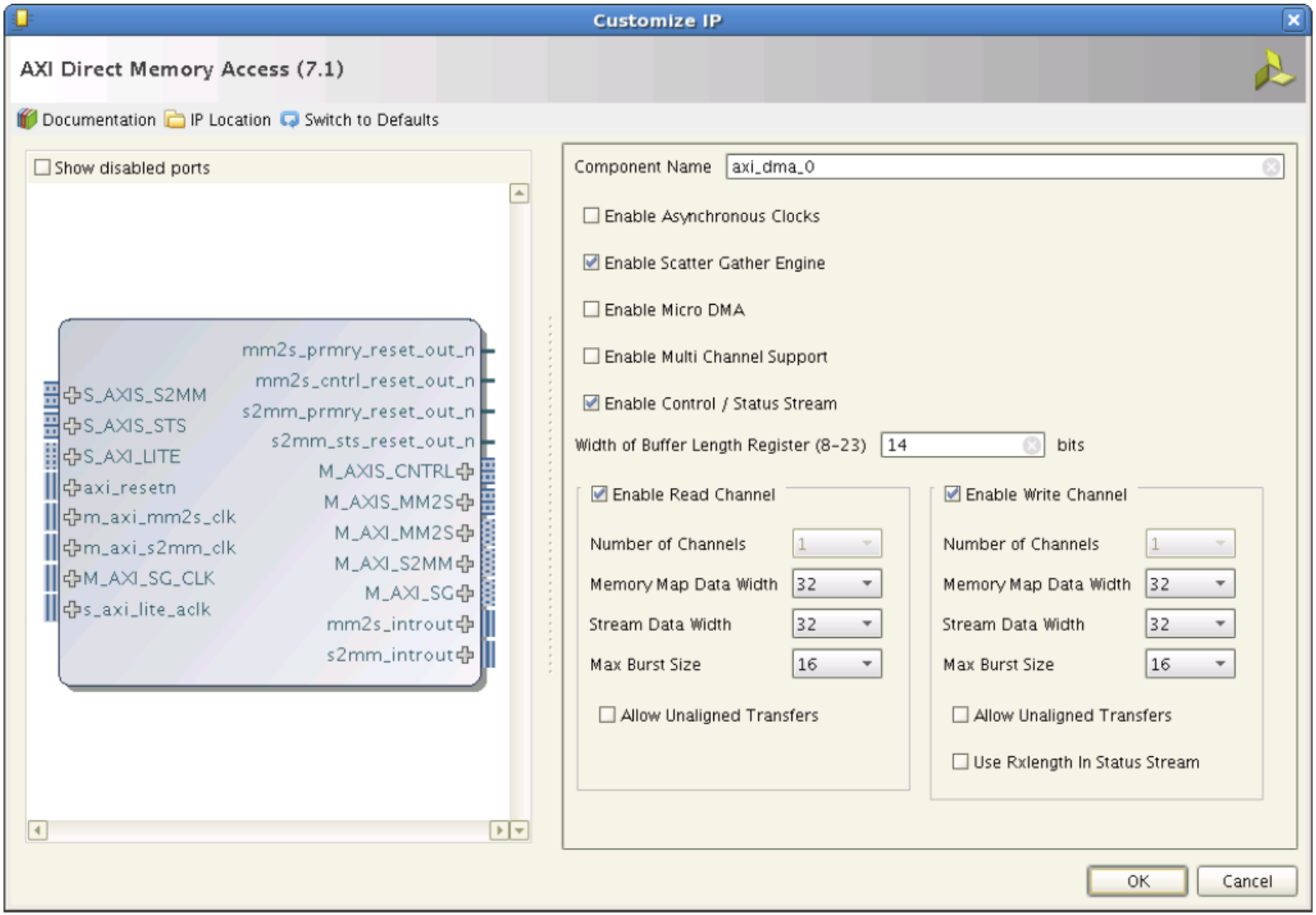

Source: LogiCORE IP AXI DMA v7.1: Product Guide

# Scatter Gather DMA Mode




Source: Symbian OS Internals/13. Peripheral Support

# Customizing AXI DMA Core




Source: LogiCORE IP AXI DMA v7.1: Product Guide

# Parameters of AXI DMA Core (1)

**Number of Channels**

the number of channels: 1..16

**Memory Map Data Width**

data width in bits of the AXI MM2S Memory Map Read data bus: 32, 64, 128, 256, 512 or 1,024

