ECE5917
SoC Architecture: ARM/AMBA

Tae Hee Han: than@skku.edu

Semiconductor Systems Engineering

Sungkyunkwan University

Outline

- ARM Processor Architecture

- AMBA Bus

What Is ARM?

- Advanced RISC Machine

Reduced Instruction Set Computer

- Founded in November 1990
  - Spun out of Acorn Computers – Joint venture of Acorn computer, Apple computer and VLSI technology
- Designs the ARM range of RISC processor cores
- Licenses ARM core designs to semiconductor partners who fabricate and sell to their customers
  - ARM does not fabricate silicon itself
- Also develop technologies to assist with the design-in of the ARM architecture
  - Software tools, boards, debug hardware, application software, bus architectures, peripherals etc.
- Market-leader for low-power and cost-sensitive embedded applications
  - Architectural simplicity which allows very small implementations which result in very low power consumption

The History of ARM

- Developed at Acorn Computers Limited, of Cambridge, England, between 1983 and 1985
- Problems with CISC:
  - Slower then memory parts
    - Due to memory operand and instruction decoding
  - Variable clock cycles per instruction
- Solution – the Berkeley RISC I:
  - Competitive
  - Easy to develop (less than a year)
  - Cheap
  - Pointing the way to the future

ARM AArch32

Evolution of the AArch32 (1)

- Original ARM architecture:
  - 32-bit RISC architecture focused on core instruction set
  - 16 Registers - 1 being the Program counter – generally accessible
  - Conditional execution on all instructions
  - Load/Store Multiple operations - Good for Code Density
  - Shifts available on data processing and address generation
  - Original architecture had 26-bit address space
    - Augmented by a 32-bit address space early in the evolution
- Thumb instruction set was the next big step
  - ARMv4T architecture (ARM7TDMI)
  - Introduced a 16-bit instruction set alongside the 32-bit instruction set
    - Different execution states for different instruction sets
    - Switching ISA as part of a branch or exception
    - Not a full instruction set – ARM still essential

Evolution of the AArch32 (2)

- ARMv5TEJ (ARM926EJ-S) introduced:
  - Better interworking between ARM and Thumb
    - Bottom bit of the address used to determine the ISA
  - DSP-focused additional instructions
  - Jazelle-DBX for Java byte code interpretation in hardware
  - Some architecting of the virtual memory system
- ARMv6K (ARM1136JF-S) introduced:
  - Media processing – SIMD within the integer datapath
  - Enhanced exception handling
  - Overhaul of the memory system architecture to be fully architected
    - Supported only 1 level of cache
- ARMv7 rolled in a number of substantive changes:
  - Thumb-2* - variable length instruction set
  - TrustZone*
  - Jazelle-RCT
  - Neon

AArch32 Architecture (1)

- Typical RISC architecture:
  - Large uniform register file
  - Load/store architecture
  - Simple addressing modes
  - Uniform and fixed-length instruction fields

AArch32 Architecture (2)

- Enhancements:
  - Each instruction controls the ALU and shifter
  - Auto-increment and auto-decrement addressing modes
  - Multiple Load/Store
  - Conditional execution
- Results:
  - High performance
  - Low code size
  - Low power consumption
  - Low silicon area

Processor Modes (1)

- The ARM has **seven** basic processor modes:
  - User : unprivileged mode under which most tasks run
  - Privileged:
    - System : privileged mode using the same registers as user mode
    - FIQ : entered when a high priority (fast) interrupt is raised
    - IRQ : entered when a low priority (normal) interrupt is raised
    - Supervisor : entered on reset and when a Software Interrupt instruction is executed
    - Abort : used to handle memory access violations
    - Undef : used to handle undefined instructions

  Exception modes

- Mode changes can be made under software control, or can be caused by external interrupts or exception processing

Processor Modes (2)

- User mode:
  - Normal program execution mode
  - System resources unavailable
  - Mode changed by exception only
- Exception modes:
  - Entered upon exception
  - Full access to system resources
  - Mode changed freely

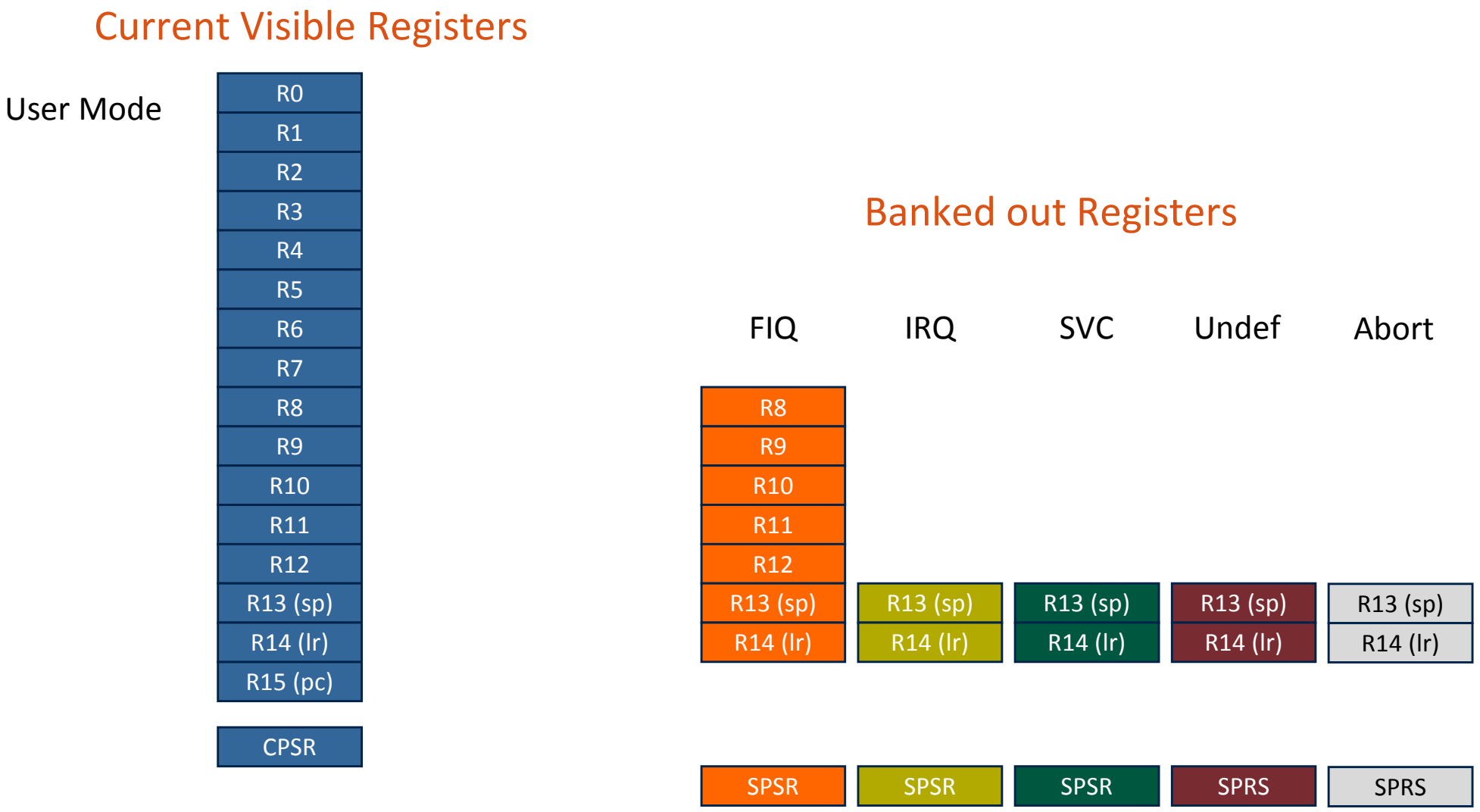
AArch32 Registers (1)

- ARM has 37 registers all of which are 32-bits long.
  - 31 general purpose registers including a program counter (PC)
    - At any one time, 16 visible (R0 - R15), Others speed up the exception process
  - 6 status registers
    - 1 dedicated current program status register (CPSR)
    - 5 dedicated saved program status registers (SPSR)
- The current processor mode governs which of several banks is accessible. Each mode can access
  - a particular set of R0-R12 registers
  - a particular R13 (the stack pointer, sp) and R14 (the link register, lr)
  - the program counter, R15 (pc)
  - the current program status register, CPSR
- Privileged modes (except System) can also access
  - a particular SPSR (saved program status register)

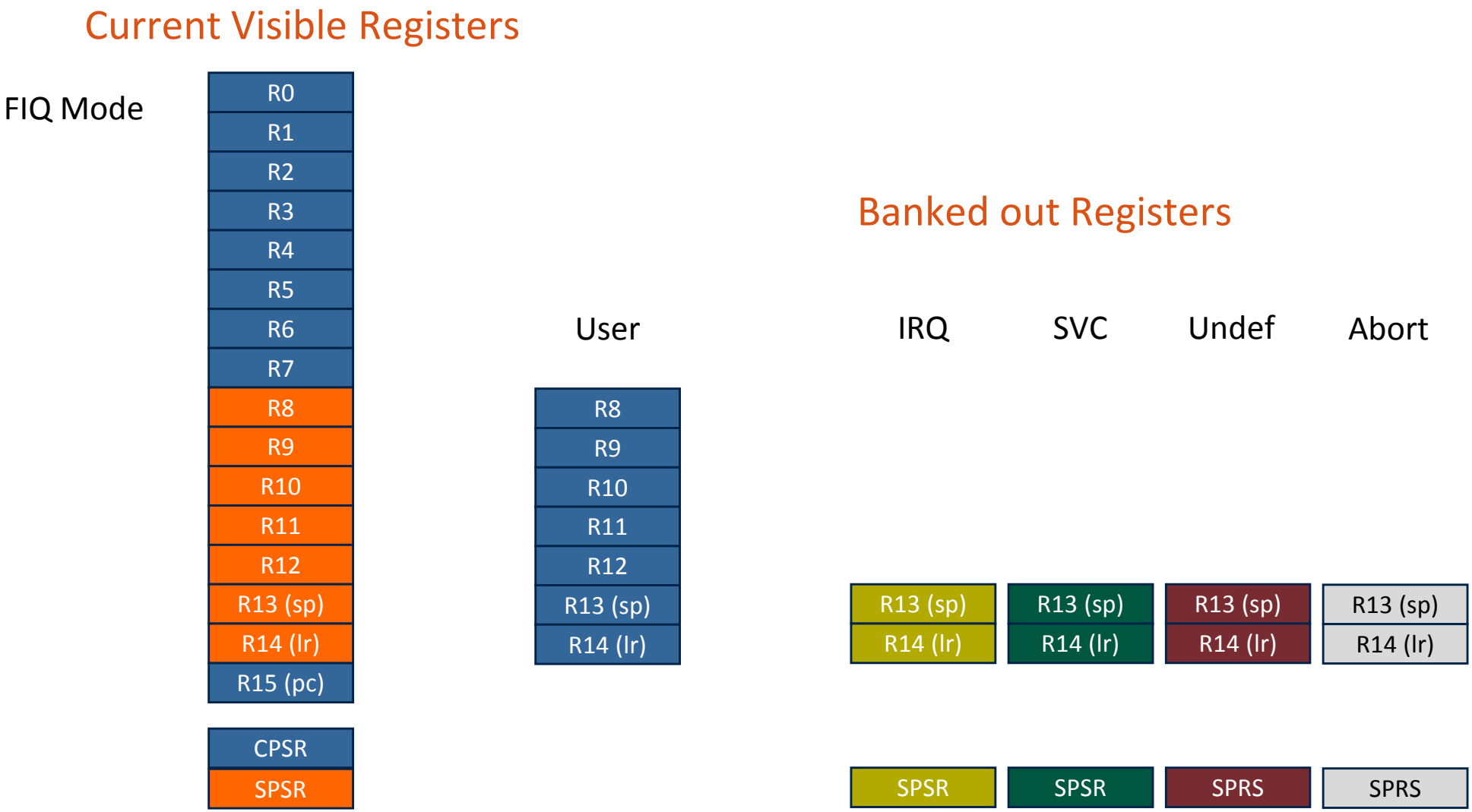
AArch32 Registers (2)

- General-purpose registers
  - The GP registers R0 – R15 can be split into three groups, which are differentiated in the way they are banked and in their special-purpose uses:
    - The unbanked registers, R0 – R7
      - Each of them refers to the same 32-bit physical register in all processor modes
      - They are completely GP registers, with no special uses implied
    - The banked registers, R8 – R14
      - The physical register referred to by each of them depends on the current processor mode
      - Where a particular physical register is intended, w/o depending on the current processor mode, a more specific name is used
    - Register 15, R15 (Program Counter)

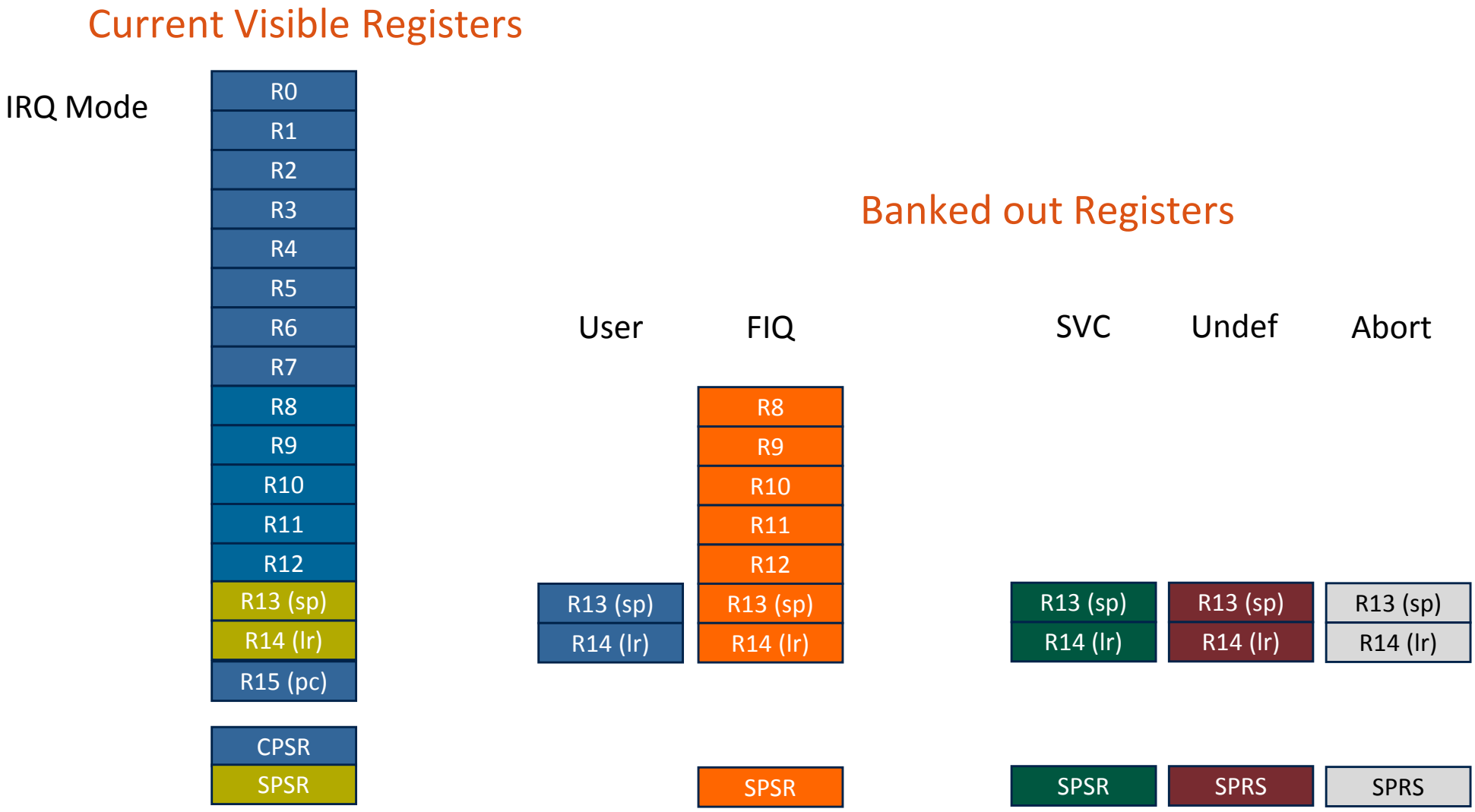
AArch32 Registers (3-1)



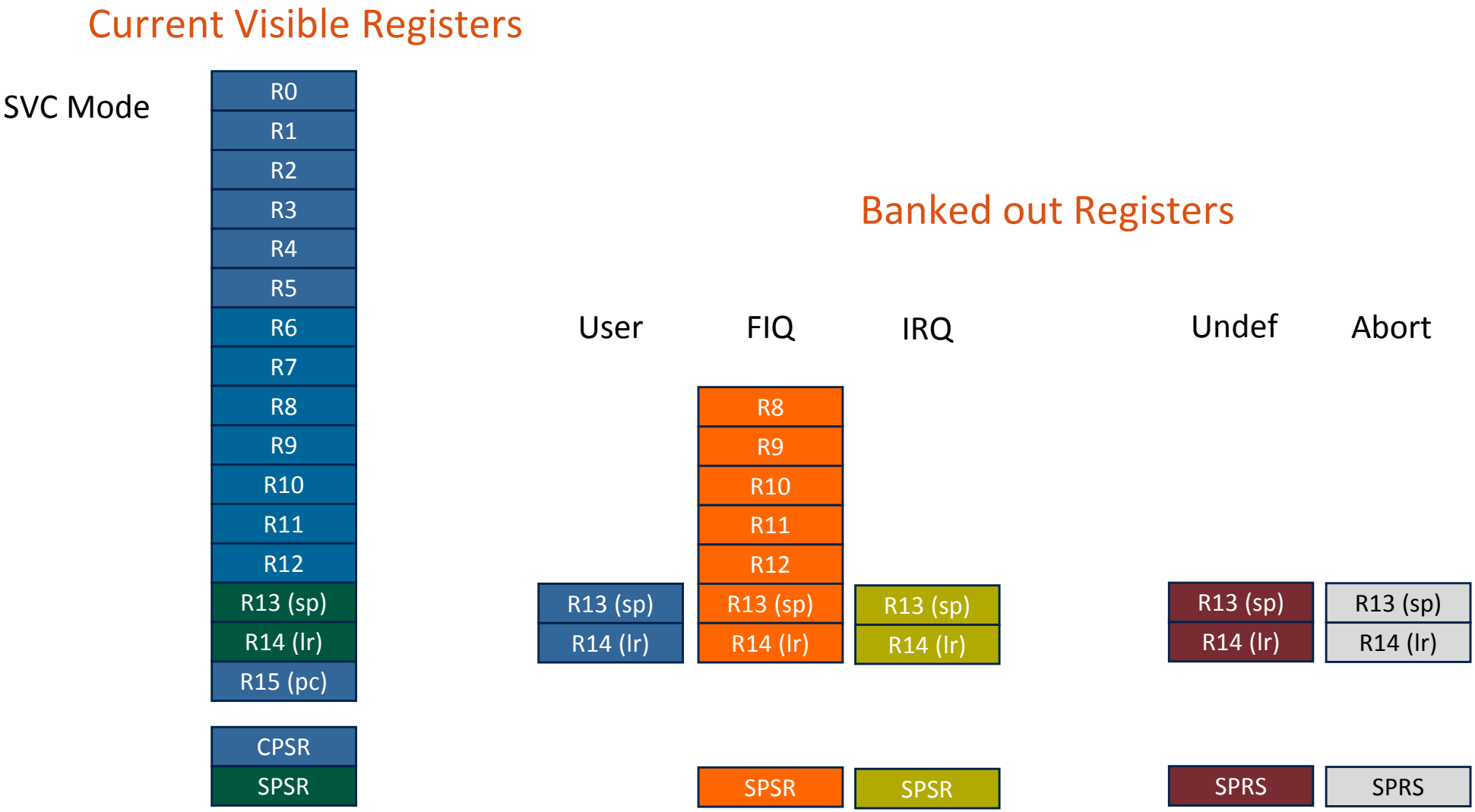
AArch32 Registers (3-2)



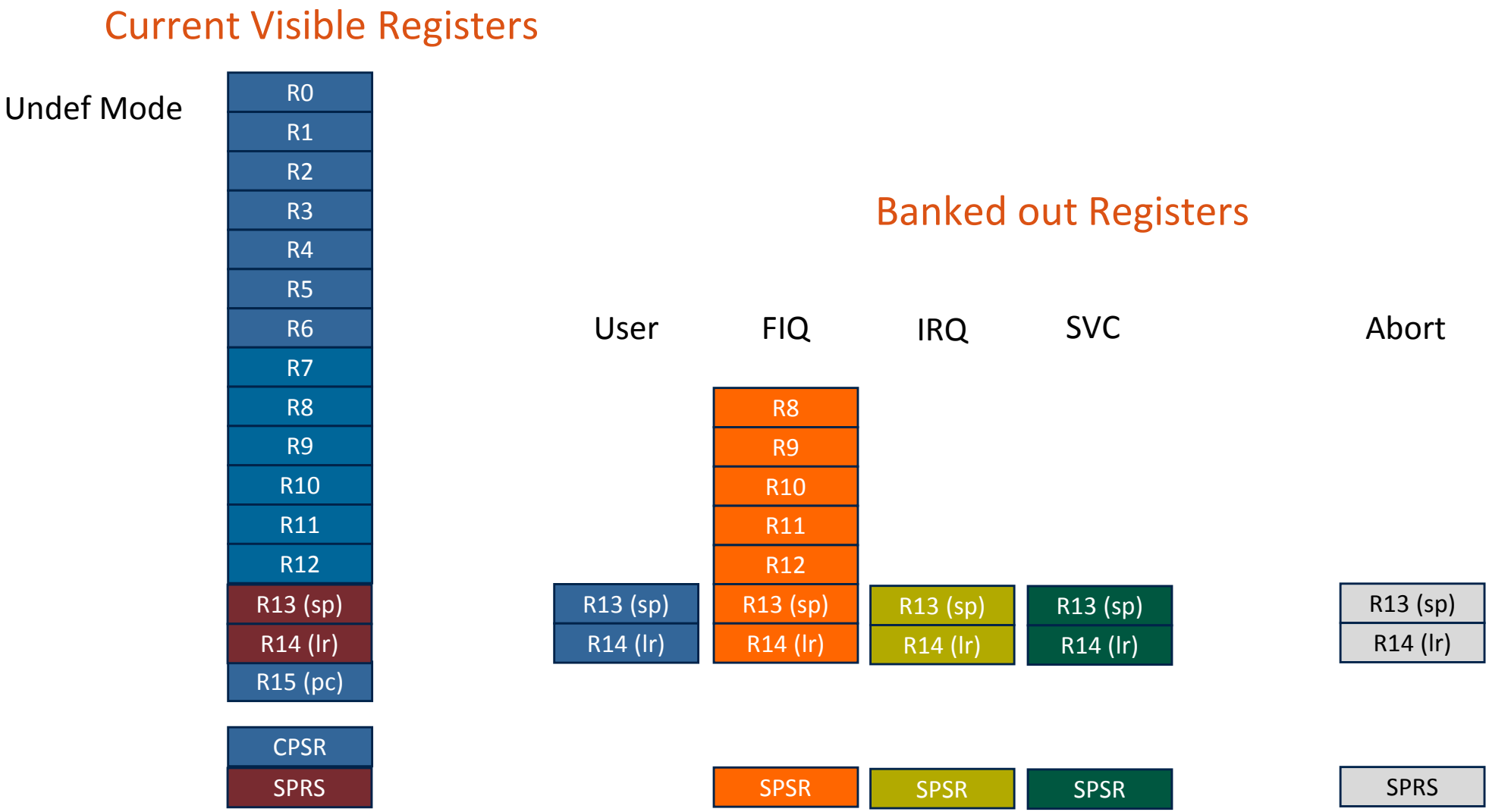
AArch32 Registers (3-3)