**Stream Data Width**

data width in bits of the AXI MM2S AXI4-Stream Data bus: 8, 16, 32, 64, 128, 512 or 1,024;
Stream Data Width ≤ Memory Map Data Width;

**Max Burst Size**

maximum size of burst on the AXI4-Memory Map side of MM2S: 2, 4, 8,16, 32, 64, 128, or 256

# Options of AXI DMA Core (1)

**Enable Asynchronous Clocks**

0 – all clocks inputs should be connected to the same clock signal
1 – separate asynchronous clocks for MM2S interface, S2MM interface, AXI4-Lite control interface, and the Scatter Gather Interface

**Enable Scatter Gather Engine**

0 – Simple DMA Mode operation
1 – Scatter Gather Mode operation; the Scatter Gather Engine included in AXI DMA

# Options of AXI DMA Core (2)

**Enable Micro DMA**

0 – regular DMA
1 – area-optimized DMA; the maximum number of bytes per transaction = MMap_Data_width * Burst_length/8; addressing restricted to burst boundries

**Enable Multi Channel Support**

0 – the number of channels fixed at 1 for both directions
1 – the number of channels can be greater than 1

# Options of AXI DMA Core (3)

**Enable Control/Status Stream**

0 – no AXI4 Control/Status Streams
1 – The AXI4 Control stream allows user application metadata associated with the MM2S channel to be transmitted to a target IP. The AXI4 Status stream allows user application metadata associated with the S2MM channel to be received from a target IP.

**Width of Buffer Length Register (8-23)**

For Simple DMA mode:
the number of valid bits in the MM2S_LENGTH and S2MM_LENGTH registers
For Scatter Gather mode:
the number of valid bits used for the Control field *buffer length* and Status field *bytes transferred* in the Scatter/Gather descriptors

# Options of AXI DMA Core (4)

**Allow Unaligned Transfers**

Enables or disables the MM2S Data Realignment Engine (DRE). If the DRE is enabled, data reads can start from any Buffer Address byte offset, and the read data is aligned such that the first byte read is the first valid byte out on the AXI4-Stream.

**Use RxLength In Status Stream**

Allows AXI DMA to use a receive length field that is supplied by the S2MM target IP in the App4 field of the status packet. This gives AXI DMA a pre-determined receive byte count, allowing AXI DMA to command the exact number of bytes to be transferred.

# Simple DMA Transfer
## Programming Sequence for MM2S channel (1)

1. **Start the MM2S channel running** by setting the run/stop bit to 1, MM2S_DMACR.RS = 1.
2. If desired, **enable interrupts** by writing a 1 to MM2S_DMACR.IOC_IrqEn and MM2S_DMACR.Err_IrqEn.
3. Write a **valid source address** to the MM2S_SA register.
4. Write the **number of bytes to transfer** in the MM2S_LENGTH register.

   The MM2S_LENGTH register must be written last.

   All other MM2S registers can be written in any order.