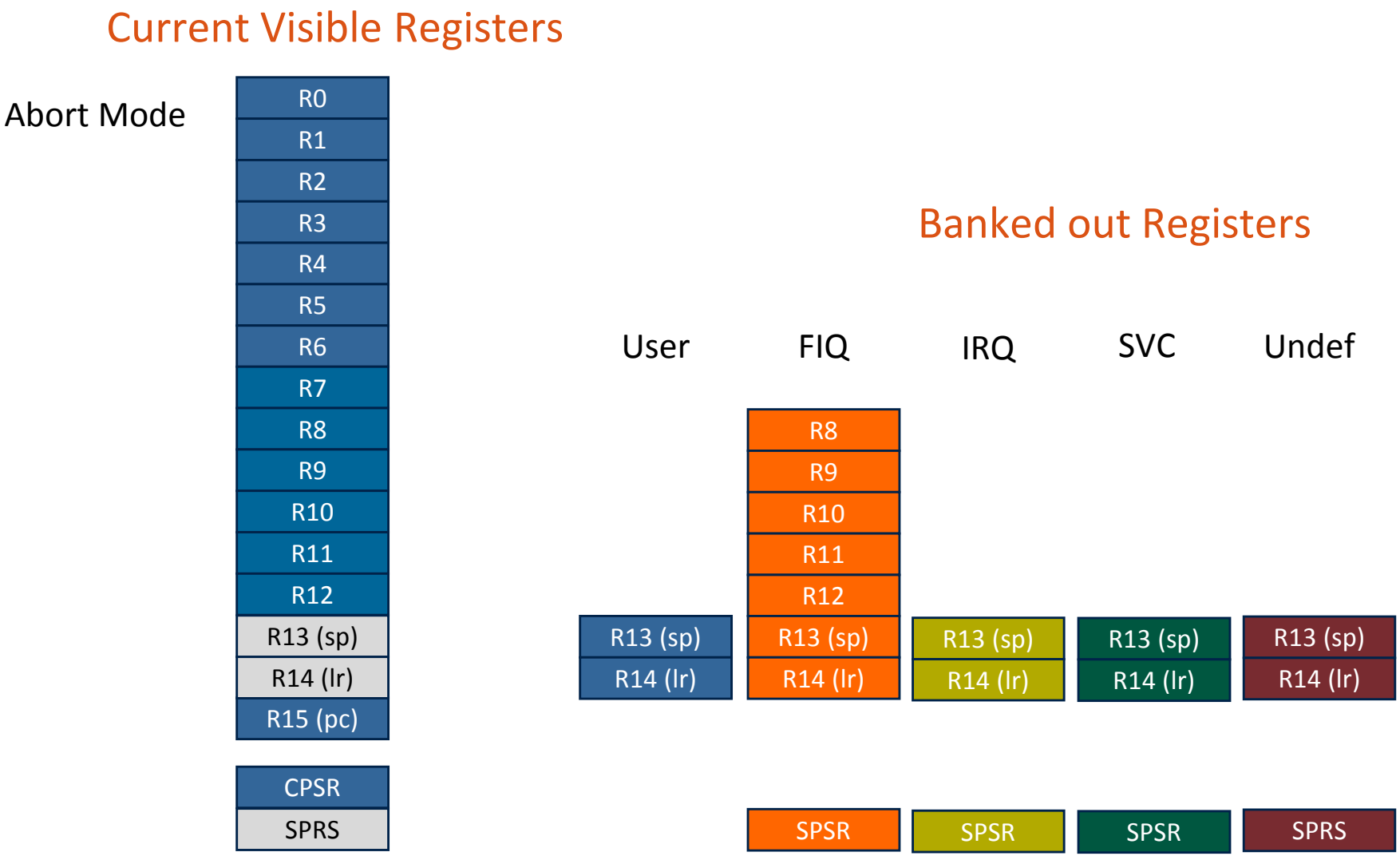
AArch32 Registers (3-4)



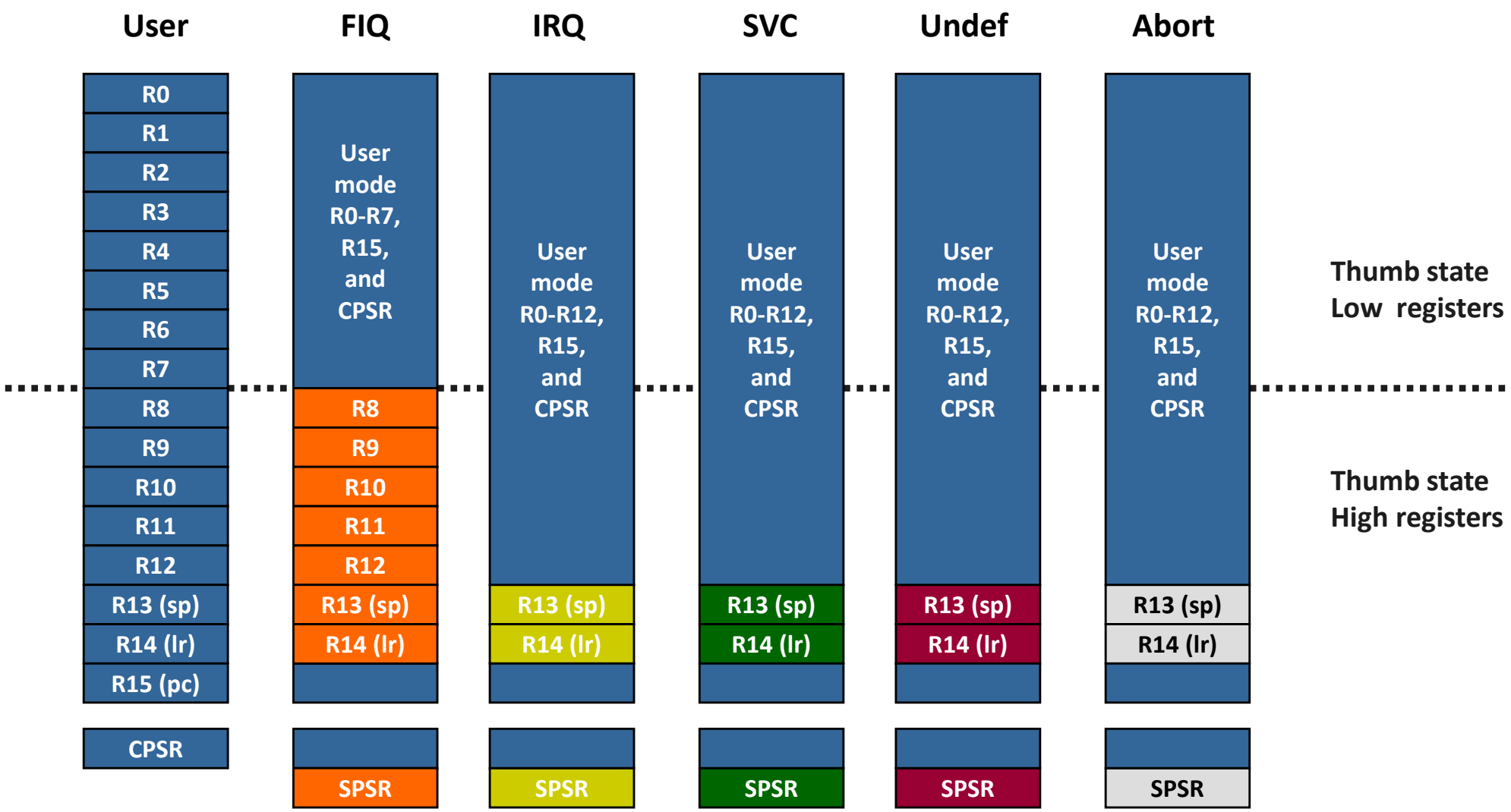
AArch32 Registers (3-5)



AArch32 Registers (3-6)

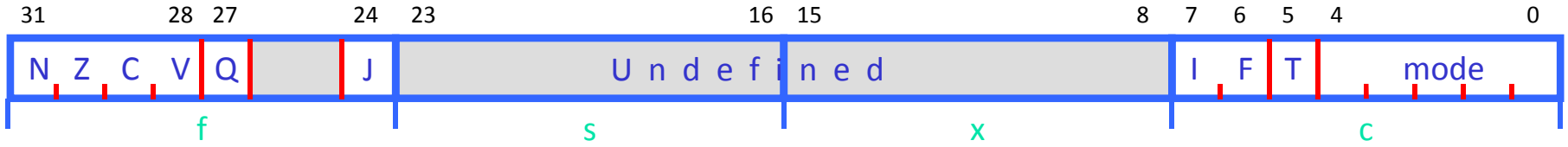


AArch32 Register Organization Summary



Note: System mode uses the User mode register set

Program Status Register



- Condition code flags
  - N = Negative result from ALU
  - Z = Zero result from ALU
  - C = ALU operation Carried out
  - V = ALU operation oVerflow
- Sticky Overflow flag - Q flag
  - Architecture 5TE/J only
  - Indicates if saturation has occurred
- J bit
  - Architecture 5TEJ only
  - J = 1: Processor in Jazelle state
- Interrupt Disable bits.
  - I = 1: Disables the IRQ.
  - F = 1: Disables the FIQ.
- T Bit
  - Architecture xT only
  - T = 0: Processor in ARM state
  - T = 1: Processor in Thumb state
- Mode bits
  - Specify the processor mode

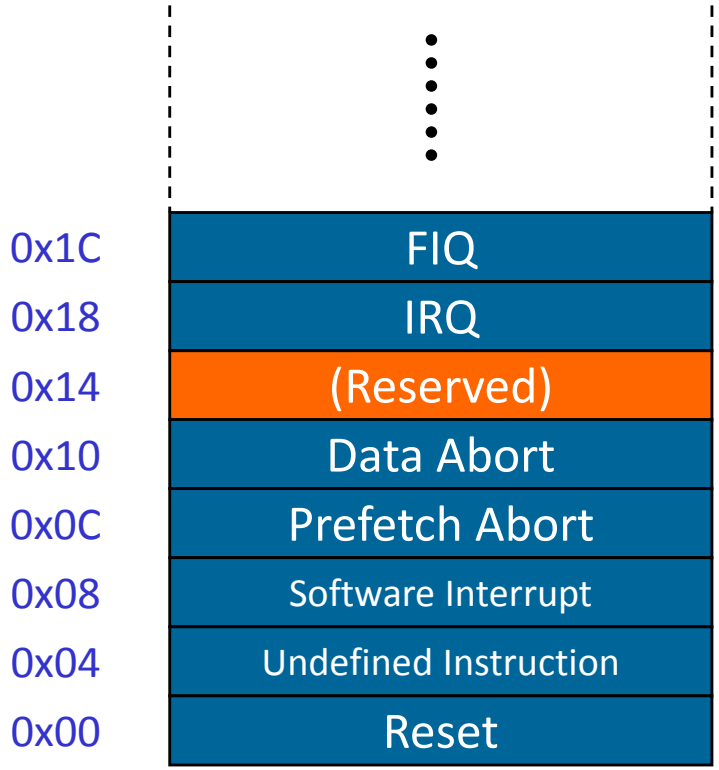
Program Counter (R15)

- When the processor is executing in ARM state:
  - All instructions are 32 bits wide
  - All instructions must be word aligned
  - Therefore the pc value is stored in bits [31:2] with bits [1:0] undefined (as instruction cannot be halfword or byte aligned)
- When the processor is executing in Thumb state:
  - All instructions are 16 bits wide
  - All instructions must be halfword aligned
  - Therefore the pc value is stored in bits [31:1] with bit [0] undefined (as instruction cannot be byte aligned)
- When the processor is executing in Jazelle state:
  - All instructions are 8 bits wide
  - Processor performs a word access to read 4 instructions at once

Exception Handling

- When an exception occurs, the ARM:
  - Copies CPSR into SPSR_<mode>
  - Sets appropriate CPSR bits
    - Change to ARM state
    - Change to exception mode
    - Disable interrupts (if appropriate)
  - Stores the return address in LR_<mode>
  - Sets PC to vector address
- To return, exception handler needs to:
  - Restore CPSR from SPSR_<mode>
  - Restore PC from LR_<mode>

This can only be done in ARM state



Vector Table

Vector table can be at
0xFFFF0000 on ARM720T
and on ARM9/10 family devices

Data Sizes and Instruction Set

- The AArch32 is a 32-bit architecture
- When used in relation to the AArch32:
  - **Byte** means 8 bits
  - **Halfword** means 16 bits (two bytes)
  - **Word** means 32 bits (four bytes)
- Most AArch32 implement two instruction sets
  - 32-bit ARM Instruction Set: Standard 32-bit instruction set
  - 16-bit Thumb Instruction Set
    - 16-bit compressed form
    - Code density better than most CISC
    - Dynamic decompression in pipeline

AArch32 Instruction Set Basic Features

- Load/Store architecture
- 3-address data processing instructions
- Conditional execution
- Load/Store multiple registers
- Shift & ALU operation in single clock cycle

Conditional Execution and Flags

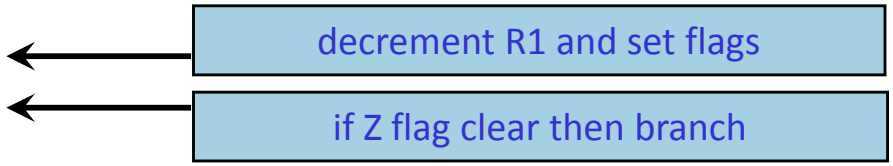
- AArch32 instructions can be made to execute conditionally by postfixing them with the appropriate condition code field
  - This improves code density *and* performance by reducing the number of forward branch instructions

```
CMP   R3,#0
BEQ   skip
ADD   R0,R1,R2
skip
```

```
CMP   R3,#0
ADDNE R0,R1,R2
```

- By default, data processing instructions do not affect the condition code flags but the flags can be optionally set by using “S”. CMP does not need “S”.

```
loop
  …
  SUBS R1,R1,#1
  BNE loop
```



Condition Codes

- The possible condition codes are listed below:
  - Note AL is the default and does not need to be specified

Suffix	Description	Flags tested
EQ	Equal	Z=1
NE	Not equal	Z=0
CS/HS	Unsigned higher or same	C=1
CC/LO	Unsigned lower	C=0
MI	Minus	N=1
PL	Positive or Zero	N=0
VS	Overflow	V=1
VC	No overflow	V=0
HI	Unsigned higher	C=1 & Z=0
LS	Unsigned lower or same	C=0 or Z=1
GE	Greater or equal	N=V
LT	Less than	N!=V
GT	Greater than	Z=0 & N=V
LE	Less than or equal	Z=1 or N=!V
AL	Always	

Examples of Conditional Execution

- Use a sequence of several conditional instructions

```
if (a==0) func(1);
    CMP     R0,#0
    MOVEQ   R0,#1
    BLEQ    func
```

- Set the flags, then use various condition codes

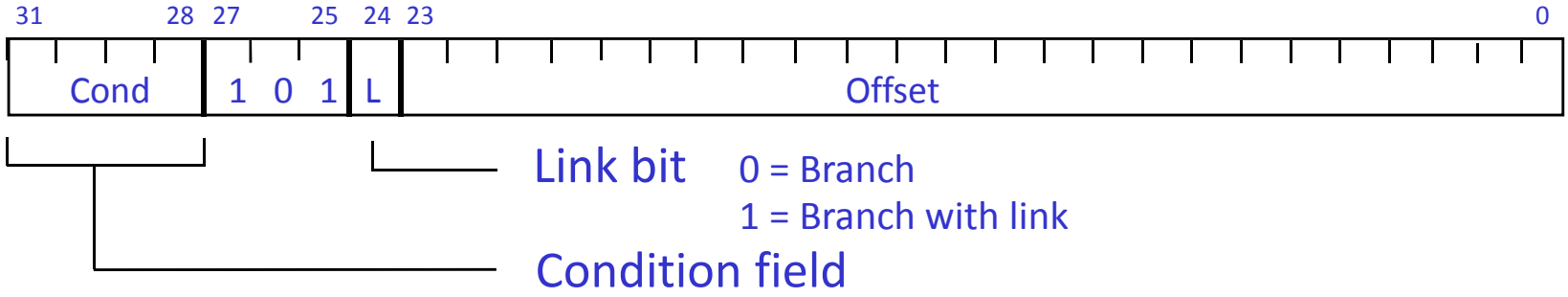
```
if (a==0) x=0;
if (a>0)  x=1;
    CMP     R0,#0
    MOVEQ   R1,#0
    MOVGT   R1,#1
```

- Use conditional compare instructions

```
if (a==4 || a==10) x=0;
    CMP     R0,#4
    CMPNE   R0,#10
    MOVEQ   R1,#0
```

Branch Instructions

- Branch : `B{<cond>} label`
- Branch with Link : `BL{<cond>} subroutine_label`



- The processor core shifts the offset field left by 2 positions, sign-extends it and adds it to the PC
  - ± 32 Mbyte range
  - How to perform longer branches?

Can set up LR manually if needed, then load into PC

MOV lr, pc

LDR pc, =dest

ADS linker will automatically generate long branch veneers for branches beyond 32MB range.

Data Processing Instructions

- Consist of :
  - Arithmetic: `ADD` `ADC` `SUB` `SBC` `RSB` `RSC`
  - Logical: `AND` `ORR` `EOR` `BIC`
  - Comparisons: `CMP` `CMN` `TST` `TEQ`
  - Data movement: `MOV` `MVN`
- These instructions only work on registers, NOT memory
- Syntax:

```
<Operation>{<cond>}{S} Rd, Rn, Operand2
```

  - Comparisons set flags only - they do not specify Rd
  - Data movement does not specify Rn
- Second operand is sent to the ALU via barrel shifter

The Barrel Shifter

LSL : Logical Left Shift

Multiplication by a power of 2

ASR: Arithmetic Right Shift

Division by a power of 2, preserving the sign bit

LSR : Logical Shift Right

Division by a power of 2

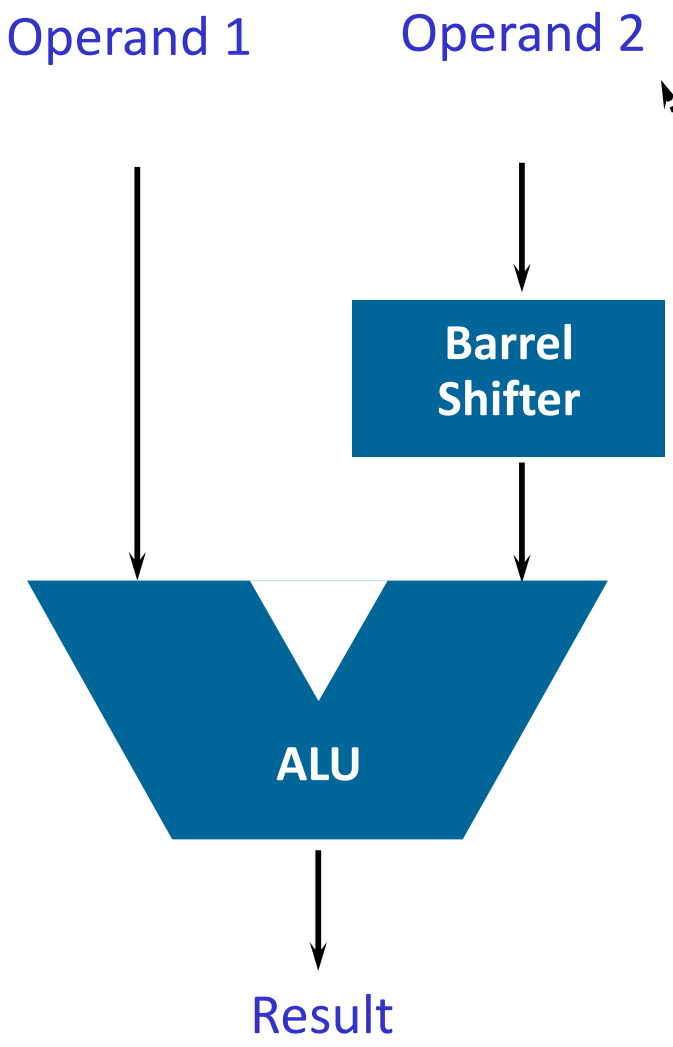
ROR: Rotate Right

Bit rotate with wrap around from LSB to MSB

RRX: Rotate Right Extended

Single bit rotate with wrap around from CF to MSB

Using the Barrel Shifter: The Second Operand



- Register, optionally with shift operation
  - Shift value can be either be:
    - 5 bit unsigned integer
    - Specified in bottom byte of another register
  - Used for multiplication by constant
- Immediate value
  - 8 bit number, with a range of 0-255.
    - Rotated right through even number of positions
  - Allows increased range of 32-bit constants to be loaded directly into registers

Data Processing Instructions

Conditional codes

+

Data processing instructions

+

Barrel shifter

=

Powerful tools for efficient coded programs

Data Processing Instructions

e.g.:

if (z==1) R1=R2+(R3*4)

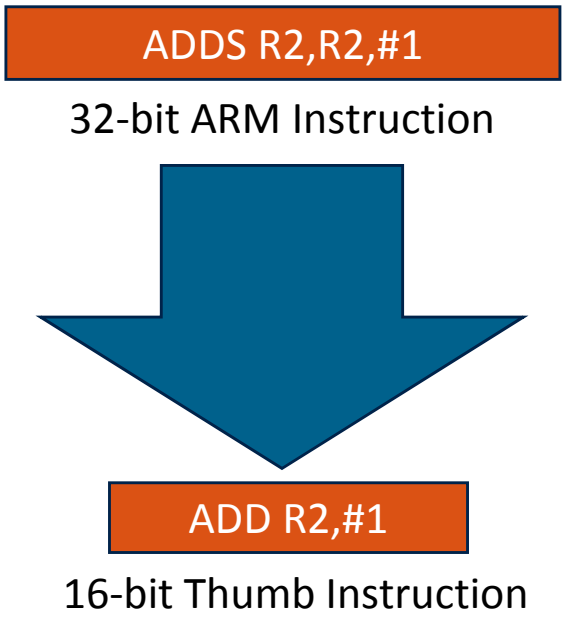
compiles to

EQADDS R1,R2,R3, LSL #2

(SINGLE INSTRUCTION !)

THUMB Instruction Set

- Thumb is a 16-bit instruction set
  - Optimized for code density from C code (~65% of ARM code size)
  - Improved performance from narrow memory
  - Subset of the functionality of the ARM instruction set
- Core has additional execution state - Thumb
  - Switch between ARM and Thumb using BX instruction

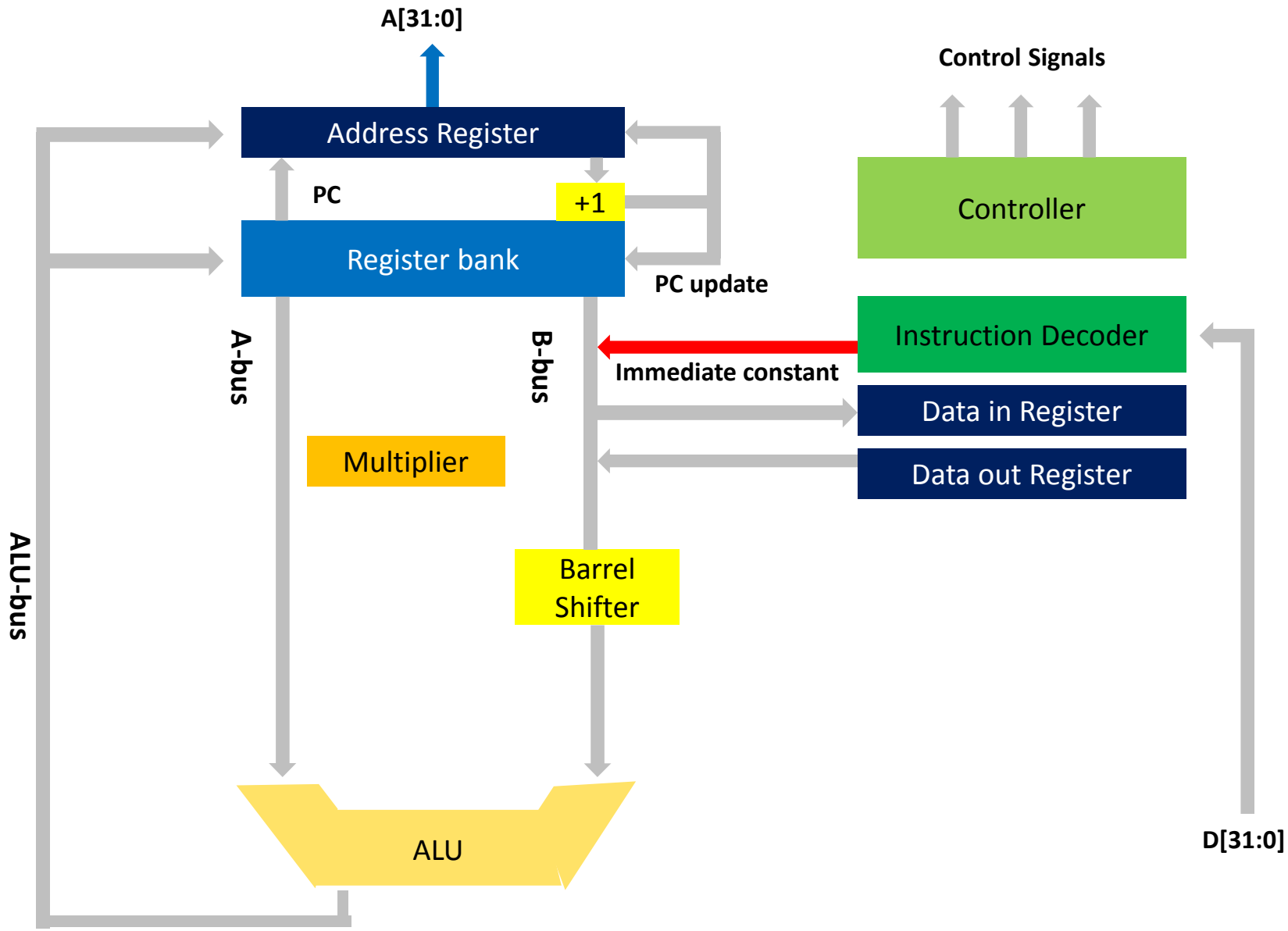


- For most instructions generated by compiler:
  - Conditional execution is not used
  - Source and destination registers identical
  - Only Low registers used
  - Constants are of limited size
  - Inline barrel shifter not used

ARM7 Family

- 32/16-bit RISC Architecture
- Unified bus architecture
  - Both instructions and data use the same bus
- 3 stage pipelining
  - Fetch / decode / execution
- Coprocessor interface
- Embedded ICE-RT support
- JTAG interface unit
- Optional support for MMU