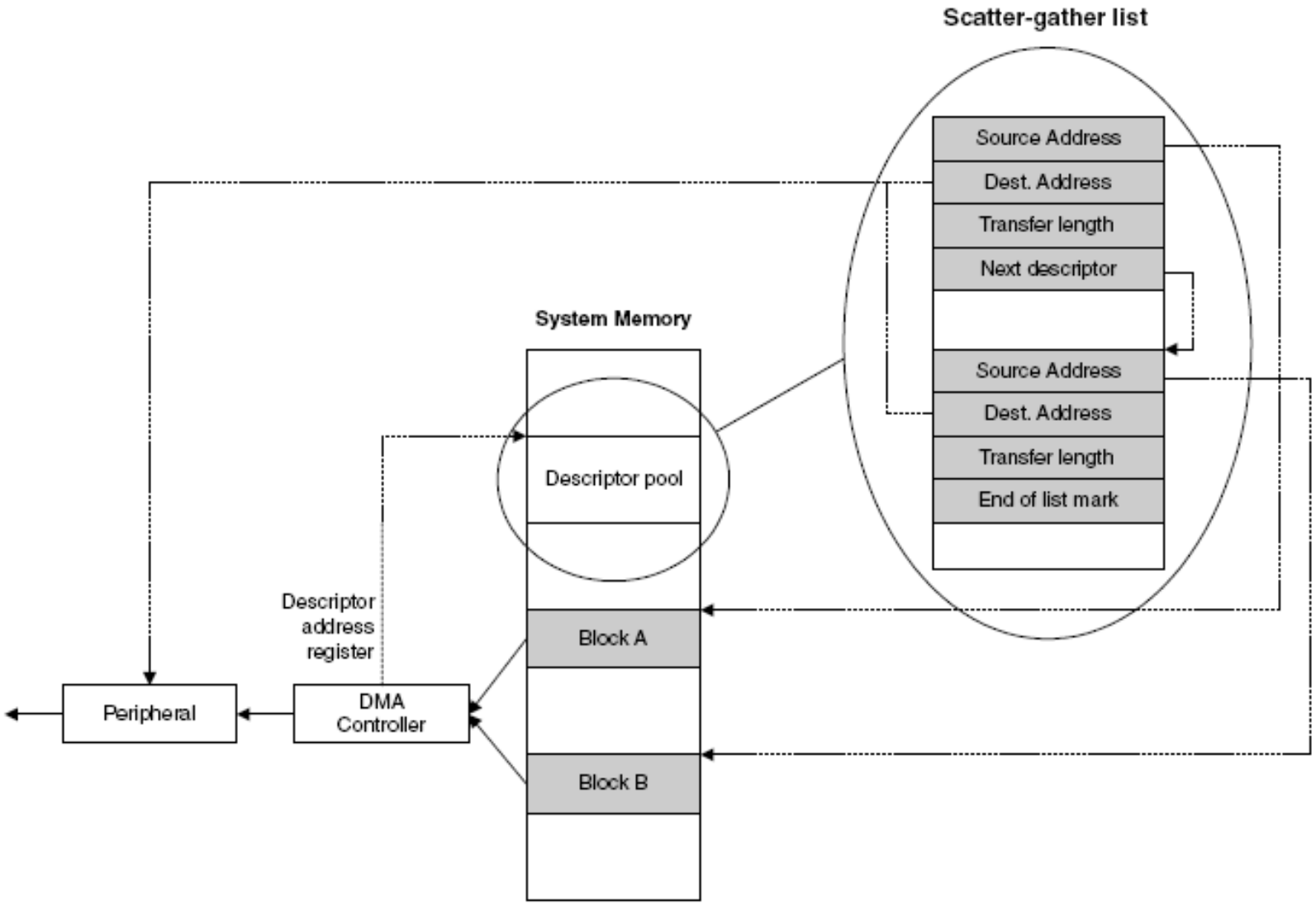
# Simple DMA Transfer
## Programming Sequence for S2MM channel (1)

1. **Start the S2MM channel running** by setting the run/stop bit to 1, S2MM_DMACR.RS = 1.
2. If desired, **enable interrupts** by by writing a 1 to S2MM_DMACR.IOC_IrqEn and S2MM_DMACR.Err_IrqEn.
3. Write a **valid destination address** to the S2MM_DA register.
4. Write the **length in bytes of the receive buffer** in the S2MM_LENGTH register.

   The S2MM_LENGTH register must be written last.