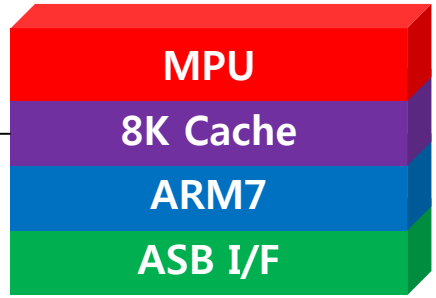
ARM7 Block Diagram

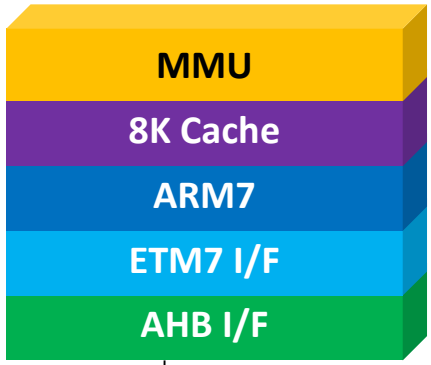


ARM7 Family

ARM740T

- Embedded RTOS core
- Hard Macro IP





ARM720T

- Platform OS core
- 100MHz (0.13um)
- 0.20mW/MHz
- Hard Macro IP

ARM7-S

ARM7TDMI-S

- Base Integer core
- 80-110MHz (0.18um)
- 0.28mW/MHz
- Synthesizable (soft IP)

ARM7

ARM7TDMI

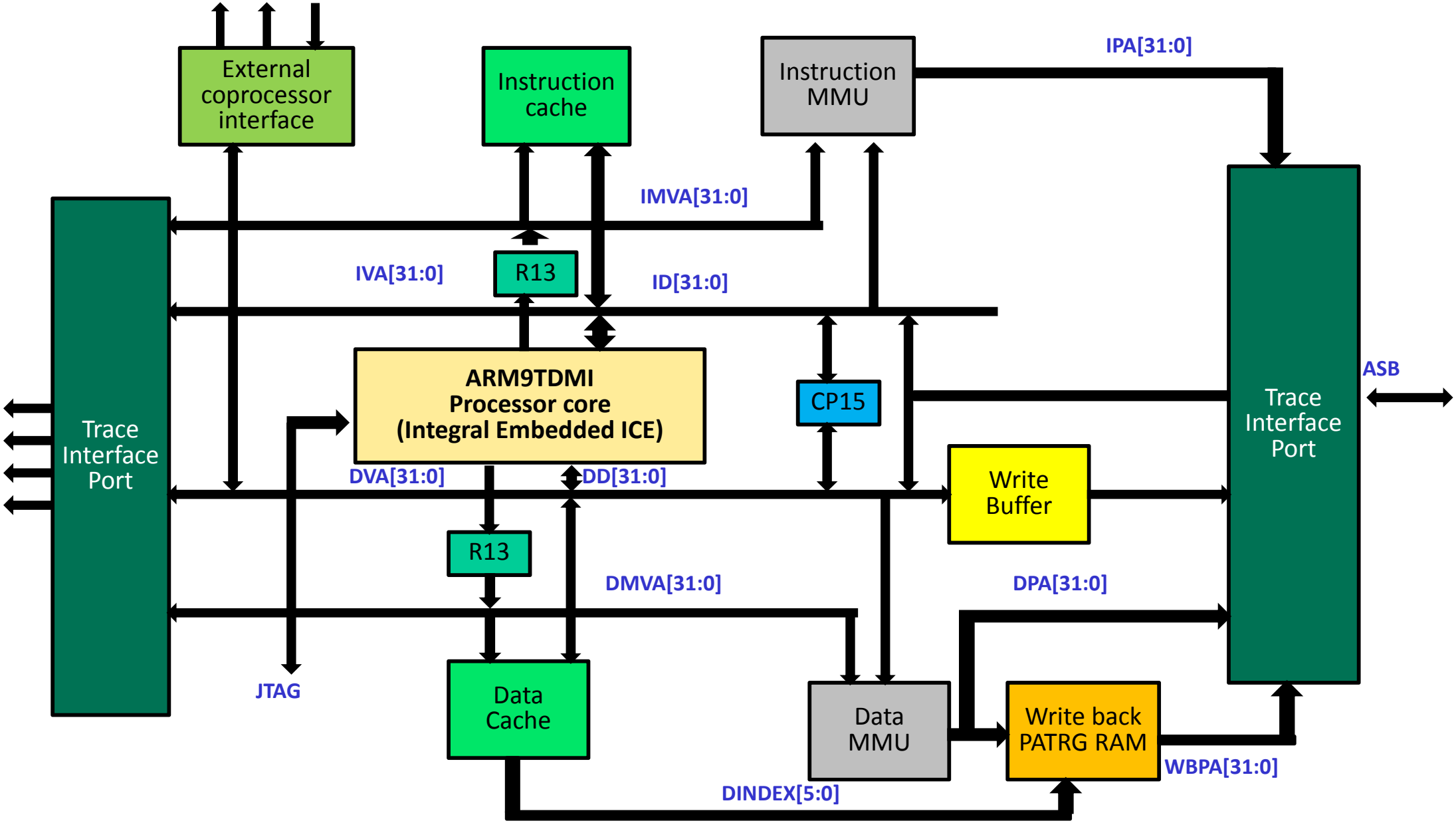
- Base Integer core(Hard Macro IP)
- 100MIPS (0.18um)
- 0.25mW/MHz
- NEW : low-voltage layout

- ARM v4T ISA
- Von Neumann Architecture
- Thumb Support
- Debug Support
- 3-Stage Pipeline

ARM9 Family

- 32/16-bit RISC Architecture
- Harvard Architecture
  - Separate memory bus architecture
- 5 stage pipelining
  - Fetch/Decode/Execute/Memory/Write
- Coprocessor interface
- EmbeddedICE-RT support
- JTAG interface unit
- Embedded Trace Macrocell

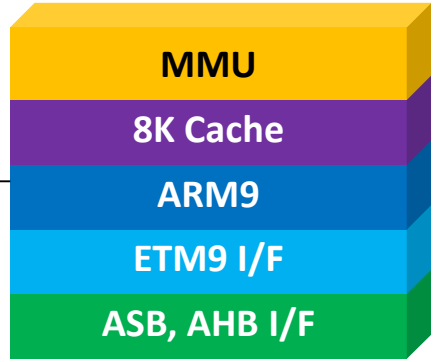
ARM920T Block Diagram

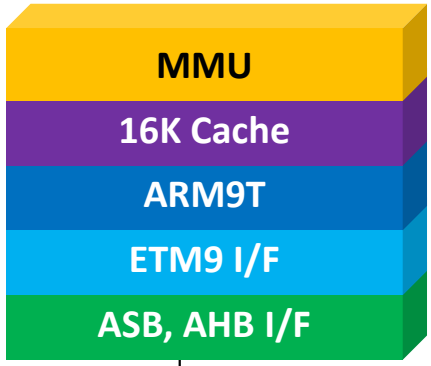


ARM9 Family

ARM922T

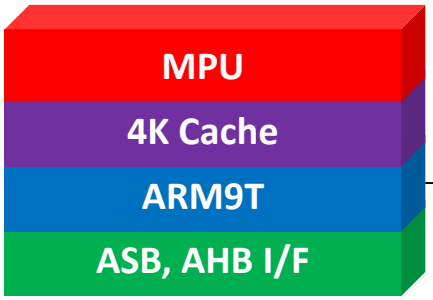
- Platform OS processor core
- Up to 200MIPS (0.18um)
- 0.8mW/MHz





ARM920T

- Platform OS processor core
- Up to 200MIPS (typical)
- 180MHz (typical)



ARM940T

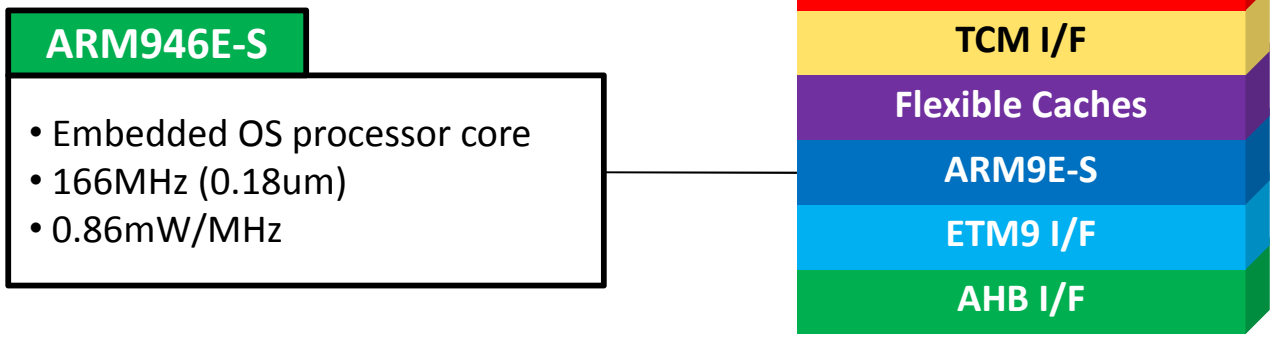
- Embedded RTOS processor core

ARM9TDMI

- Base Integer processor core

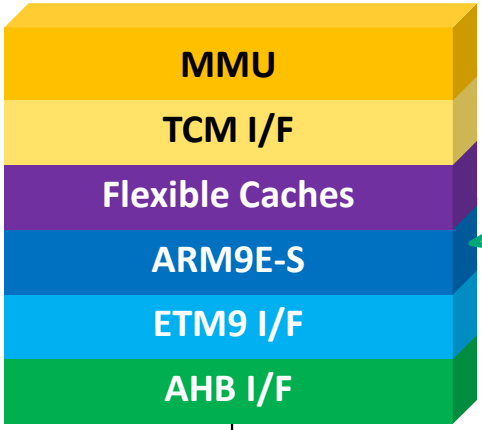
- ARM v4T ISA
- Harvard architecture
- Thumb Support
- Debug Support
- 5-Stage Pipeline
- Dual caches
- AMBA ASB, AHB bus

ARM9E Family



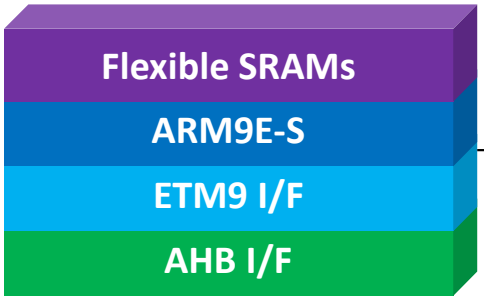
ARM946E-S

- Embedded OS processor core
- 166MHz (0.18um)
- 0.86mW/MHz



ARM926EJ-S

- Platform OS processor core
- 200MIPS
- 220MHz (0.18um)



ARM966E-S

- Embedded RTOS processor core

ARM968E-S

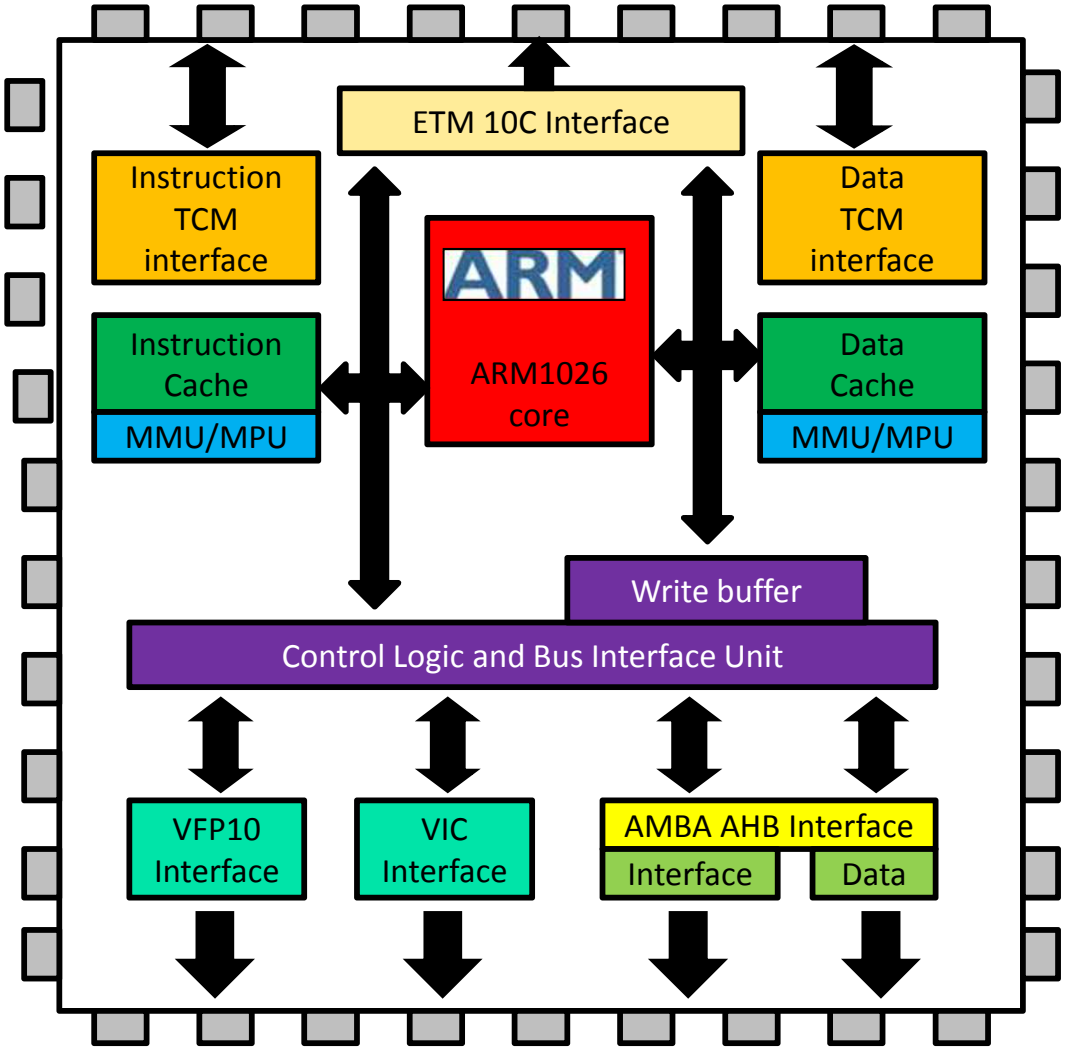
- Low Power Embedded core
- 270MHz (0.13um)
- 0.14mW/MHz

- Synthesizable (Soft IP)
- ARM v5TE ISA
- Thumb Support
- Debug Support
- 5-Stage Pipeline
- Dual Caches
- AMBA AHB bus

ARM10 Family

- 6 stage pipeline
- Parallel load / store unit
  - Allow computation to continue while data transfers complete
- 64-bit data bus
- Optional support for vector floating-point
  - 7th stage in pipeline
  - Increase FP performance
  - Out-of-order completion

ARM1026EJ-S block diagram

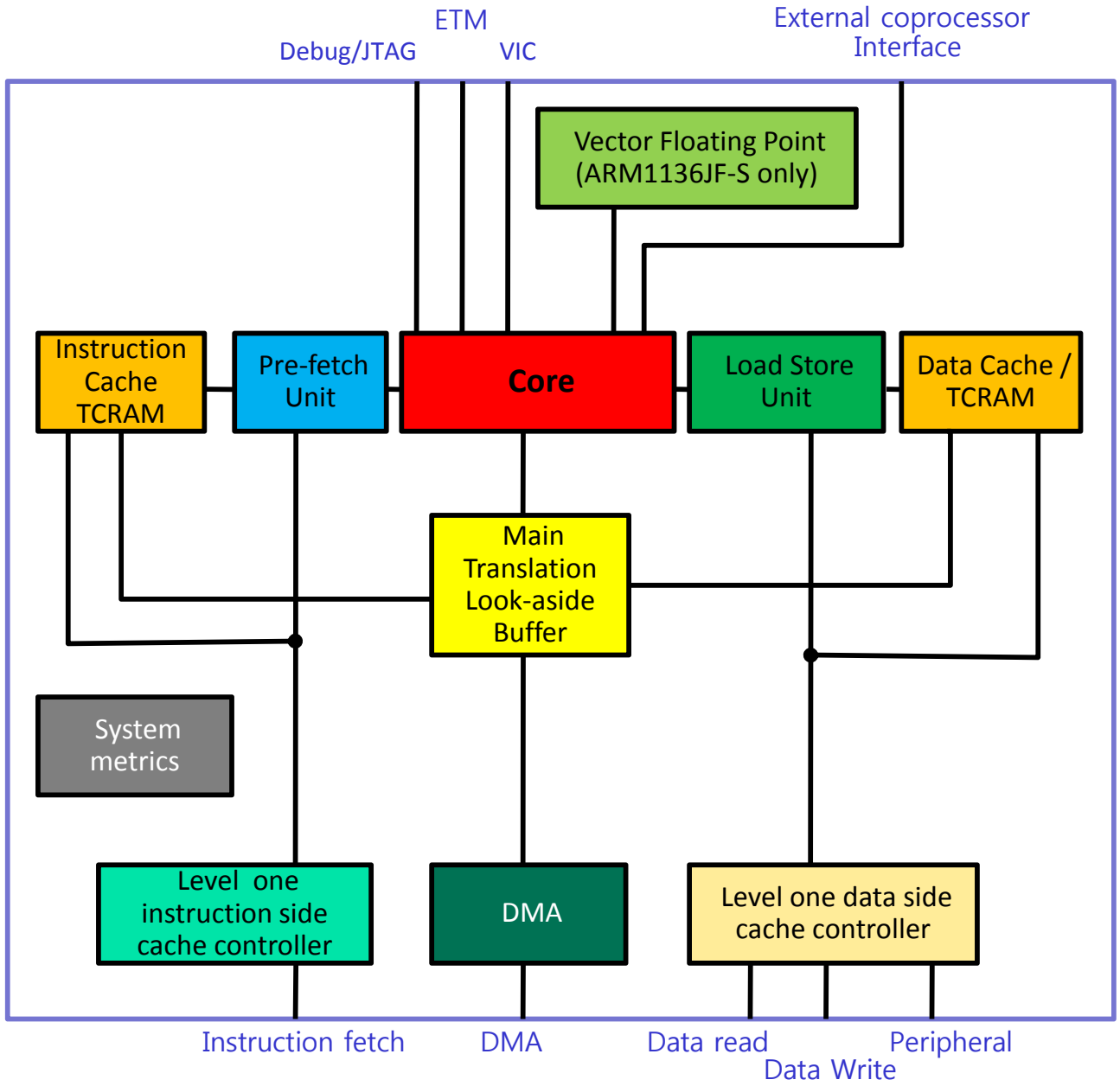


- ARM10EJ-S integer core in an ARMv5TEJ implementation
  - 32-bit ARM
  - 16-bit Thumb
  - 8-bit Jazelle instruction set
- MMU
  - Single TLB for both instruction and data
- Memory Protection Unit (MPU)
  - Partition external memory into 8-protection regions
- Cache (I/D)
  - Configurable to 0KB or 4-128KB
- Tightly Coupled Memory (TCM)
  - Configurable to 0KB or 4KB-1MB

ARM11 Family

- 8 stage pipeline
  - load-store pipeline
  - Arithmetic pipeline
- Parallel load / store unit
  - Allow computation to continue while data transfers complete
  - Non-blocking cache
- 64-bit data bus
- Out-of-order completion

ARM1136JF-S block diagram

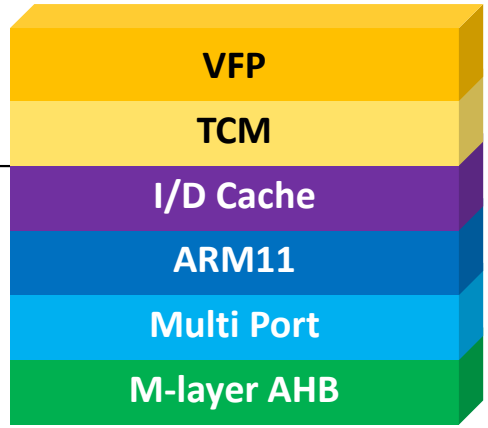


- ARM11 Core in an ARMv6 implementation
  - 32-bit ARM
  - 16-bit Thumb
  - 8-bit Jazelle instruction set
- Load Store Unit(LSU)
  - Manages all load and store operations
  - Load-Store pileline
    - Decouples loads and stores from the MAC and ALU pipelines
- Prefetch Unit
- Memory system
  - Harvard architecture
  - 64-bit datapaths
  - 2-channel DMA into TCM

ARM11 Family

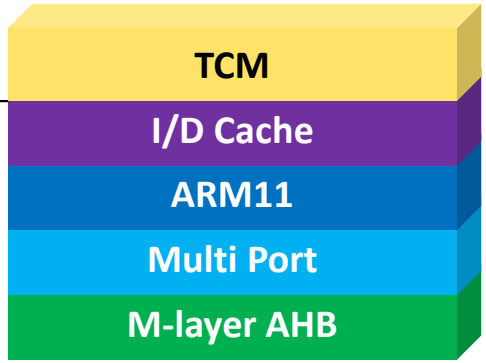
ARM1136JF-S

- Platform OS core
- 620MHz (90nm)
- 0.45mW/MHz
- Floating Point coprocessor



ARM1136J-S

- Platform OS core
- 620MHz (90nm)
- 0.45mW/MHz



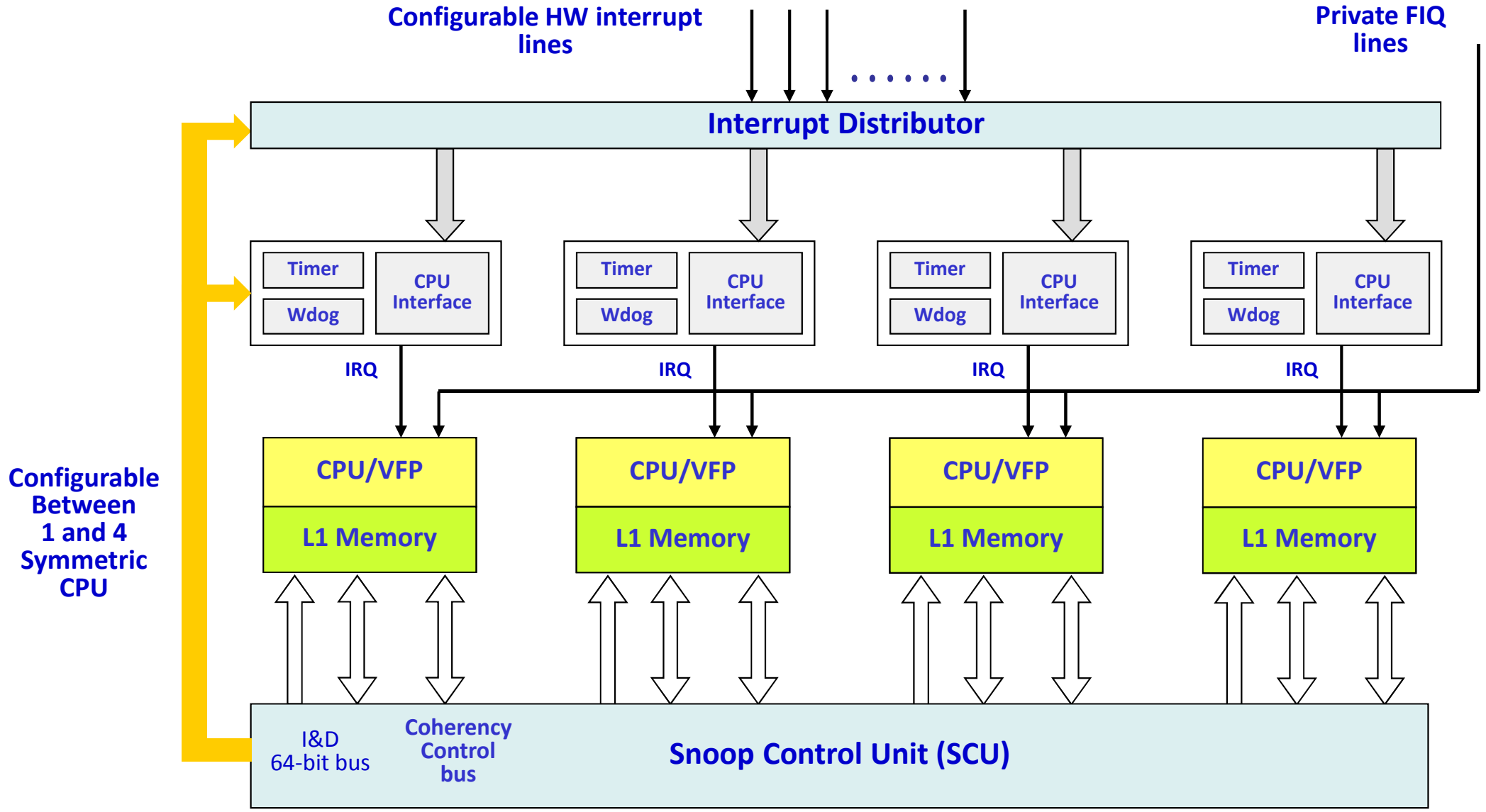
ARM11

ARM11

- Platform OS core

- ARM v6 ISA
- Thumb Code Compression
- DSP extensions
- 8-Stage Pipeline (35% enhanced)
- SIMD media processing
- Multi-port 64bit memory system
- Optional VFP coprocessor
- TCM(scratch-pad) memory

ARM's First MPCore Architecture based on ARM11



INTEL SA & XSCALE Family

SA1110

- 233MHz (Typical)

SRAM

I/D MMU
32K I/D Cache
2K Mini/D Cache
64-bit BIU
Xcale Core

PXA210 (250)

- Up to 400MHz

SA1100

- 199MHz

MMU
8K-D Cache
16K-I Cache
DRAM
SA Core

MMU
16K-I Cache
DRAM
SA Core

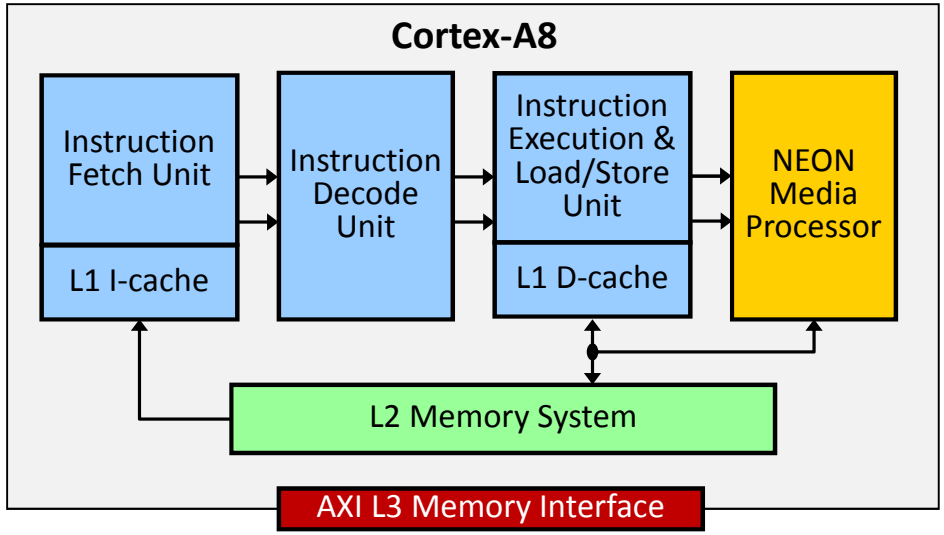
SA110

- Up to 203MHz(Typical)
- Only Core Support

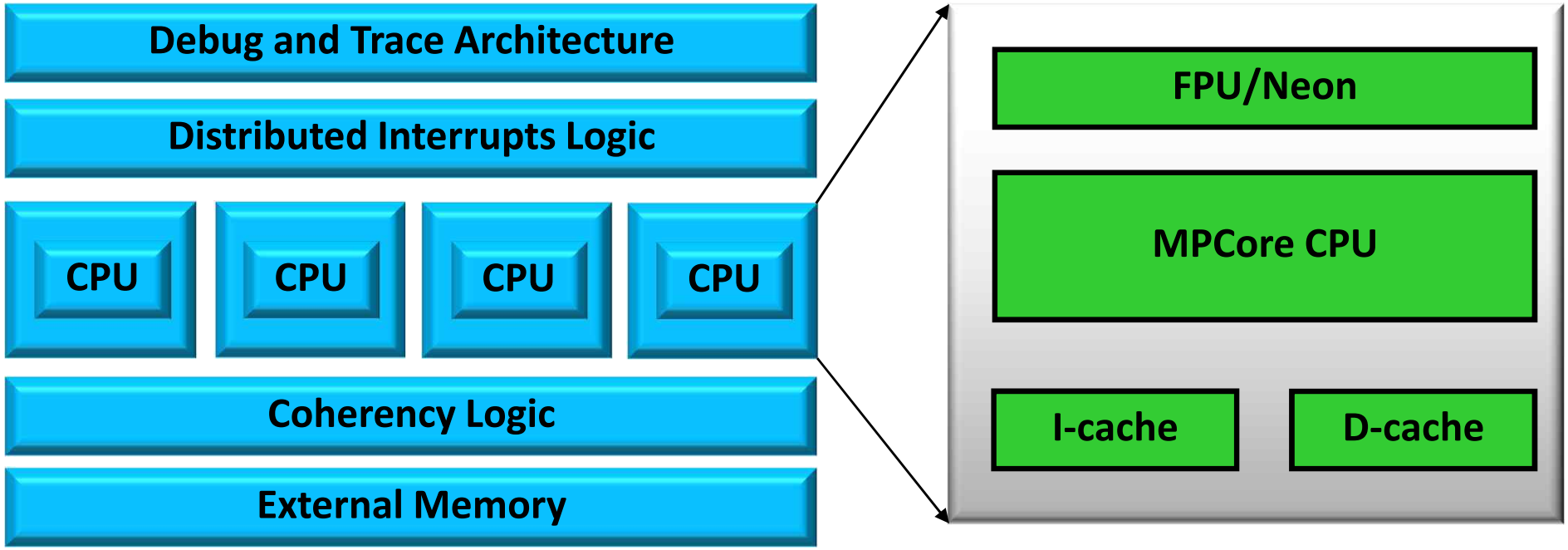
- SA core (StrongARM)
  - ARM8
  - ARM v4 Architecture
  - Harvard Architecture, I/D Cache
  - 5-Stage Pipeline
- Xscale
  - ARM5T Architecture
  - Thumb mode Support
  - 7-stage integer, 8-stage
  - memory superpipeline

Cortex-A8 Architecture

- Cortex-A Base Architecture
  - Thumb-2 technology for power efficient execution
  - TrustZone™ for secure applications
  - v6 SIMD for compatibility with ARM11
  - Media acceleration applications
- Cortex-A8 Extensions
  - Jazelle-RCT for efficient acceleration of execution environments such as Java and Microsoft .NET
  - NEON technology accelerating multimedia gaming and signal processing applications
  - VFPv3 supports full IEEE 754 specification and has been expanded to support 32 registers



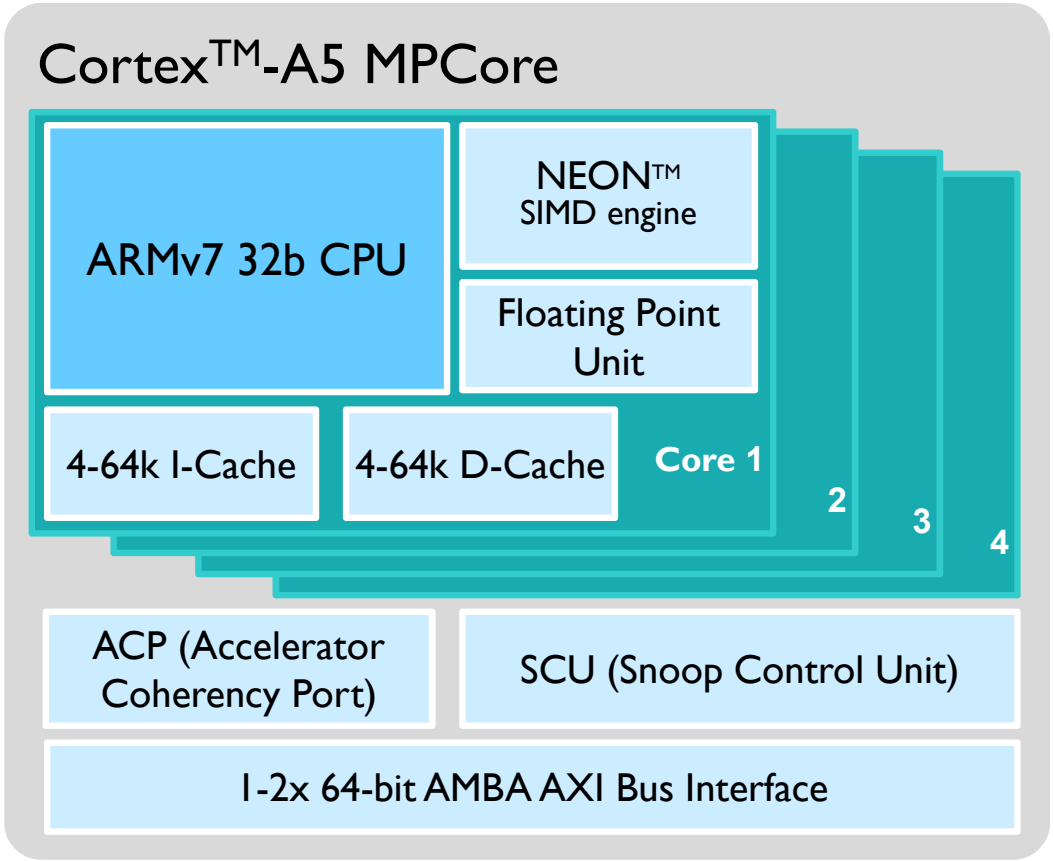
Current ARM MP Core Architecture Overview



- Scalable 1-4 cores solutions (ARM11, Cortex-A5, Cortex-A7, Cortex-A9, Cortex-A15, Cortex-A53, Cortex-A57)
- Supporting AMP/SMP models and any combination of the two
- Memory system and cache subsystems optimized for MP
- Interrupt Controller designed for distribution across multiple cores

Source: ARM (2013)

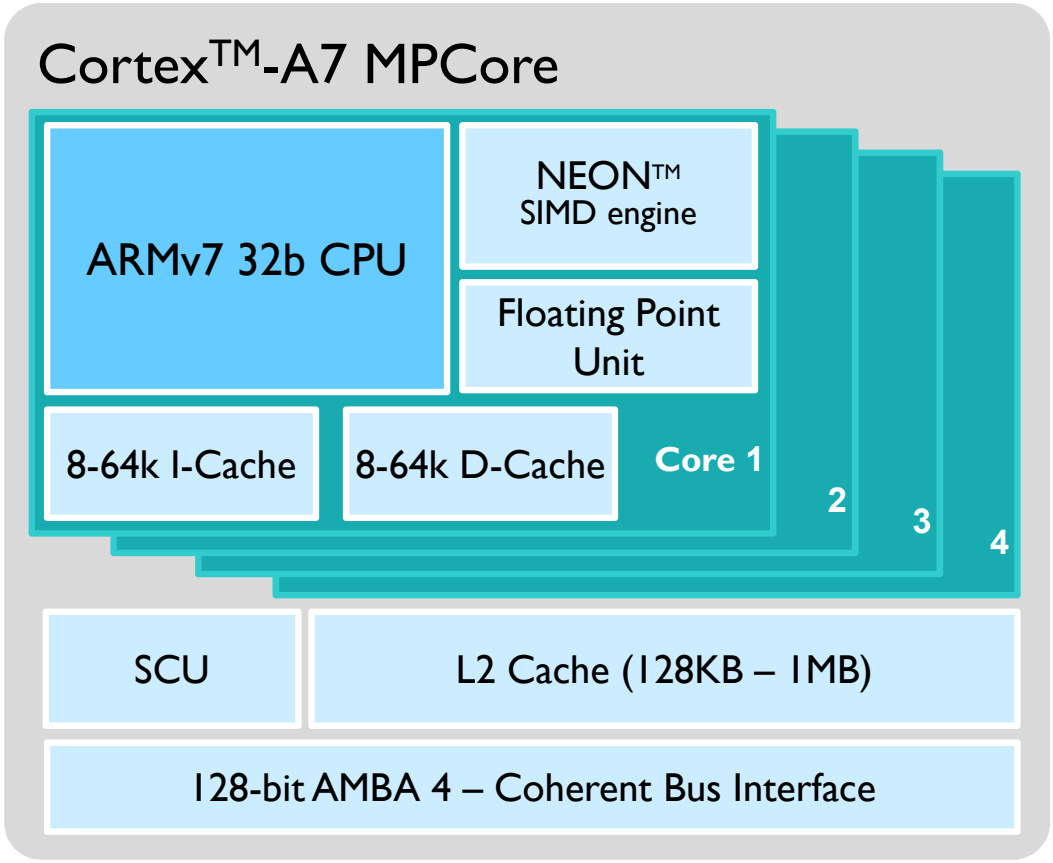
Cortex-A5: High Volume and Value



- Power-efficient Performance
  - Simple in-order 8-stage, single-issue pipeline
  - Improved branch prediction
- Full feature set of Cortex-A9
  - NEON, FPU, TrustZone
  - Symmetric Multi-processing (SMP)
- Highly configurable
  - Uniprocessor only version available
  - Optional NEON / FPU
  - Optional external L2 cache

Source: ARM (2013)

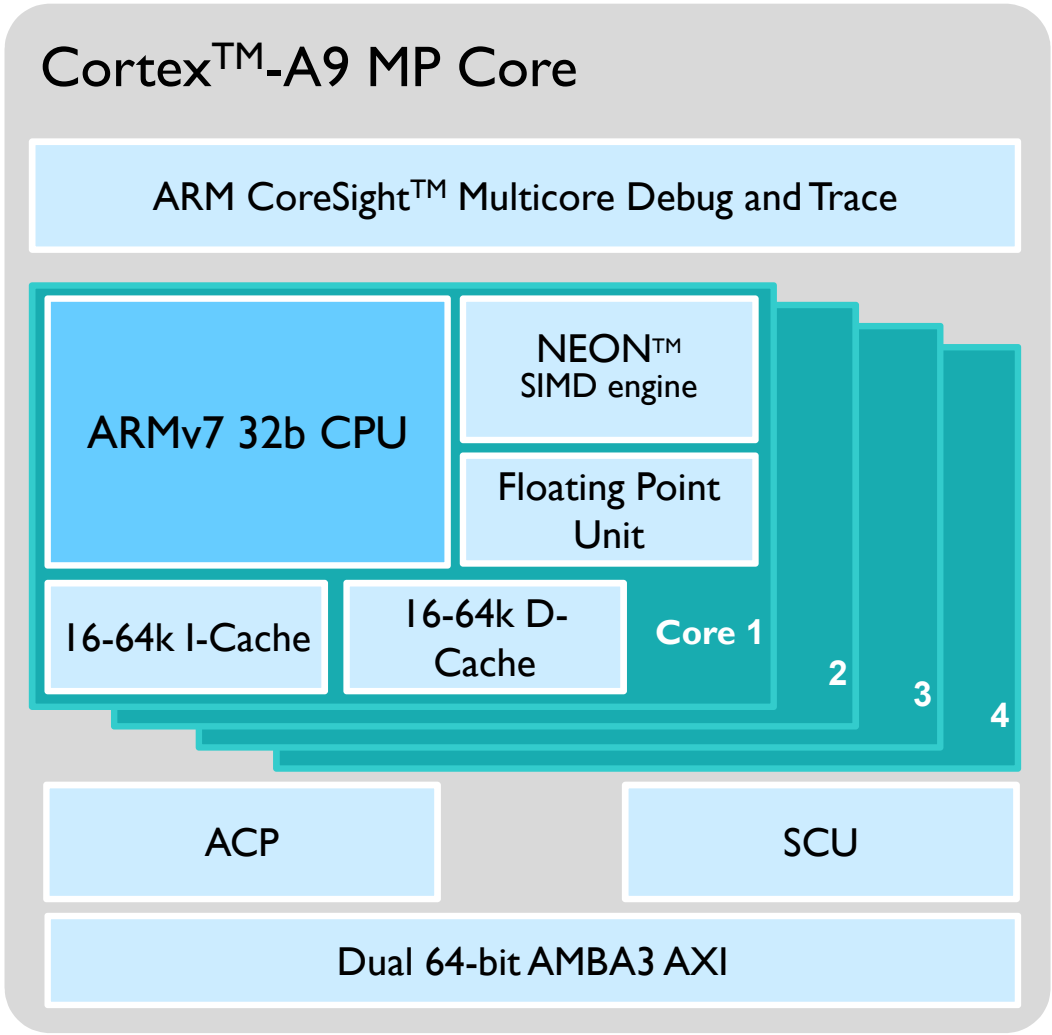
Cortex-A7: Most Efficient ARMv7



- Power efficient μ-architecture
  - In-order 8-stage, partial dual-issue
  - Efficient memory system
    - Integrated L2 cache
- Full feature set of Cortex-A15
  - Hardware enhanced virtualization
  - 1 TB physical memory (40bit addressing)
  - NEON, FPU, TrustZone
  - big.LITTLE companion to Cortex-A15 using AMBA4 ACE

Source: ARM (2013)

Cortex-A9: Widely Adopted

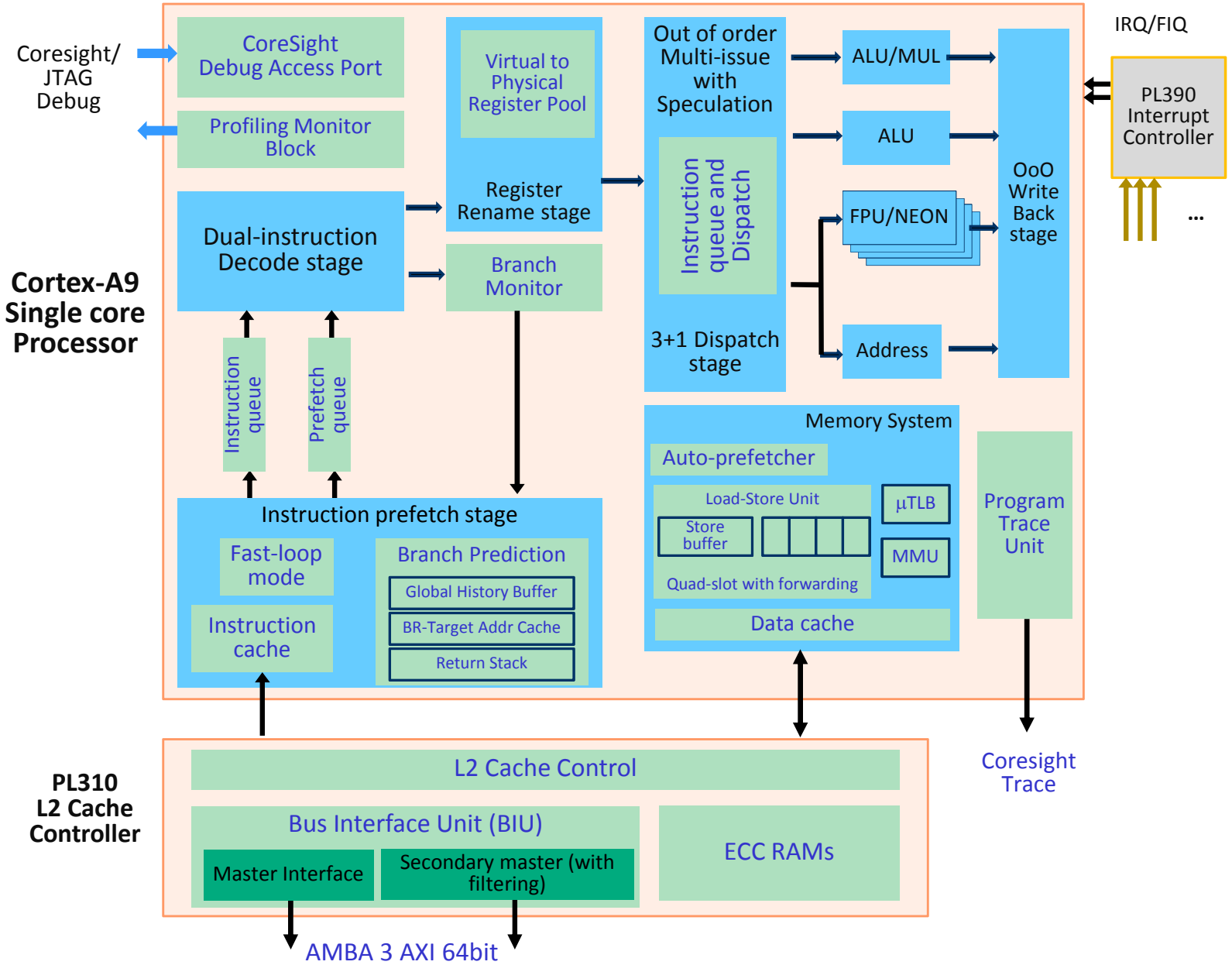


- Performance & power optimized multi-core processor
  - Dual-issue, Out-of-order pipeline
  - Scalable SMP – up to 4 cores
  - Accelerator Coherency Port (ACP)
- Flexible system architecture
  - Available as a single CPU also
  - Configurable cache sizes
  - Optional second AXI interface
  - Optimized L2 cache controller
  - NEON, FPU, TrustZone

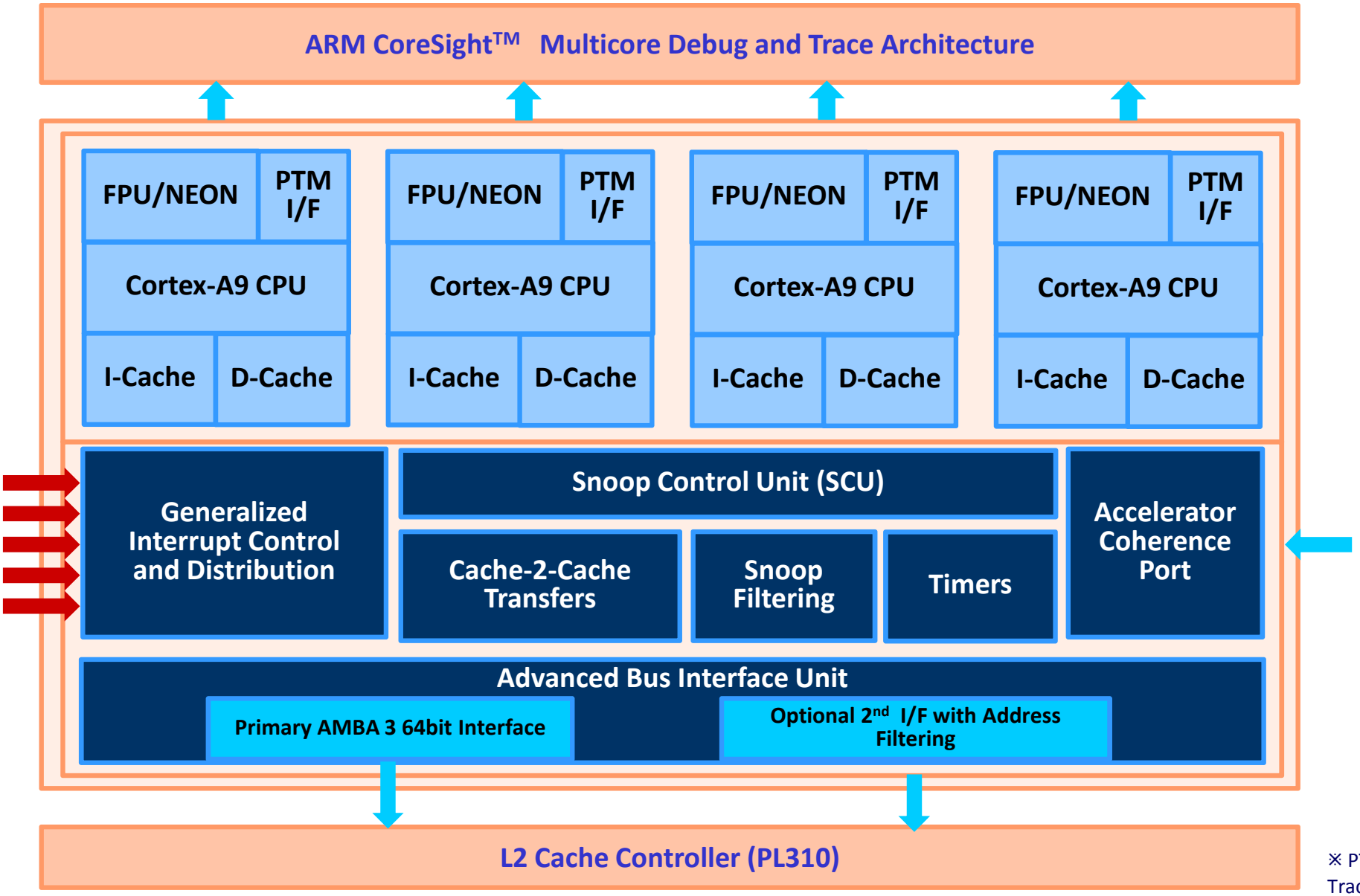
Source: ARM (2013)

Cortex-A9 Architecture

- Out-of-Order execution
- Register renaming to enable execution speculation
- Non-blocking memory system with load-store forwarding
- Fast loop mode in instruction pre-fetch to lower power consumption