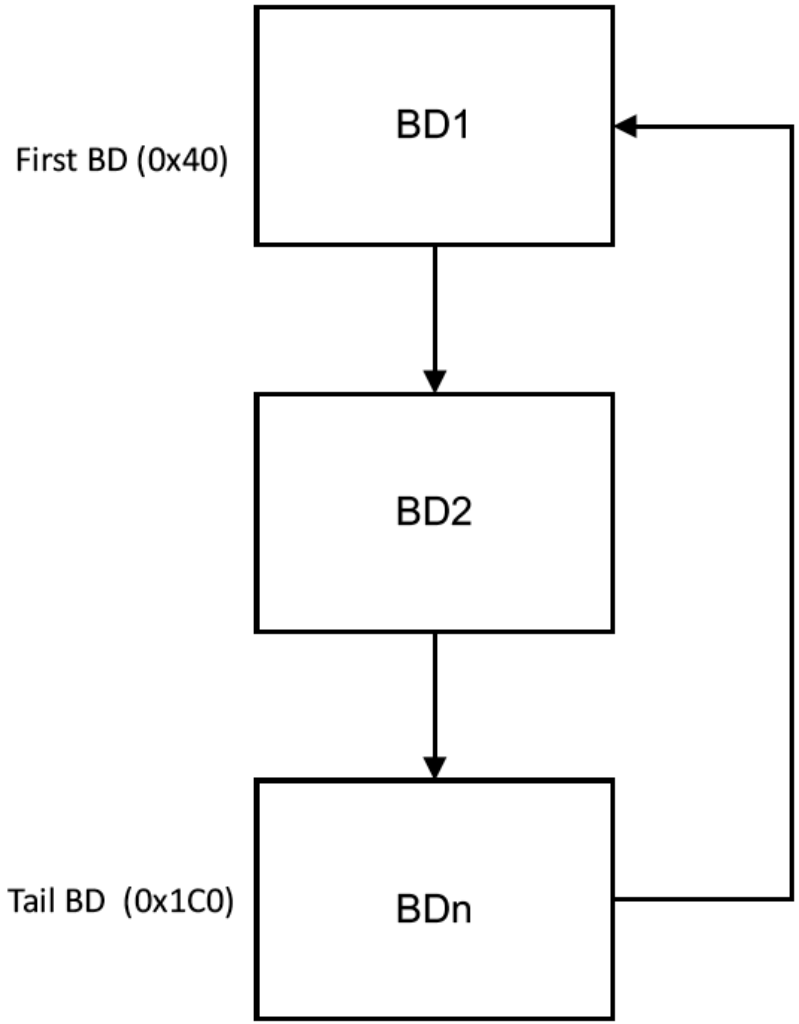
   All other S2MM registers can be written in any order.

# Scatter Gather DMA Mode




Source: Symbian OS Internals/13. Peripheral Support

# Chain of Buffer Descriptors (BDs)



# Scatter Gather DMA Transfer
## Programming Sequence for MM2S channel (1)

1. **Write the address of the starting descriptor** to **the Current Descriptor register**
2. **Start the MM2S channel running** by setting the run/stop bit to 1, MM2S_DMACR.RS = 1.
3. If desired, **enable interrupts** by writing a 1 to MM2S_DMACR.IOC_IrqEn and MM2S_DMACR.Err_IrqEn.
4. **Write a valid address to the Tail Descriptor register**.

Writing to the Tail Descriptor register triggers the DMA to start fetching the descriptors from the memory.

# Simple DMA Transfer
## Programming Sequence for S2MM channel (1)

1. **Write the address of the starting descriptor** to **the Current Descriptor register**
2. **Start the S2MM channel running** by setting the run/stop bit to 1, S2MM_DMACR.RS = 1.
3. If desired, **enable interrupts** by by writing a 1 to S2MM_DMACR.IOC_IrqEn and S2MM_DMACR.Err_IrqEn.
4. **Write a valid address to the Tail Descriptor register**.

Writing to the Tail Descriptor register triggers the DMA to start fetching the descriptors from the memory.