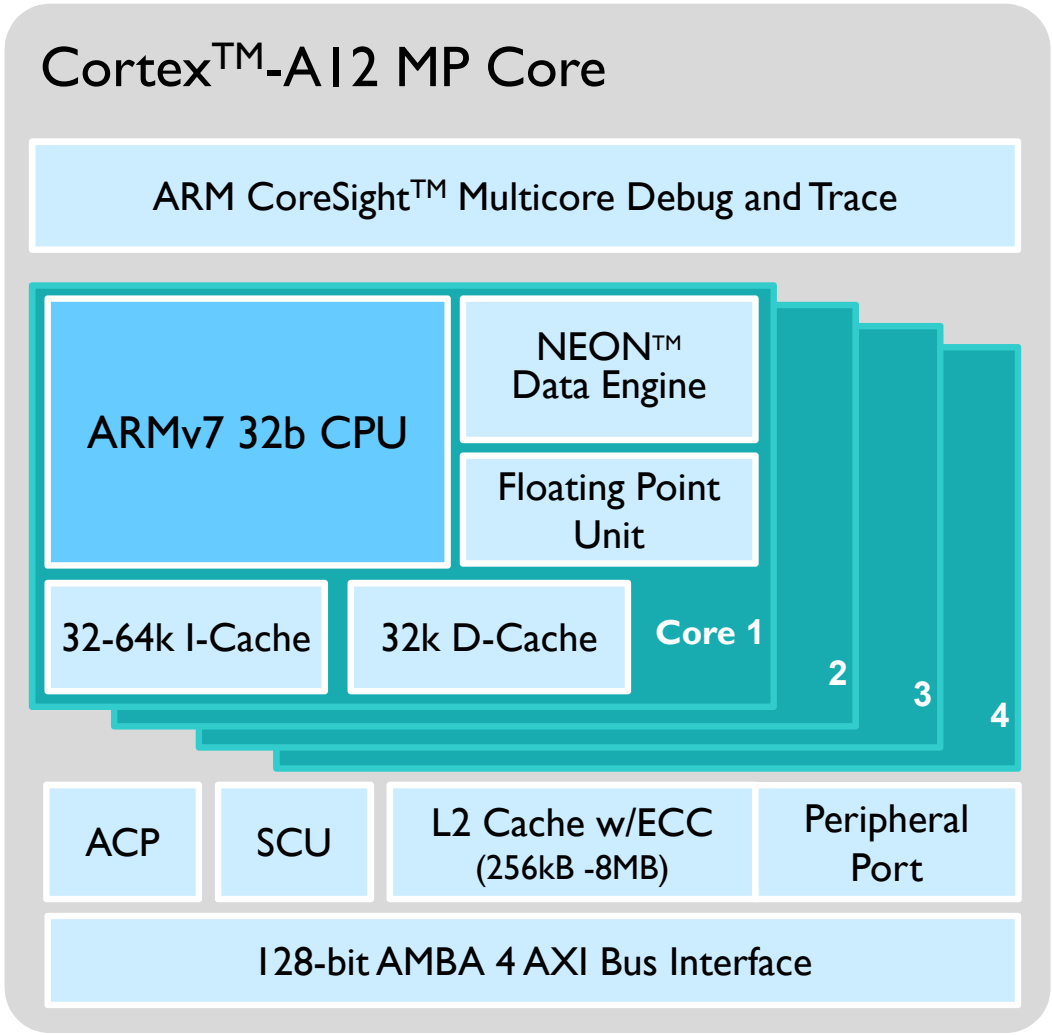


ARM Cortex-A9 MP Core Architecture



※ PTM: Program Trace Macrocell

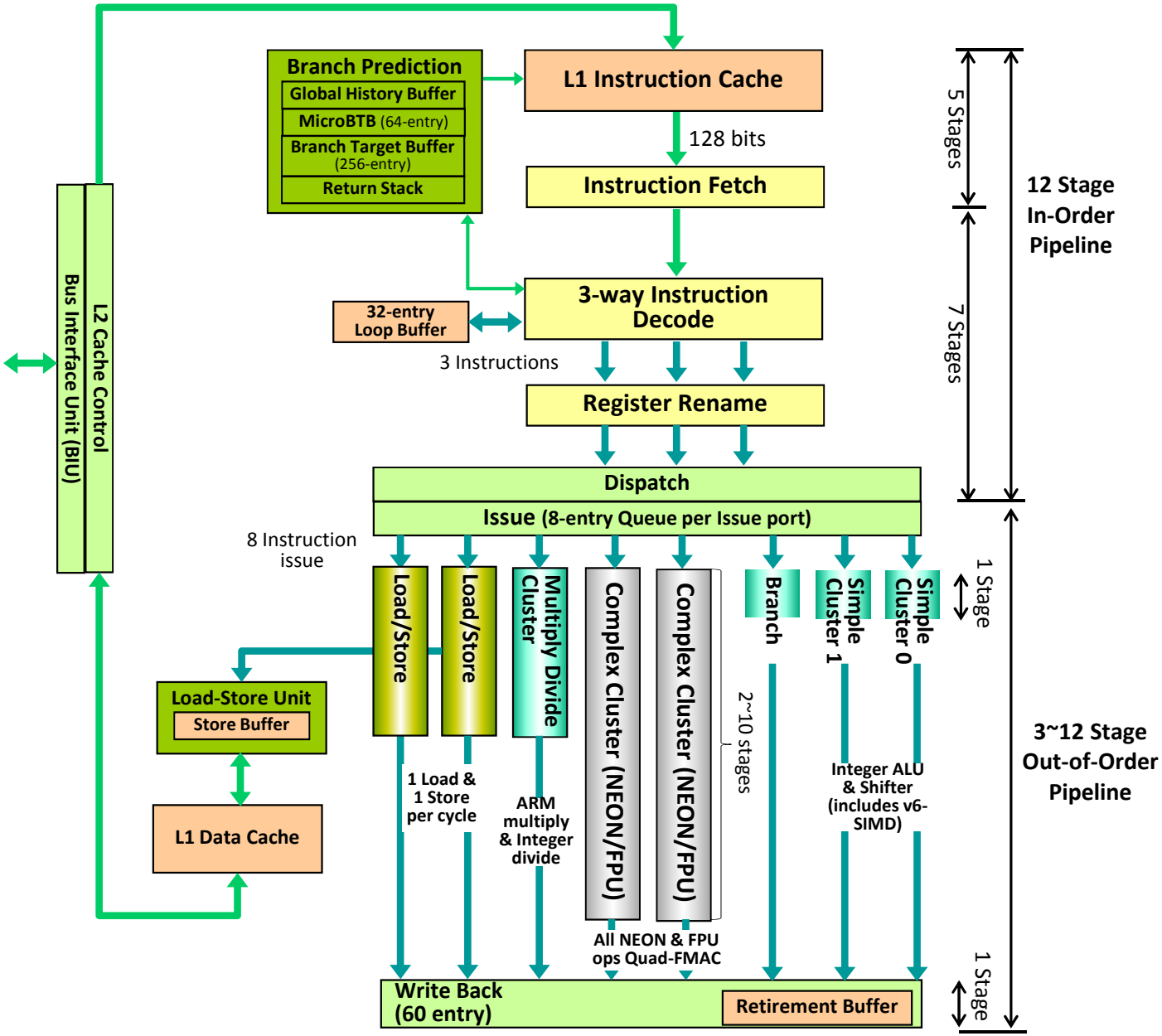
Cortex-A12 CPU: High Performance Embedded



- 40% performance uplift over Cortex-A9
- Same best-in-class energy efficiency
  - The most area- and cost-efficient solution
- Premium mobile features
  - big.LITTLE™ processing enabled
  - Greater than 4GB addressable memory
  - Security with Virtualization and TrustZone®

Source: ARM (2013)

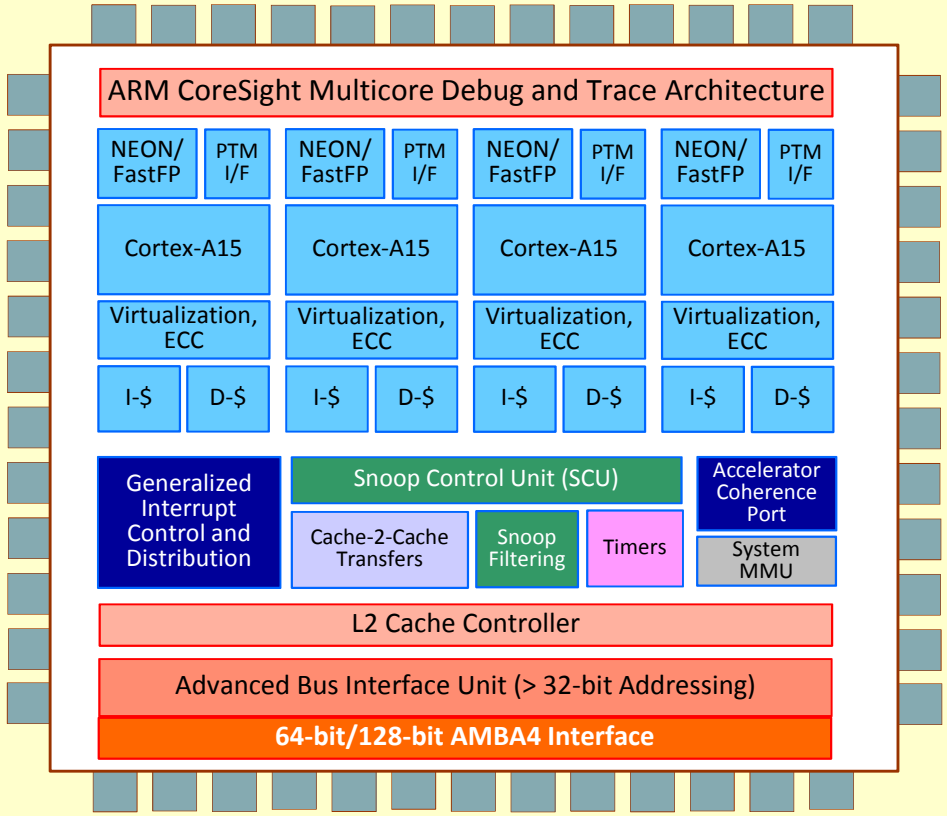
ARM Cortex-A15 Core Architecture



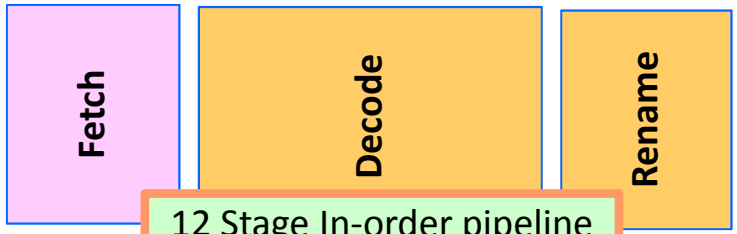
Source: ARM (2013)

ARM Cortex-A15 MP Core Architecture

- 2.5GHz in 28 HP process
  - 12 stage in-order, 3-12 stage OoO pipeline
  - 3.5 DMIPS/MHz ~ 8750 DMIPS @ 2.5GHz
- ARMv7A with 40-bit PA
- Dynamic repartitioning Virtualization
  - Fast state save and restore
  - Move execution between cores/clusters
- 128-bit AMBA 4 ACE bus
  - Supports system coherency
- ECC on L1 and L2 caches



Cortex-A15 Pipeline



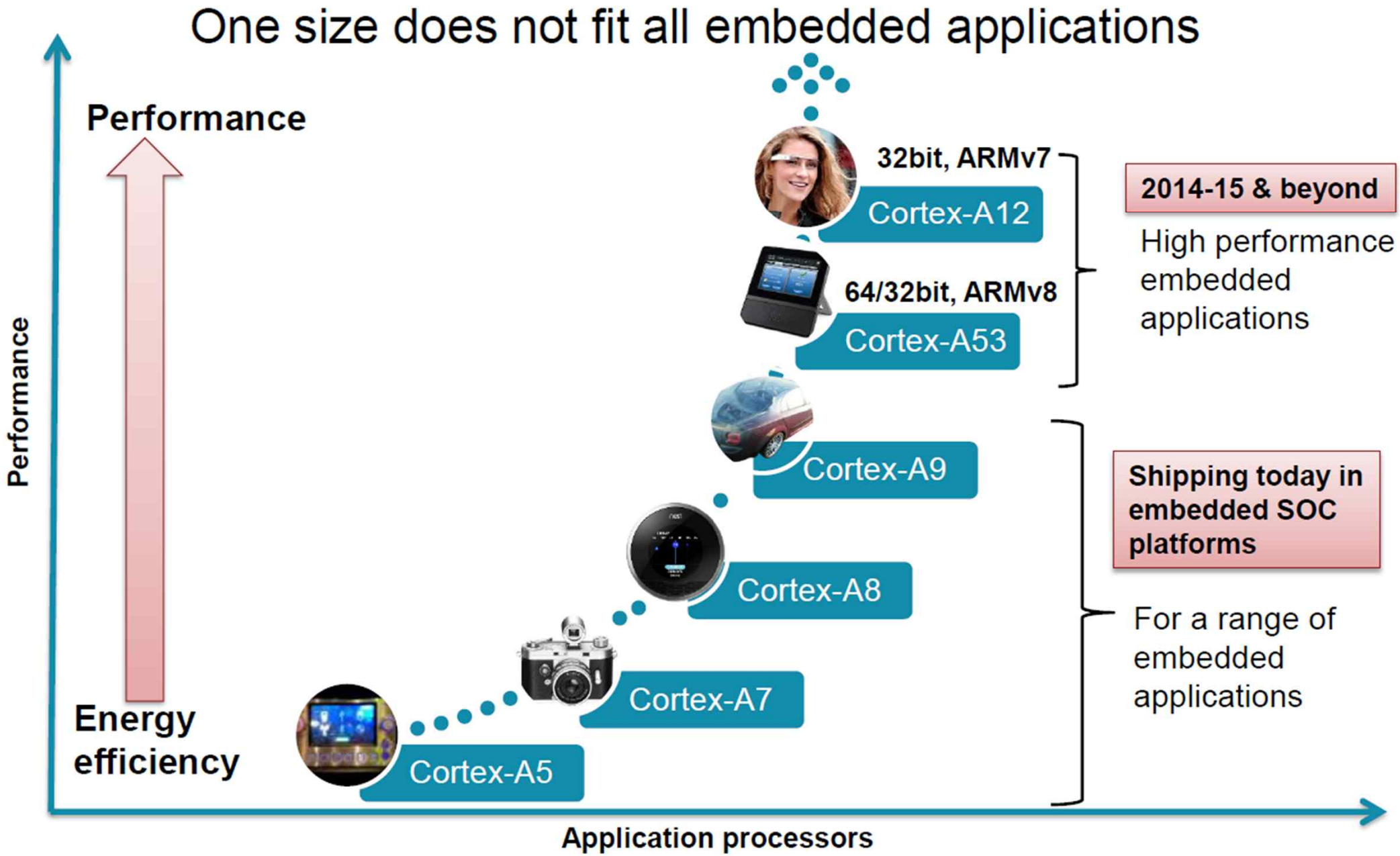
12 Stage In-order pipeline (Fetch, 3 Instruction Decode, Rename)

3~12 Stage out-of-order pipeline (capable of 8 issue)

※ PTM: Program Trace Macrocell

Source: ARM (2012)

ARM Application Processors for Embedded

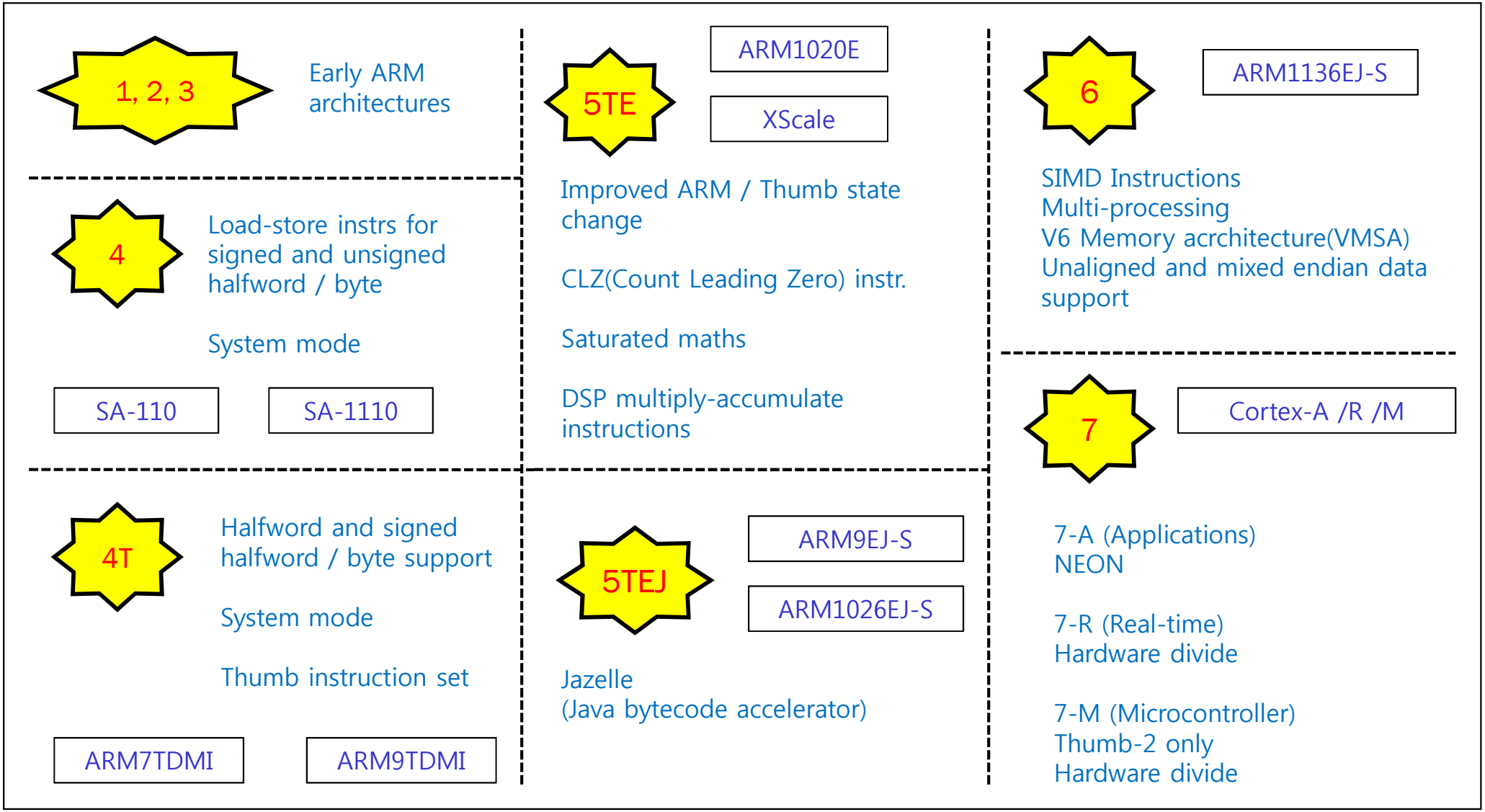


Summary ARMv7-A

Cortex-A5 MPCore	Thumb / Thumb-2 / DSP / SIMD / Optional VFPv4-D16 FPU / Optional NEON / Jazelle RCT and DBX, 1–4 cores / optional MPCore, snoop control unit (SCU), generic interrupt controller (GIC), accelerator coherence port (ACP)	4-64 KB / 4-64 KB L1, MMU + TrustZone	1.57 DMIPS / MHz per core
Cortex-A7 MPCore	Thumb / Thumb-2 / DSP / VFPv4-D16 FPU / NEON / Jazelle RCT and DBX / Hardware virtualization, in-order execution, superscalar, 1–4 SMP cores, Large Physical Address Extensions (LPAE), SCU, GIC, ACP, architecture & feature set are identical to A15, 8-10 stage pipeline	32 KB / 32 KB L1, 0-4 MB L2, MMU + TrustZone	1.9 DMIPS / MHz per core
Cortex-A8	Thumb / Thumb-2 / VFPv3 FPU / NEON / Jazelle RCT and DAC, 13-stage superscalar pipeline	16-32 KB / 16-32 KB L1, 0-1 MB L2 opt ECC, MMU + TrustZone	2.0 DMIPS/MHz
Cortex-A9 MPCore	Thumb / Thumb-2 / DSP / Optional VFPv3 FPU / Optional NEON / Jazelle RCT and DBX, out-of-order speculative issue superscalar, 1–4 SMP cores, SCU, GIC, ACP	16-64 KB / 16-64 KB L1, 0-8 MB L2 opt parity, MMU + TrustZone	2.5 DMIPS/MHz per core
Cortex-A12 MPCore	Thumb-2 / DSP / VFPv4 FPU / NEON / Hardware virtualization, out-of-order speculative issue superscalar, 1–4 SMP cores, LPAE, SCU, GIC, ACP	32-64 KB / 32 KB L1, 256 KB-8 MB L2	3.0 DMIPS / MHz per core
Cortex-A15 MPCore	Thumb / Thumb-2 / DSP / VFPv4 FPU / NEON / Integer divide / Fused MAC / Jazelle RCT / Hardware virtualization, out-of-order speculative issue superscalar, 1–4 SMP cores, LPAE, SCU, GIC, ACP, 15-24 stage pipeline	32 KB I$ w/parity / 32 KB D$ w/ECC L1, 0-4 MB L2, L2 has ECC, MMU + TrustZone	At least 3.5 DMIPS/MHz per core

AArch32 Summary

AArch32 Evolution (1/4)



AArch32 Evolution (2/4)

- ARM v1
  - Developed at Acorn Computer Limited between 1983 and 1985
  - 26-bit addressing
  - No multiplication, CPSR, ...(never used in a commercial product)
- ARM v2
  - Still 26-bit addressing (version 1 extension)
  - 32-bit result multiply (with accumulate) and coprocessor support
  - Atomic load and store instruction (SWP) support (ARM v2a)
- ARM v3
  - Developed at ARM Limited (1990)
  - 32-bit addressing
  - CPSR, SPSR added
  - Multi mode support : Undefined, Abort mode added
  - ARM6, ARM610, ARM7, ARM710 core (Apple Newton PDA!)

AArch32 Evolution (3/4)

- ARM v4
  - Fully 32-bit addressing
  - ARMv4T : Thumb mode support (T variant)
  - ARMv4 is deployed for StrongARM and ARM810 / ARMv4T: ARM7TDMI, ARM720T, ARM9TDMI, ARM920T, ARM940T
- ARM v5TE{J}
  - Superset of ARMv4(1999)
  - Enhanced DSP Instruction
  - Jazelle : Java H/W Acceleration
  - ARM9E, ARM10TDMI, ARM 1020E, XScale
- ARM v6{Z|T2}
  - Media instruction(SIMD) support (2001)
  - ARMv6Z (TrustZone), ARMv6T2 (Thumb-2)
  - ARM1136J, ARM1176JZ

AArch32 Evolution (4/4)

- ARM Architecture Versions and Variants(ARM v7)
  - Application profile (ARM v7-A)
    - Memory management support (MMU)
    - Highest performance at low power
      - Influenced by multi-tasking OS system requirements
    - TrustZone and Jazelle-RCT for a safe, extensible system
  - Real-time profile (ARM v7-R)
    - Protected memory (MPU)
    - Low latency and predictability ‘real-time’ needs
    - Evolutionary path for traditional embedded business
  - Microcontroller profile (ARM v7-M, ARM v6-M)
    - Lowest gate count entry point
    - Deterministic and predictable behavior a key priority
    - Deeply embedded use

Extensions to ARMv7

- MPE – Multiprocessing Extensions
  - Added Cache and TLB Maintenance Broadcast for efficient MP
- VE - Virtualization Extensions
  - Adds hardware support for virtualization:
    - 2 stages of translation in the memory system
    - New mode and privilege level for holding an Hypervisor
      - With associated traps on many system relevant instructions
    - Support for interrupt virtualization
  - Combines with a System MMU
- LPAE – Large Physical Address Extensions
  - Adds ability to address up to 40-bits of physical address space

Announcement

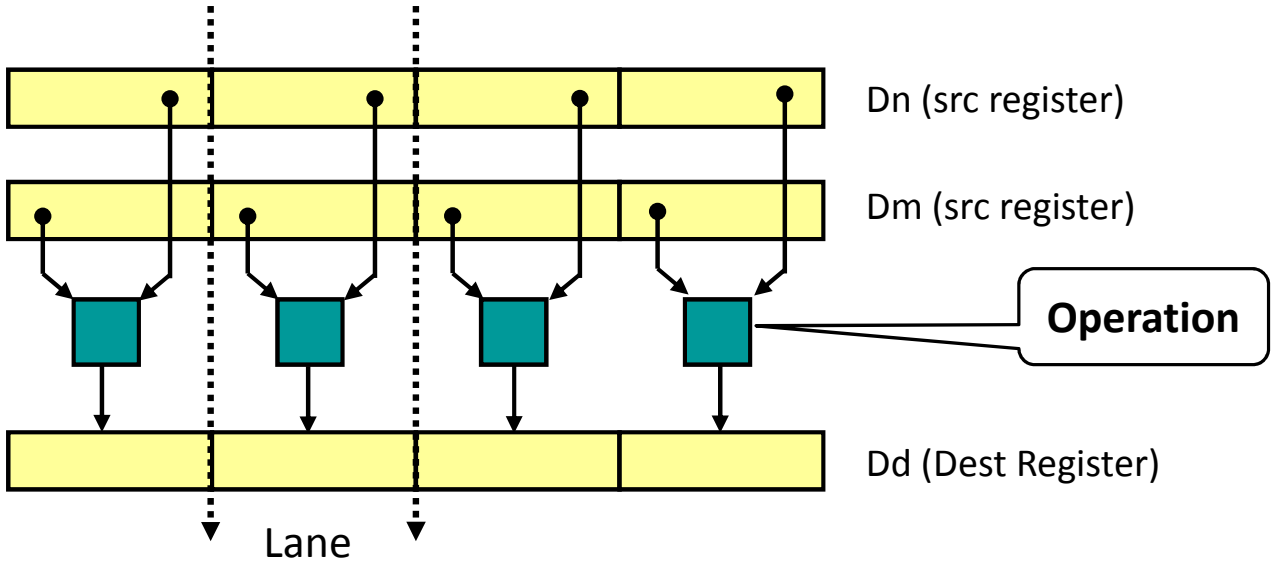
- Homework #3 (Due: Oct. 30)
  - Compare the ISA between Intel Atom and ARM Aarch-32
    - Free format: Either MS Word or MS PPT

ARM NEON

NEON?

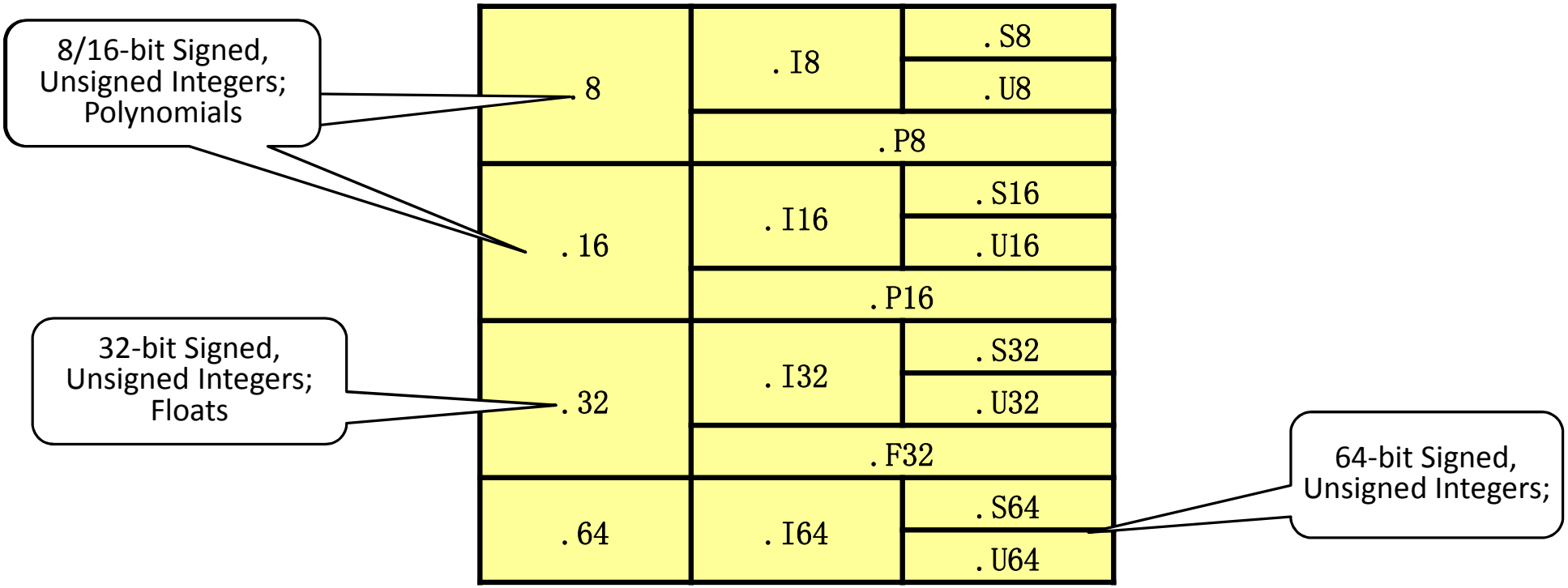
- NEON is a wide SIMD data processing architecture
  - Extension of the ARM instruction set
  - 32 registers, 64-bits wide (dual view as 16 registers, 128-bits wide)

- NEON Instructions perform “Packed SIMD” processing
  - Registers are considered as vectors of elements of the same data type
  - Data types can be: signed/unsigned 8-bit, 16-bit, 32-bit, 64-bit, single prec. float
  - Instructions perform the same operation in all lanes



NEON: Data Types

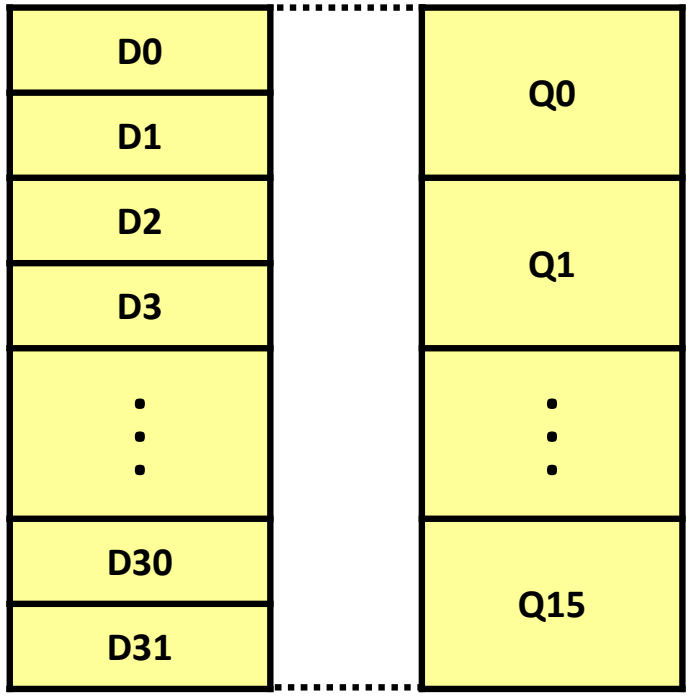
- NEON natively supports a set of common data types
  - Integer and Fixed-Point; 8-bit, 16-bit, 32-bit and 64-bit
  - 32-bit Single-precision Floating-point



- Data types are represented using a bit-size and format letter

NEON: Registers

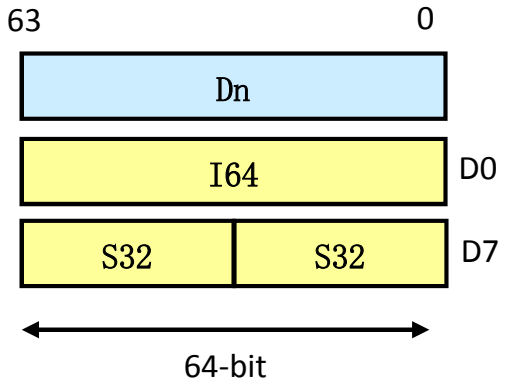
- NEON provides a 256-byte register file
  - Distinct from the core registers
  - Extension to the VFPv2 register file (VFPv3)
- Two explicitly aliased views
  - 32 x 64-bit registers (D0-D31)
  - 16 x 128-bit registers (Q0-Q15)
- Enables register trade-off
  - Vector length
  - Available registers

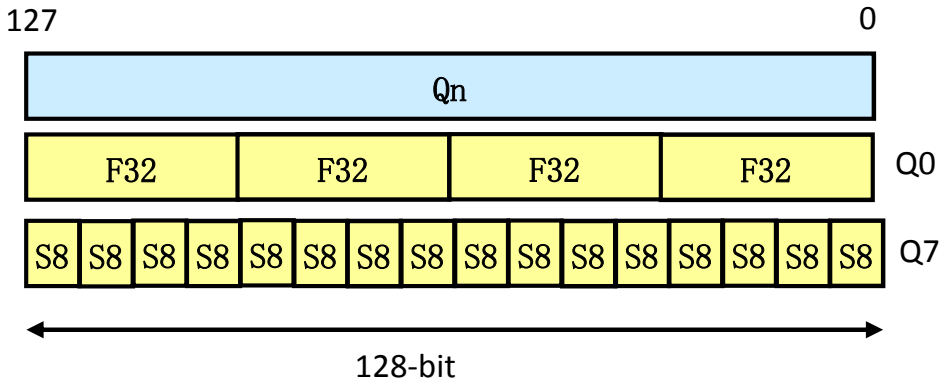


- Also uses the summary flags in the VFP FPSCR
  - Adds a QC integer saturation summary flag
  - No per-lane flags, so ‘carry’ handled using wider result (16bit+16bit → 32-bit)

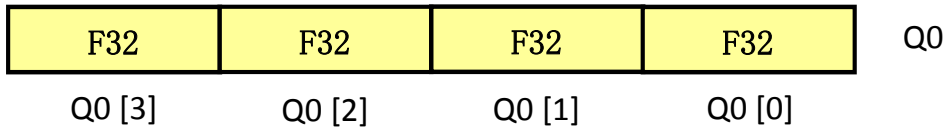
NEON: Vectors and Scalars

- Registers hold one or more elements of the same data type
  - Vn can be used to reference either a 64-bit Dn or 128-bit Qn register
  - A register, data type combination describes a vector of elements





- Some instructions can reference individual scalar elements
  - Scalar elements are referenced using the array notation Vn[x]



- Array ordering is always from the least significant bit

ARM VFP

VFP – Overview

- VFP – “Vector Floating-point”
  - A floating point hardware accelerator
- Described as a “coprocessor”
  - Originally a tightly-coupled coprocessor
  - Executed instructions from ARM instruction stream via dedicated interface
  - Now more tightly integrated into the CPU
- Single and Double precision floating-point
  - Fully IEEE compliant: ANSI/IEEE Std. 754-1985
  - Vector operation (Short vectors of up to 8 SP or 4 DP numbers) are handled particularly efficiently by the VFP architecture
    - Most arithmetic instructions can be used on these vectors, allowing SIMD parallelism
    - The FP Load & Store instructions have multiple register forms, allowing vectors to be transferred to and from memory efficiently

VFP – Registers

- VFP has 32 GP registers, each capable of holding a SP FP number or a 32-bit integer
- In D variants, these registers can also be used in pairs to hold up to 16 DP FP numbers:
  - FPSID (Read Only): Can be read to determine which implementations of the VFP architecture is being used
  - FPSCR: Supplies all user-level status and control
    - Status bits hold comparison results and cumulative flags for FP exceptions
    - Control bits are provided to select rounding options and vector length/stride, and to enable FP exception traps
  - FPEXC: Contains a few bits for system-level status and control

VFP – Instructions

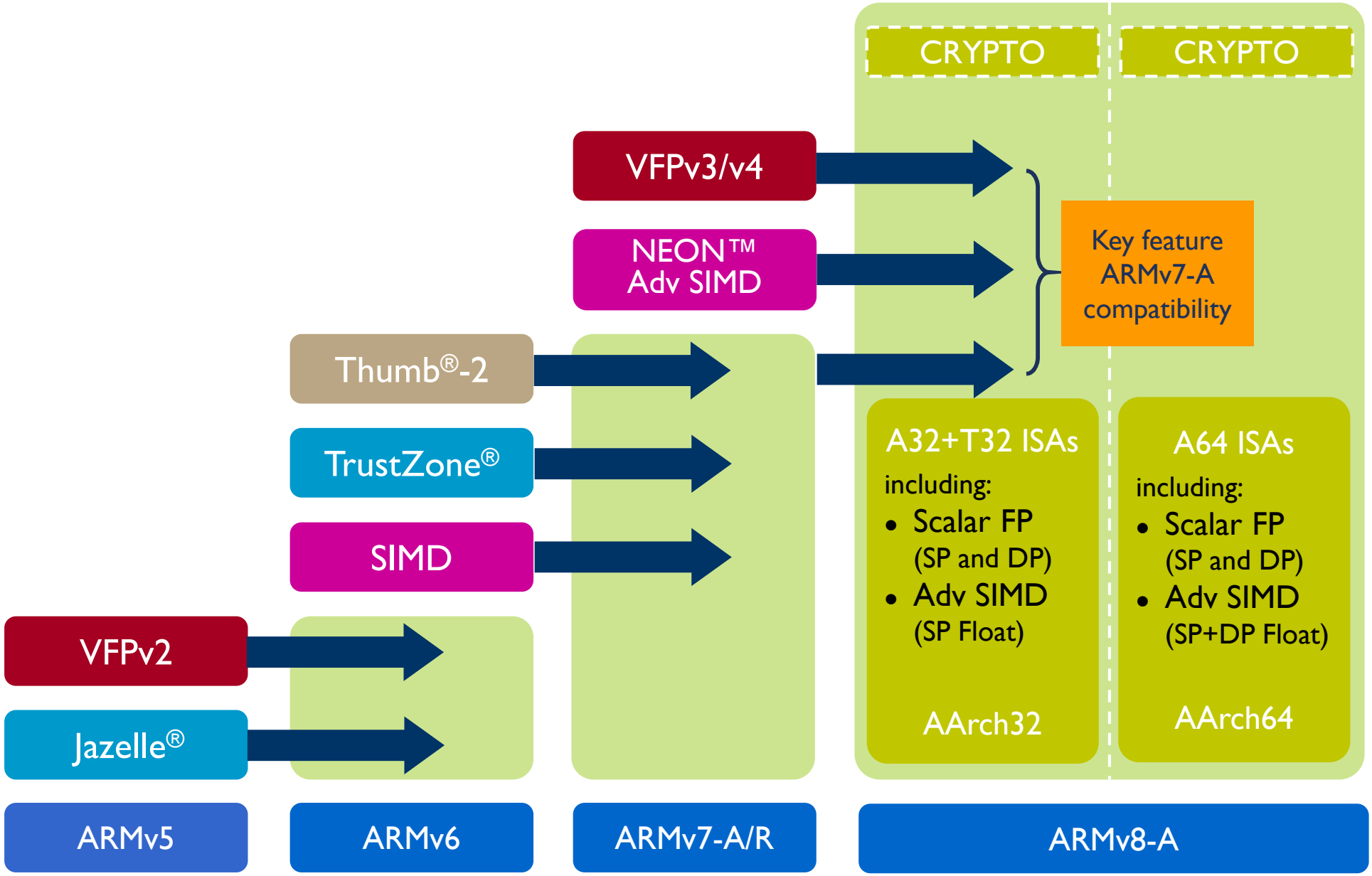
- Load /store
  - Some of these instructions allow multiple register values to be transferred, providing floating-point equivalents to ARM `LDM` and `STM` instructions
- Transfer 32-bit values directly between VFP and ARM GP registers
- Transfer 32-bit values directly between VFP system registers and ARM GP registers
- Add, subtract, multiply, divide, and square root
  - Can be used on short vectors as well as on individual floating-point values
- Copy floating-point values between registers
- Perform combined MAC operations on FP values and short vectors
- Perform conversions between SP, DP, unsigned 32-bit integers and two's complement signed 32-bit integers
- Compare floating-point values in registers with each other or with zero

VFP – Versions

- VFPv1 is obsolete
- VFPv2 is an optional extension to the ARMv5TE/5TEJ and ARMv6 architectures
- VFPv3-D32 is broadly compatible with VFPv2 but adds exception-less FPU usage, has 32 64-bit FPU registers as standard, adds VCVT instructions to convert between scalar, float and double, adds immediate mode to VMOV such that constants can be loaded into FPU registers
- VFPv3-D16: as above, but it has only 16 64-bit FPU registers
- VFPv3-F16 is uncommon; it supports IEEE754-2008 half-precision (16-bit) floating point
- VFPv4 in/for Cortex-A5, has a fused multiply-accumulate

ARM AArch64

ARM Architecture Evolution: AArch 32 to AArch 64



ARM v8 (AArch64) - Motivation

- Work on 64-bit architecture started in 2007
- Fundamental motivation is evolution into 64-bit
  - Ability to access a large virtual address space
  - Foresee a future need in ARM's traditional markets
  - Enables expansion of ARM market presence
- Developing ecosystem takes time
  - Development started ahead of strong demand
  - ARM now seeing strong partner interest in 64-bit
    - Though still some years from "must have" status

ARM v8 (AArch64) - Fundamentals

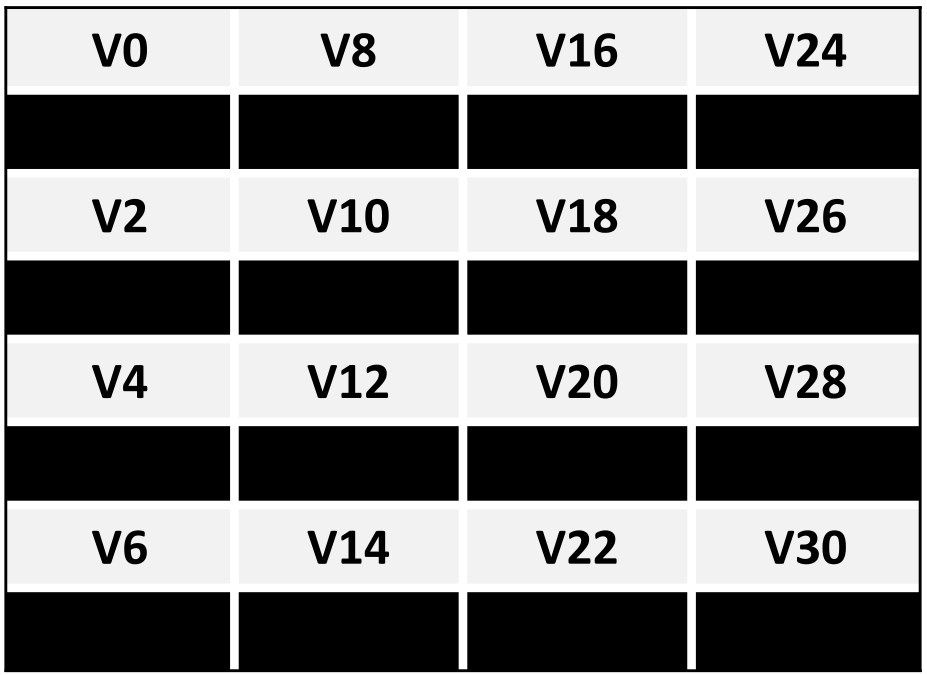
- New instruction set (A64)
- Revised exception handling for exceptions in AArch64 state
  - Fewer banked registers and modes
- Support for all the same architectural capabilities as in ARMv7
  - TrustZone
  - Virtualization
- Memory translation system based on the ARMv7LPAE table format
  - LPAE format was designed to be easily extendable to AArch64-bit
  - Up to 48 bits of virtual address from a translation table base register

ARM v8 (AArch64) – New Instruction Set

- New fixed length Instruction set
  - Instructions are 32-bits in size
  - Clean decode table based on a 5-bit register specifiers
- Instruction semantics broadly the same as in AArch32
  - Changes only where there is a compelling reason to do so
- 31 general purpose registers accessible at all times
  - Improved performance and energy
  - General purpose registers are 64-bits wide
  - No banking of general purpose registers
  - Stack pointer is not a general purpose register
  - PC is not a general purpose register
  - Additional dedicated zero register available for most instructions

AArch64 – Unbanked Registers

- 64-bit GP register file used for:
  - Scalar integer computation
    - 32 and 64-bit
  - Address computation
    - 64-bit
- Media register file used for:
  - Scalar Single and Double Precision FP
    - 32 and 64-bit
  - Advanced SIMD for Integer and FP
    - 64 and 128-bit wide vectors
  - Cryptography



ARM v8 (AArch64) – Key differences from AArch32

- New instructions to support 64-bit operands
  - Most instructions can have 32-bit or 64-bit arguments
  - Addresses assumed to be 64-bits in size
    - LP64 and LLP64 are the primary data models targeted
- Far fewer conditional instructions than in AArch32
  - Conditional {branches, compares, selects}
- No arbitrary length load/store multiple instructions
  - LD/ST ‘P’ for handling pairs of registers added

AArch32 / AArch64 Relationship

- Changes between AArch32 and AArch64 occur on "exception/exception return" only
  - Increasing exception level cannot decrease register width (or vice versa)
  - No Branch and Link between AArch32 and AArch64
- Allows AArch32 applications under AArch64 OS Kernel
  - Alongside AArch64 applications
- Allows AArch32 guest OS under AArch64 Hypervisor
  - Alongside AArch64 guest OS
- Allows AArch32 Secure side with AArch64 Non-secure side
  - Protects AArch32 Secure OS investments into ARMv8
- Requires architected relationship between AArch32 and AArch64 registers

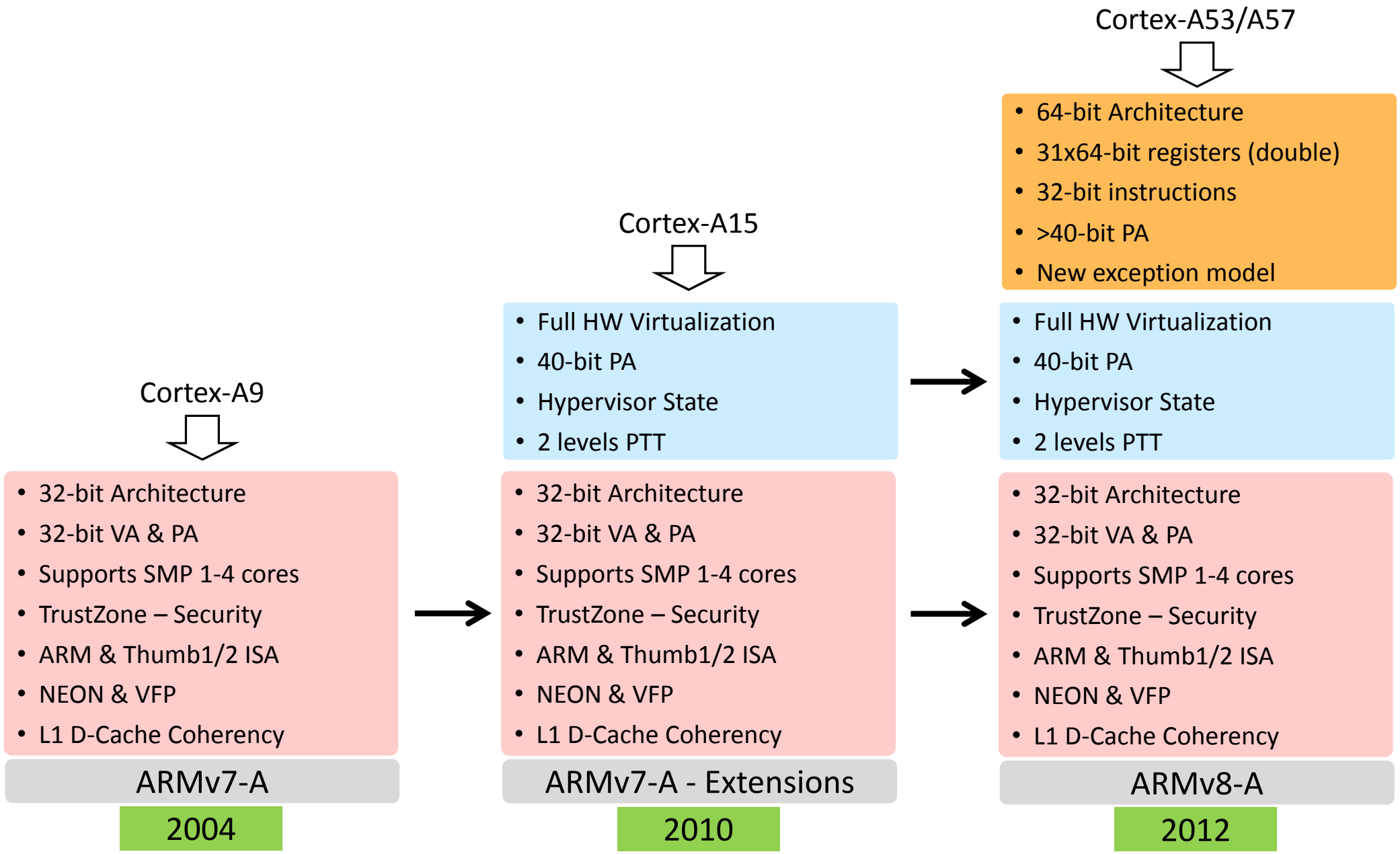
ARM v8 (AArch64) – Advanced SIMD and FP Instruction Set

- A64 Advanced SIMD and FP semantically similar to A32
  - Advanced SIMD shares the floating-point register file as in AArch32
- A64 provides 3 major functional enhancements:
  - More 128 bit registers: 32 x 128 bit wide registers
    - Can be viewed as 64-bit wide registers
  - Advanced SIMD supports DP floating-point execution
  - Advanced SIMD support full IEEE 754 execution
    - Rounding-modes, Denorms, NaN handling
- Register packing model in A64 is different from A32
  - 64-bit register view fit in bottom of the 128-bit registers
- Some Additional floating-point instructions for IEEE754-2008
  - MaxNum/MinNum instructions, Float to Integer conversions with RoundTiesAway

ARM v8 (AArch64) – Cryptography Support

- Instruction level support for Cryptography
  - Not intended to replace hardware accelerators in an SoC
- AES
  - 2 encode and 2 decode instructions
    - Work on the Advanced SIMD 128-bit registers
    - 2 instructions encode/decode a single round of AES
- SHA-1 and SHA-256 support
  - Keep running hash in two128 bit wide registers
  - Hash in 4 new data words each instruction
  - Instructions also accelerate key generation

A-series Evolution Details



ARMv8: AArch64

High Performance

Cortex™-A57 MPCore

ARM CoreSight™ Multicore Debug and Trace

ARMv8 32b/64b CPU
Virtual 44b PA

NEON™
SIMD engine with
crypto ext.

Floating Point
Unit

48k I-Cache
with DED parity

32k D-Cache
w/ECC

Core 1 2 3 4

ACP | SCU | L2 Cache w/ECC (512kB ~ 2MB)

128-bit AMBA ACE Coherent Bus Interface

Energy Efficient

Cortex™-A53 MPCore

ARM CoreSight™ Multicore Debug and Trace

ARMv8 32b/64b CPU
Virtual 40b PA

NEON™
SIMD engine with
crypto ext.

Floating Point
Unit

8~64k I-Cache
w/parity

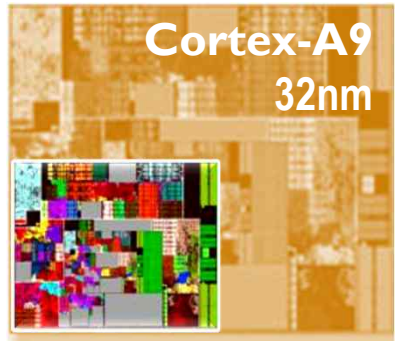
8~64k D-Cache
w/ECC

Core 1 2 3 4

ACP | SCU | L2 Cache w/ECC (128kB ~ 2MB)

128-bit AMBA ACE Coherent Bus Interface

Cortex-A53

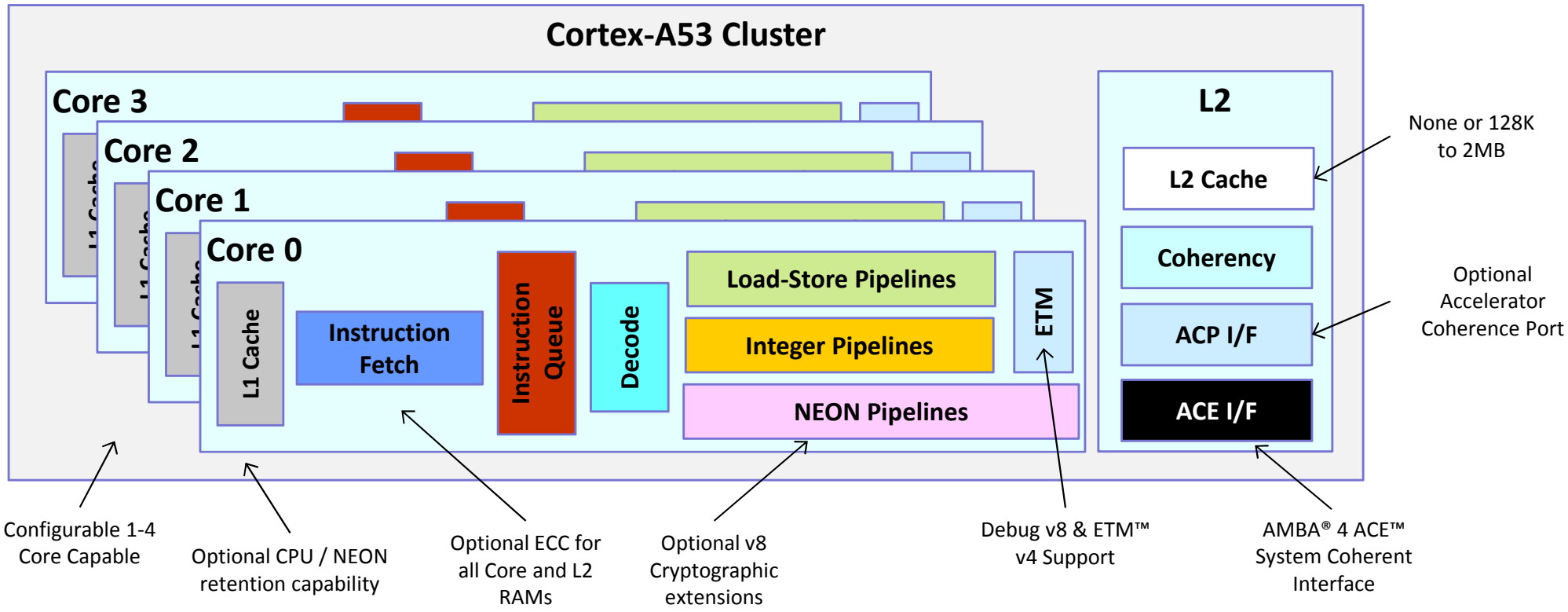


Cortex-A53,20nm

Source: ARM (2013)

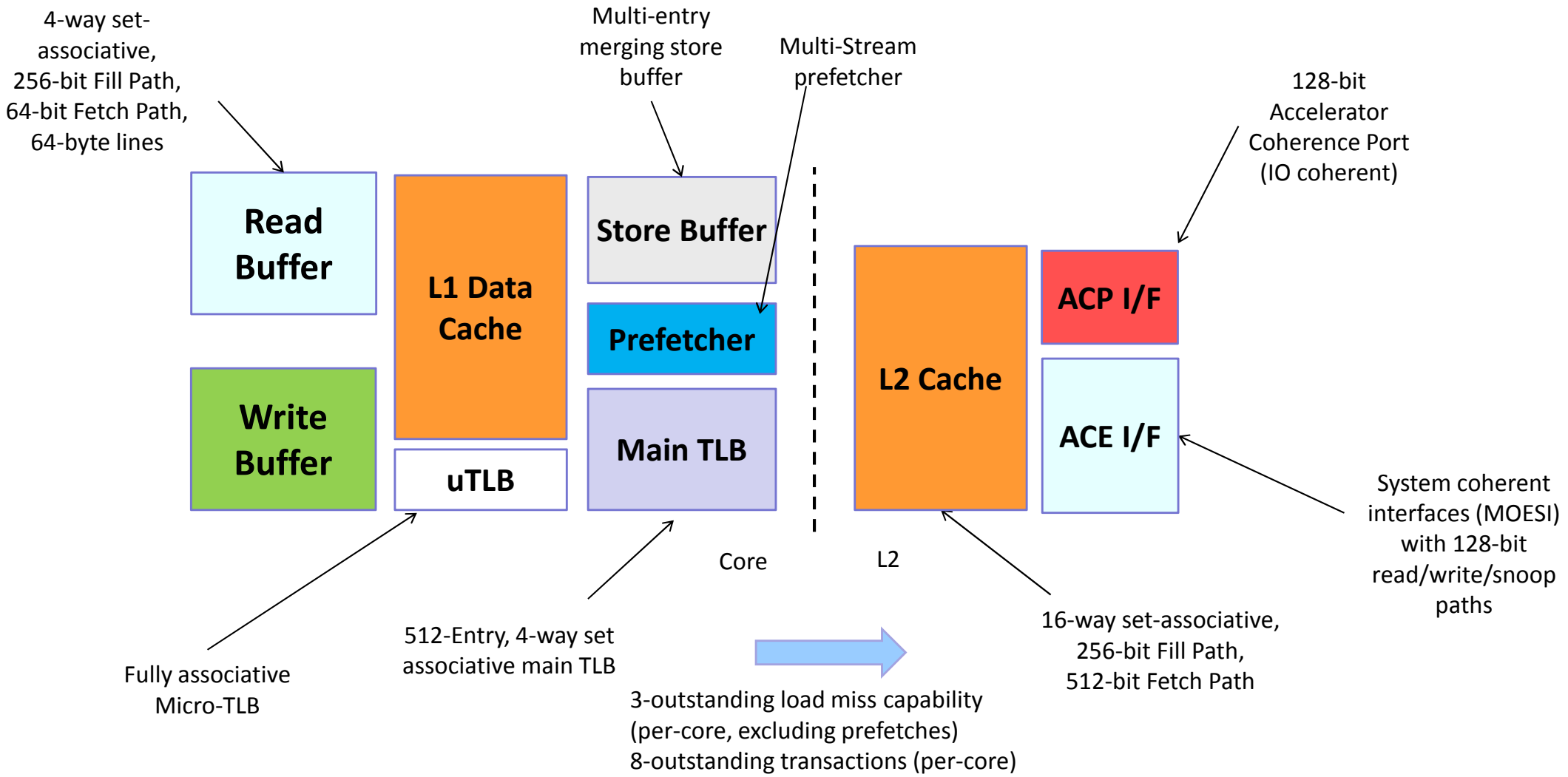
Cortex-A53 Features

- Cortex-A53 evolves our 8-Stage in-order LITTLE pipeline
  - ARM v8 AArch64 & AArch32 support at all exception levels
- Optional configurations for different markets
  - Smartphone to networking to offload



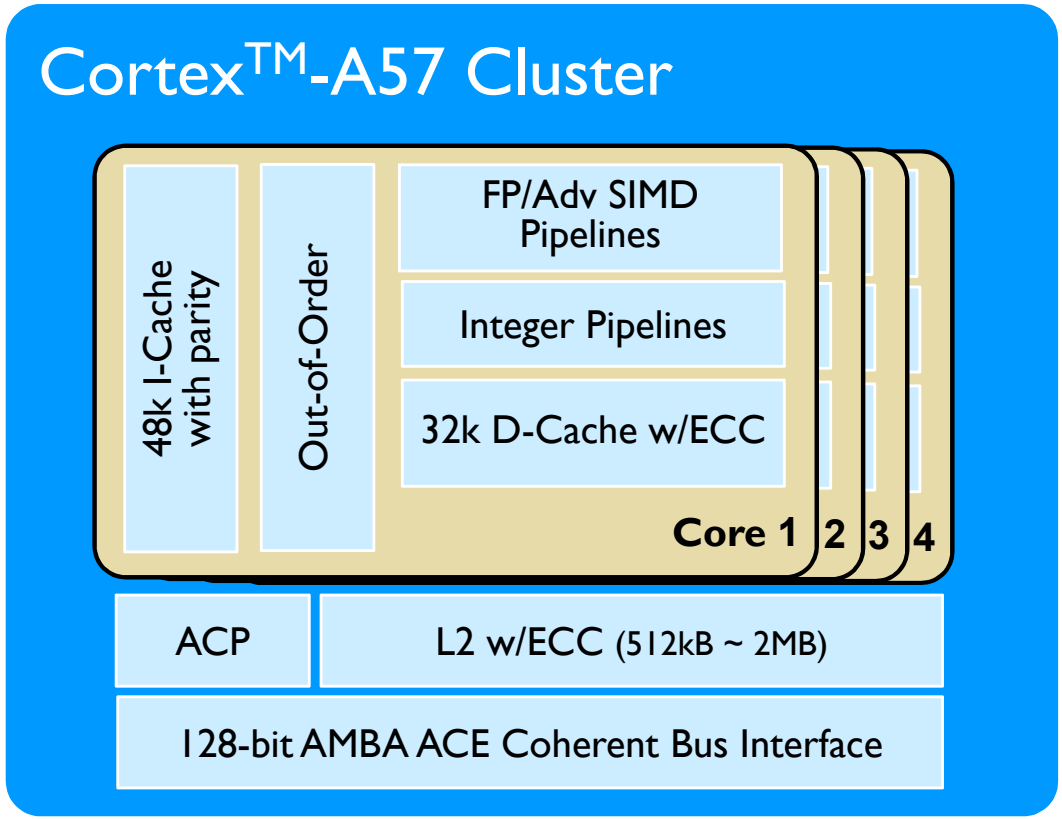
Cortex-A53 Core & L2 Memory System

- Tightly integrated Core (L1) to L2 Memory system

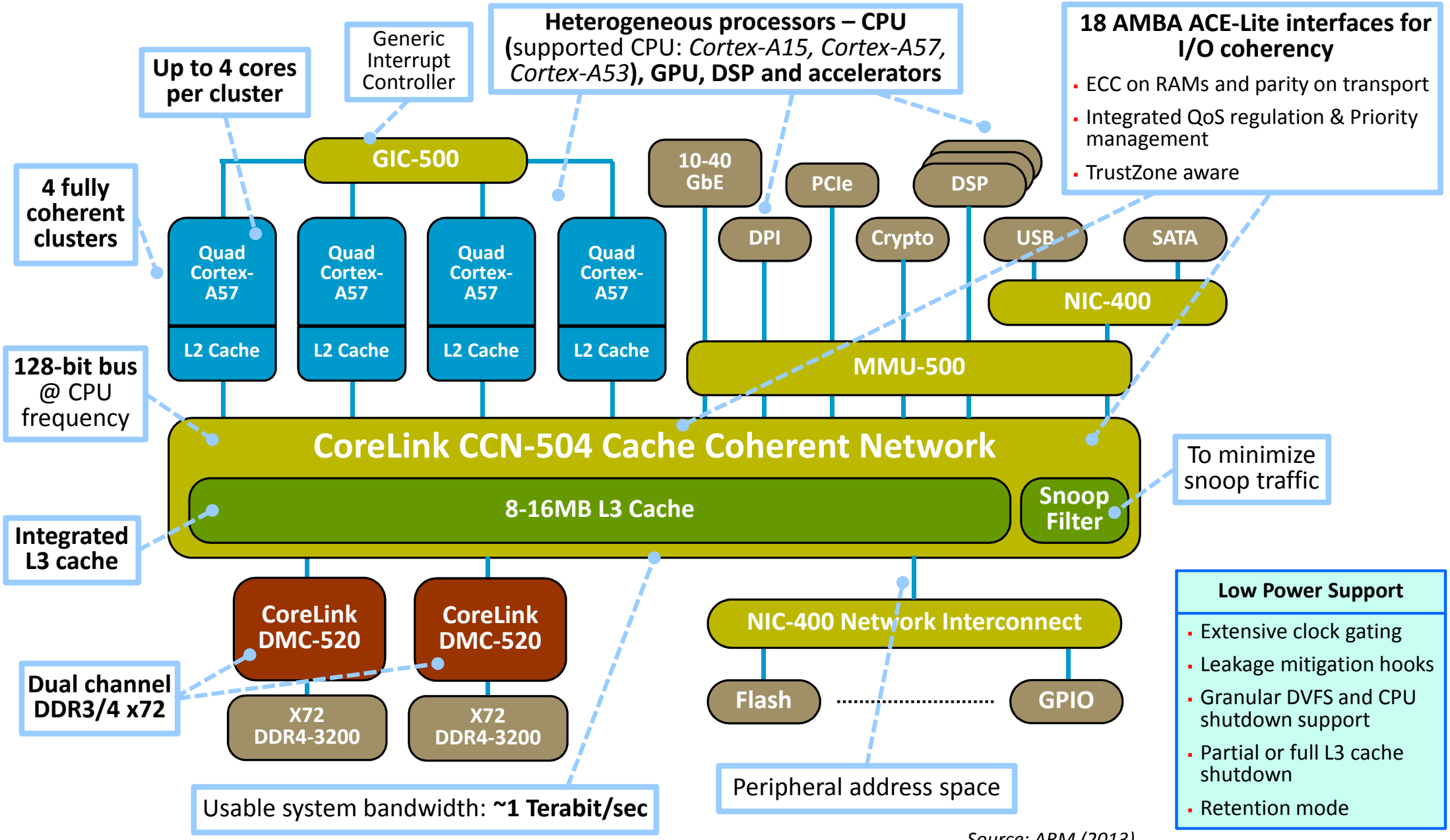


Cortex-A57 Features

- Fixed L1-cache size
  - 48KB I-cache
  - 32KB D-cache
- Configurable L2 cache size
  - 512KB/1MB/2MB
- Fully out-of-order execution
- Optional ARMv8 cryptography units
- Reliability features
  - Caches: SECDED ECC or parity
  - System-error capabilities
- 1-4 core MP configurability within a cluster
- System interface compatible with CCN-504 and CCI-400

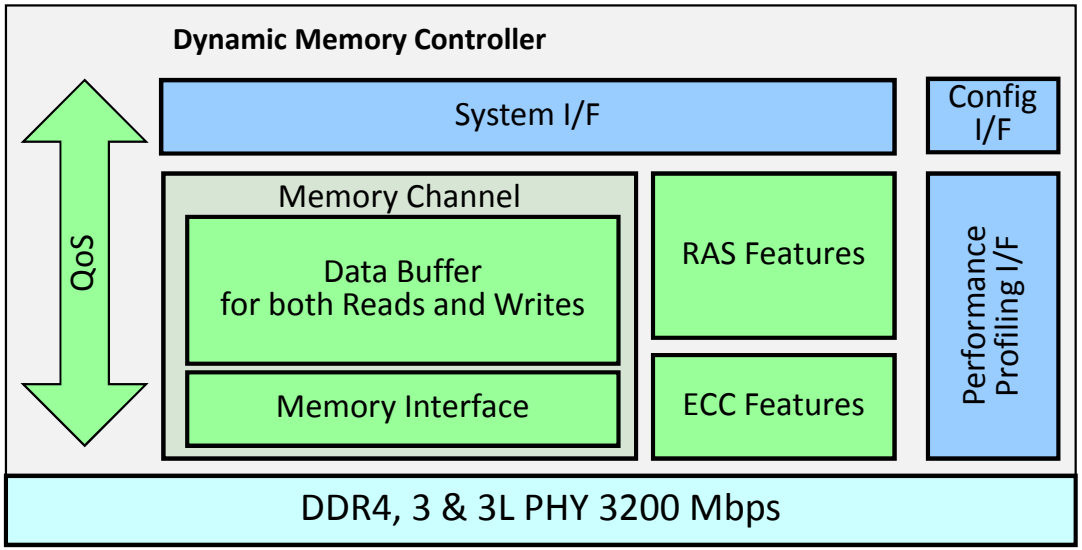


CoreLink CCN-504 Cache Coherent Network



Source: ARM (2013)

CoreLink DMC-520



Performance	Max bandwidth: 25.6 GB/s per channel Buffering to optimize read/write turnaround
Interfaces	Optimal direct connection to CCN-504 Industry standard DFI-3.0 to connect to PHY
PHY	Low-latency PHY from ARM
Memory	Support for x72 DRAM DDR3, DDR3L and DDR4 up to DDR4-3200
Low Power Support	Programmable DRAM power modes

※ RAS = Reliability, Availability, Serviceability

- 5th Generation ARM DMC targeting >95% DRAM efficiency
  - ECC and RAS features
  - Performance Profiling
  - TrustZone Address Space Control
- High efficiency through close integration with CCN-504
  - System wide QoS, designed and verified with ARM CPUs

Source: ARM (2013)

ARM Cache Architecture

CPU	Architecture	L1 Cache	L2 Cache
ARM720T	ARMv4T (32bit)	8KB Unified I&D cache, 4-way SA, 8 words/line, VIVT	-
ARM926EJS	ARMv5TEJ (32bit)	4KB~128KB configurable size, Harvard architecture, 4-way SA, 8 words/line, VIVT using Modified VA (MVA)	-
ARM1136EJS	ARMv6 (32bit)	4KB~64KB configurable size, Harvard architecture, 4-way SA, 8 words/line, VIPT	-
Cortex-A8	ARMv7-A (32bit)	16~32 KB configurable size, Harvard architecture, 4-way SA, 16 words/line, VIPT for I-cache, PIPT for D-cache	0-1 MB configurable size, unified, 8-way SA, 16 words / line, opt ECC or parity
Cortex-A9	ARMv7-A (32bit)	16~32 KB configurable size, Harvard architecture, 4-way SA, 8 words/line, VIPT for I-cache, PIPT for D-cache	0-8 MB configurable size, optional parity
Cortex-A7	ARMv7-A (32bit)	8~64 KB configurable size, Harvard architecture, I-cache: 2-way SA, 8 words/line, VIPT D-cache: 4-way SA, 16 words/line, PIPT	128KB-1 MB configurable size, 8-way SA, 16 words / line
Cortex-A15	ARMv7-A (32bit)	I-cache: 32 KB, 2-way SA, 16 words/line, Parity per 16 bits, PIPT D-cache: 32 KB, 2-way SA, 16 words/line, Parity per 32 bits, PIPT	512KB-4 MB configurable size, 16-way SA, 16 words/line, Optional parity, Strictly enforced inclusive with L1 data cache
Cortex-A53	ARMv8-A (64bit) a.k.a AArch64	8~64 KB configurable size, Harvard architecture, 2-way SA, 16 words/line, PIPT	**40-bit Physical Addresses** 128KB-2MB configurable size, 16-way SA, Strictly enforced inclusive with L1 data cache
Cortex-A57	ARMv8-A (64bit) a.k.a AArch64	I-cache: 48 KB, 3-way SA, ECC, PIPT D-cache: 32 KB, 2-way SA, ECC, PIPT	**44-bit Physical Addresses** 512KB-2MB configurable size, 16-way SA, ECC, Strictly enforced inclusive with L1 data cache

Announcement

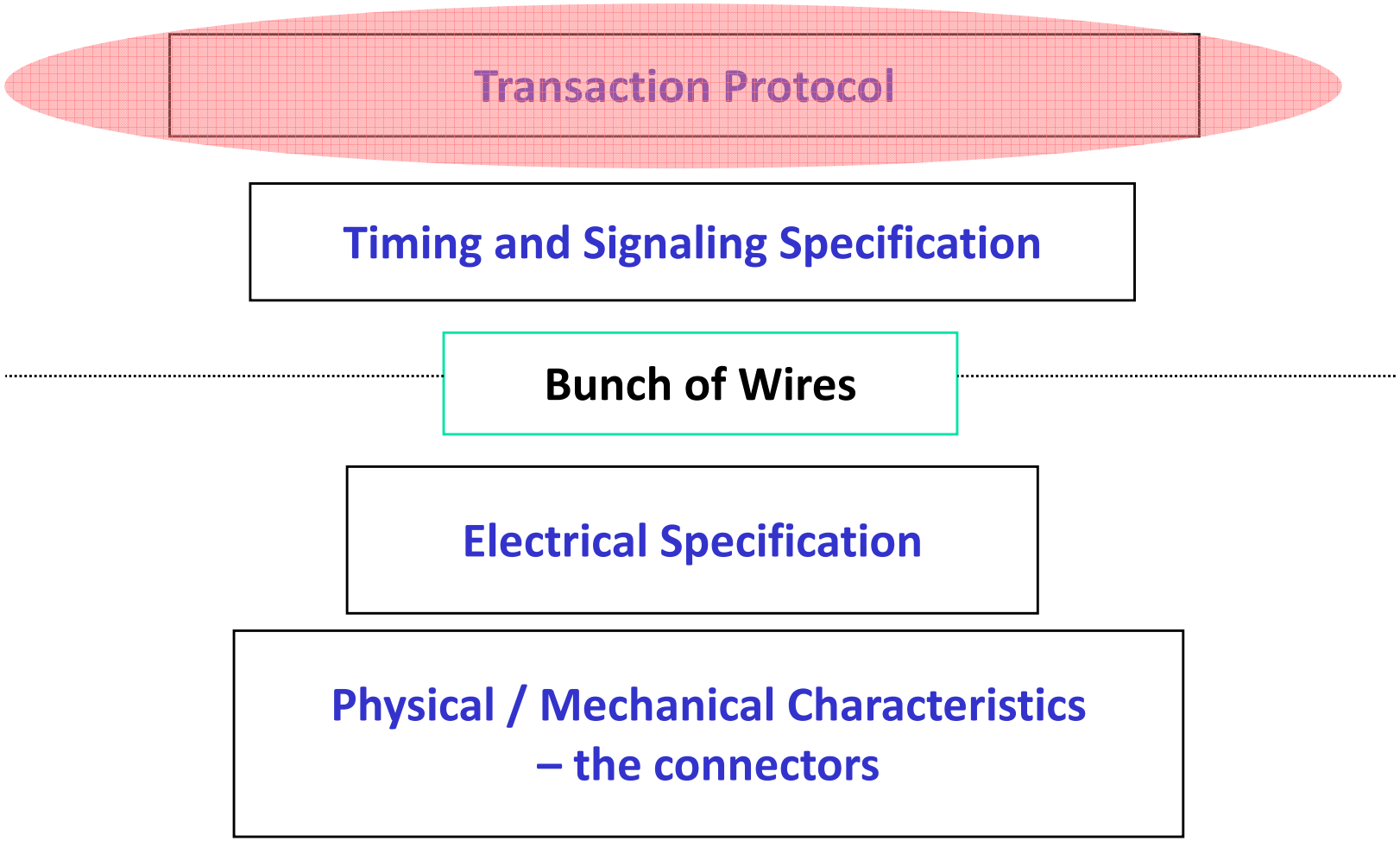
- Homework #4 (Due: Nov. 6)
  - Compare cache coherency schemes between snooping and Directory-based coherence
    - Free format: Either MS Word or MS PPT (in Korean)

ARM AMBA Bus Architecture

Master versus Slave

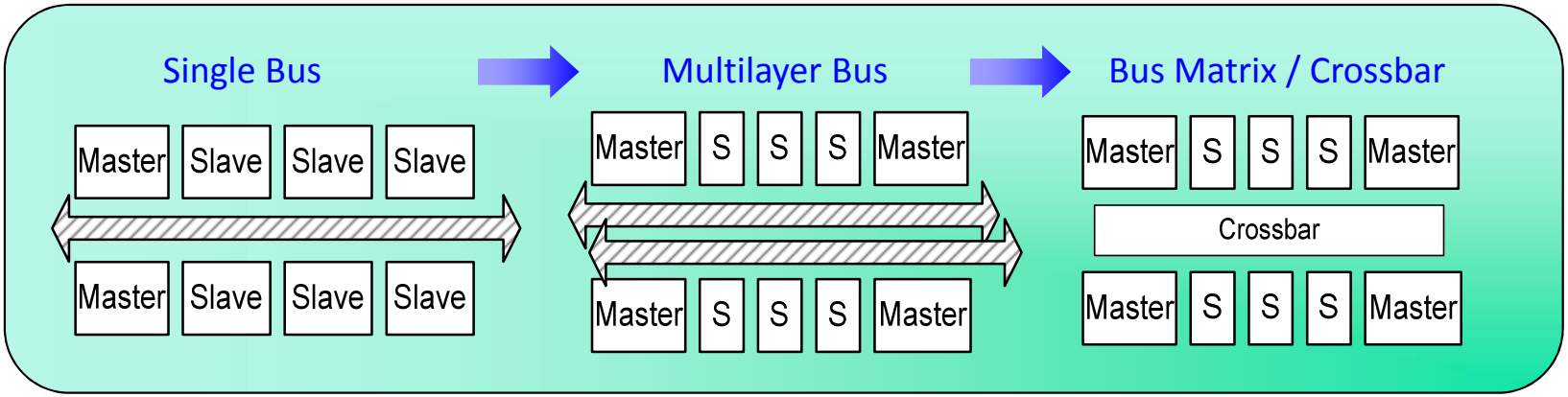
- A bus transaction includes two parts:
  - Issuing the command (and address) – request
  - Transferring the data – action
- Master is the one who starts the bus transaction by:
  - Issuing the command (and address)
- Slave is the one who responds to the address by:
  - Sending data to the master if the master ask for data
  - Receiving data from the master if the master wants to send data

What defines a BUS?

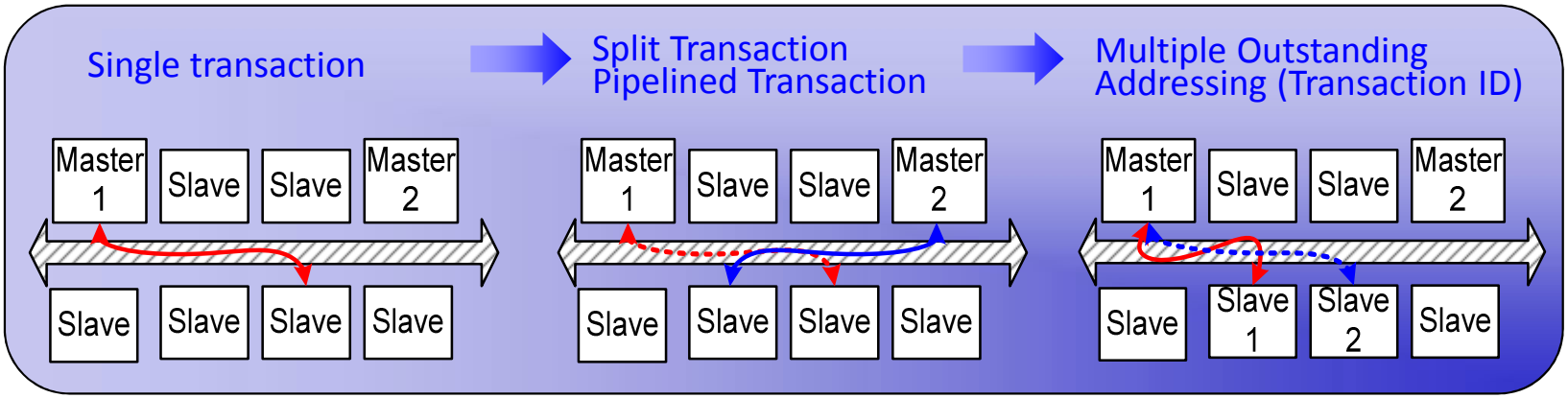


On-Chip Bus Technology

- **Bus Structure Evolution**

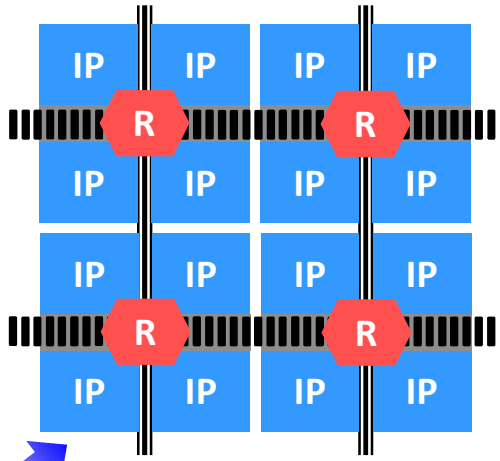


- **Bus Protocol Evolution**



ex) AMBA 1.0 → ex) AMBA 2.0 → ex) AMBA AXI
Sonics Backbone

Bus to Network

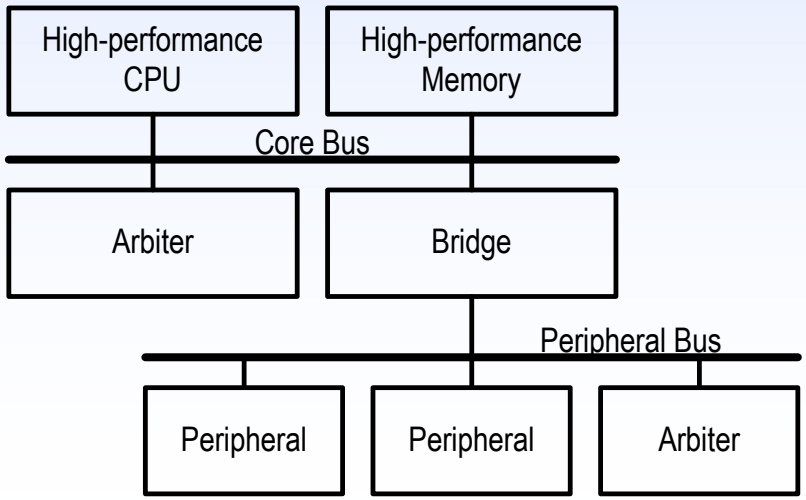


Network-on-Chip

On-chip Bus Architecture Classification

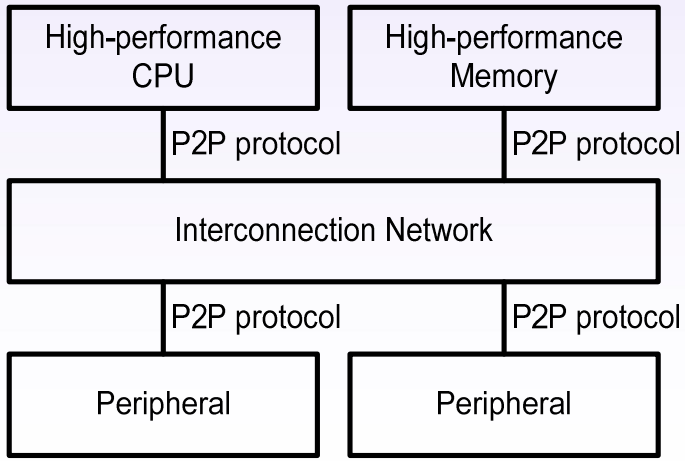
Shared bus

- Complicated protocol
  - Arbitration, multiplexing, decoding, etc.
  - Hierarchical structure
- Ex) AHB, IBM CoreConnect



Point-to-point interconnection

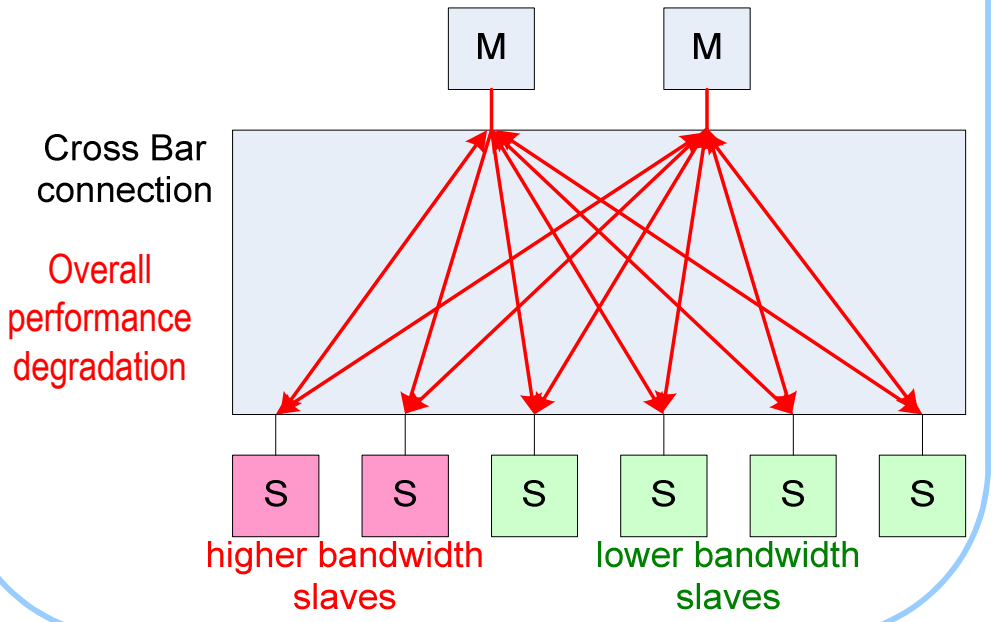
- Core-independent protocol
  - Simple and straightforward
  - Improves reusability of IP cores
  - Improves reliability
- Proprietary interconnection network
- Ex) ARM AXI, Sonics μNetwork, AHB Lite



Hierarchical Bus Architecture

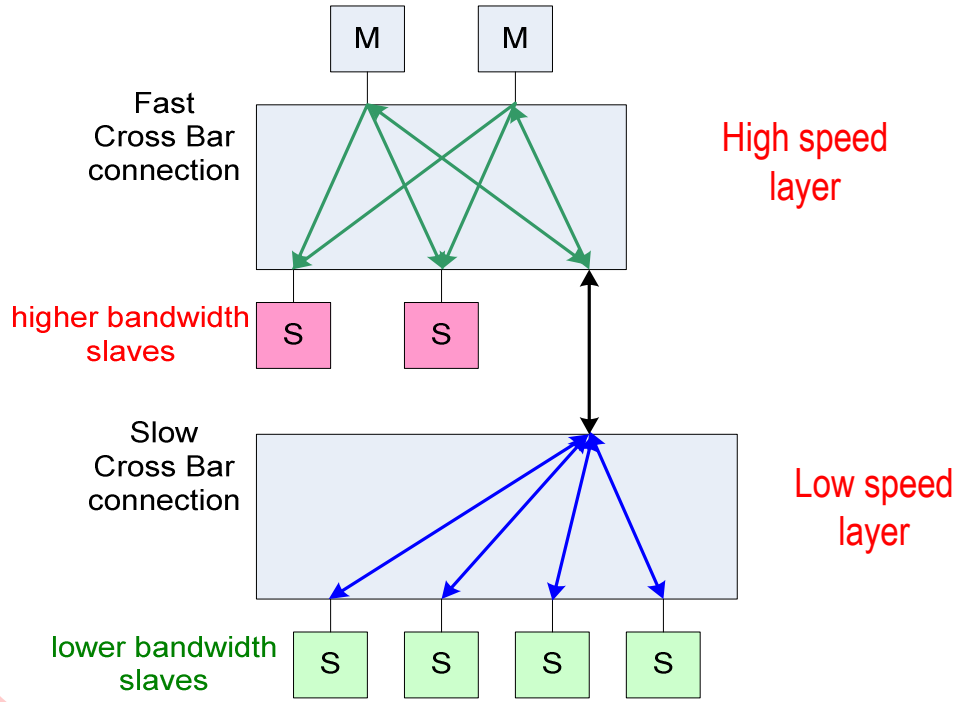
Non-Hierarchical (Flat)

- All slaves are exist in a single layer
- All slaves suffer performance degradation due to
  - the increased complexity of connection logic
  - the slowest peripheral



Hierarchical

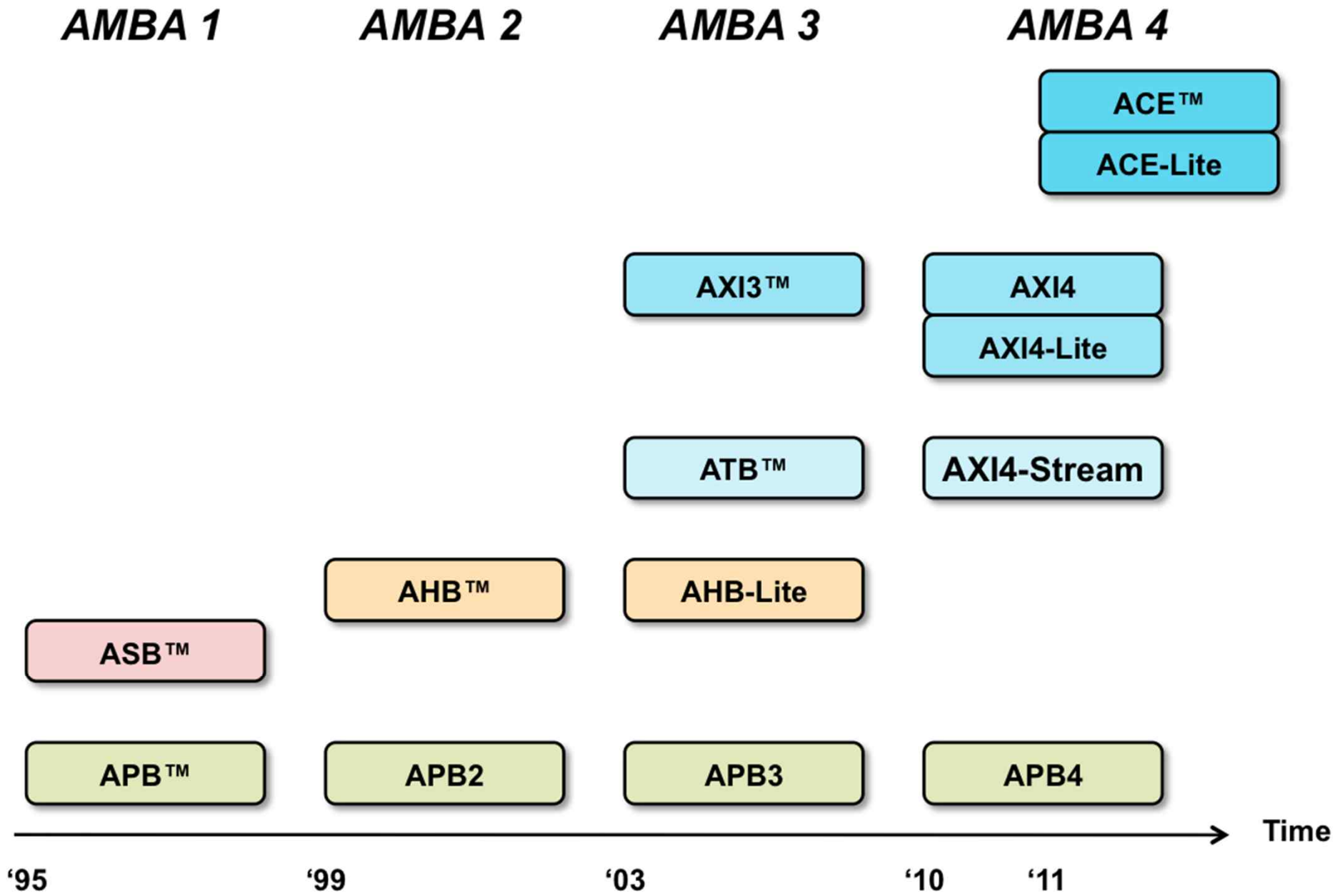
- Classified layers according to required bandwidth
- Delay decreases due to the reduced complexity of connection logic → enhanced speed



AMBA (Advanced Microcontroller Bus Architecture)

- On-chip interconnect specification for SoC
- Promotes re-use by defining a common backbone for SoC modules using standard bus architectures
  - ASB – Advanced System Bus
  - AHB – Advanced High-performance Bus (system backbone)
    - High-performance, high clock freq. modules
    - Processors to on-chip memory, off-chip memory interfaces
  - APB – Advanced Peripheral Bus
    - Low-power peripherals
    - Reduced interface complexity
  - AXI – Advanced eXtensible Interface
  - ACE – AXI Coherency Extension
  - ATB – Advanced Trace Bus

AMBA: Brief History



AMBA: Brief History

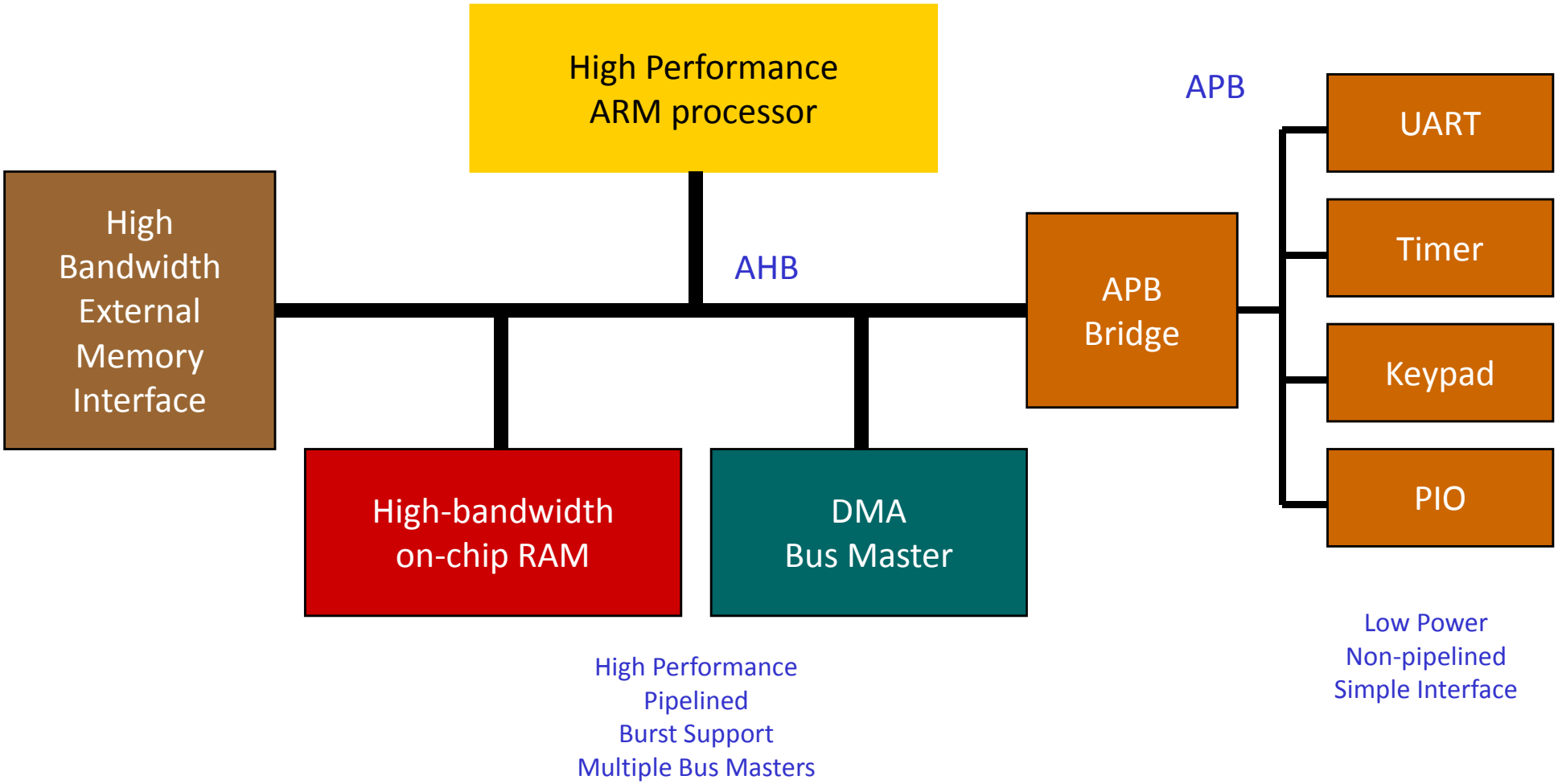
AMBA 1.0	1995	ASB	Advanced System Bus
		APB	Advanced Peripheral Bus
AMBA 2.0	1999	AHB	Advanced High-performance Bus
AMBA 3.0	2003	AXI	Advanced eXtensible Bus
		APB	Addition of wait, error response
AMBA 4.0	2010	AXI4	Some minor changes to AXI - Added support for long burst - Dropped support for write interleaving
		AXI4-Lite	Simplified AXI4 for on-chip device requiring a more powerful interface than APB
		AXI4-Stream	A point-to-point protocol without an address, just data
	2011	ACE	AXI Coherency Extensions
		ACE-Lite	Simplified ACE

Major Component in AMBA System

- Master
  - An independent component that can request to perform the read or write operation to any slave in the system (e.g.) Processor, DSP etc.
- Slave
  - A dependent component on the master. If selected, the read or write operation on its addressable space is performed under the control of the bus master (e.g.) DRAM, SRAM, Peripherals etc.
- Arbiter
  - Arbitrate the bus requests among masters such that only one master is allowed to access the bus (e.g.) fixed priority, round-robin etc.
- Decoder
  - It decides which slave to access by decoding the address from the bus master
- Multiplexer
  - It decides which master or slave access the target IP

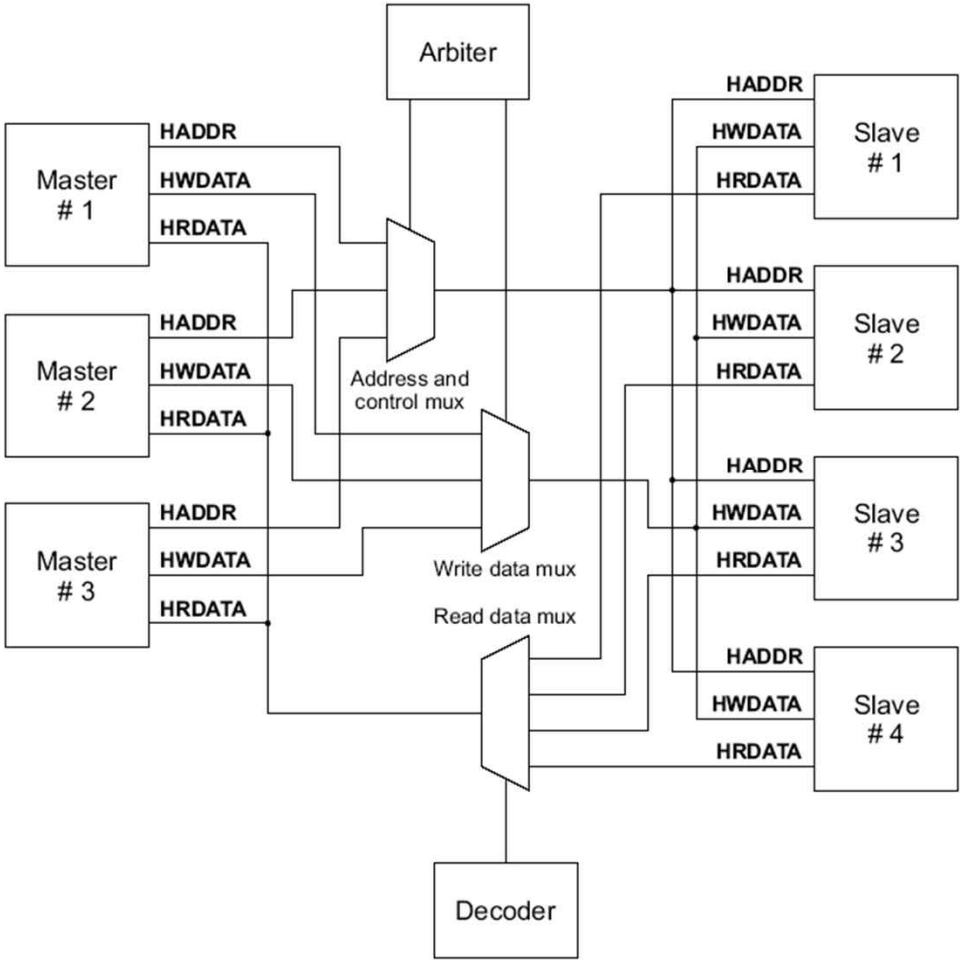
Example AMBA BUS SYSTEM

A typical AMBA-based system



AMBA 2.0 AHB Bus

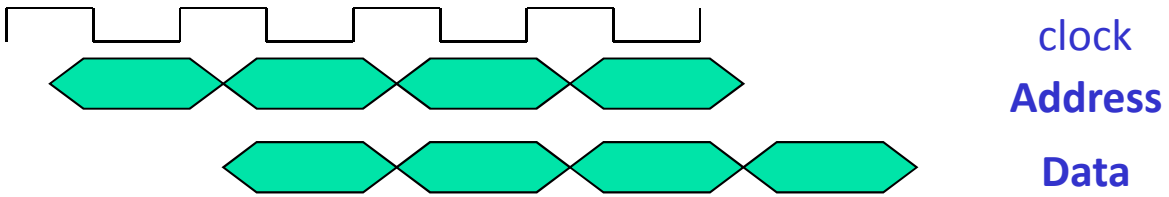
AHB Interconnection



- AMBA AHB is designed to be used with a central multiplexor interconnection scheme
  - Avoids tri-state bus

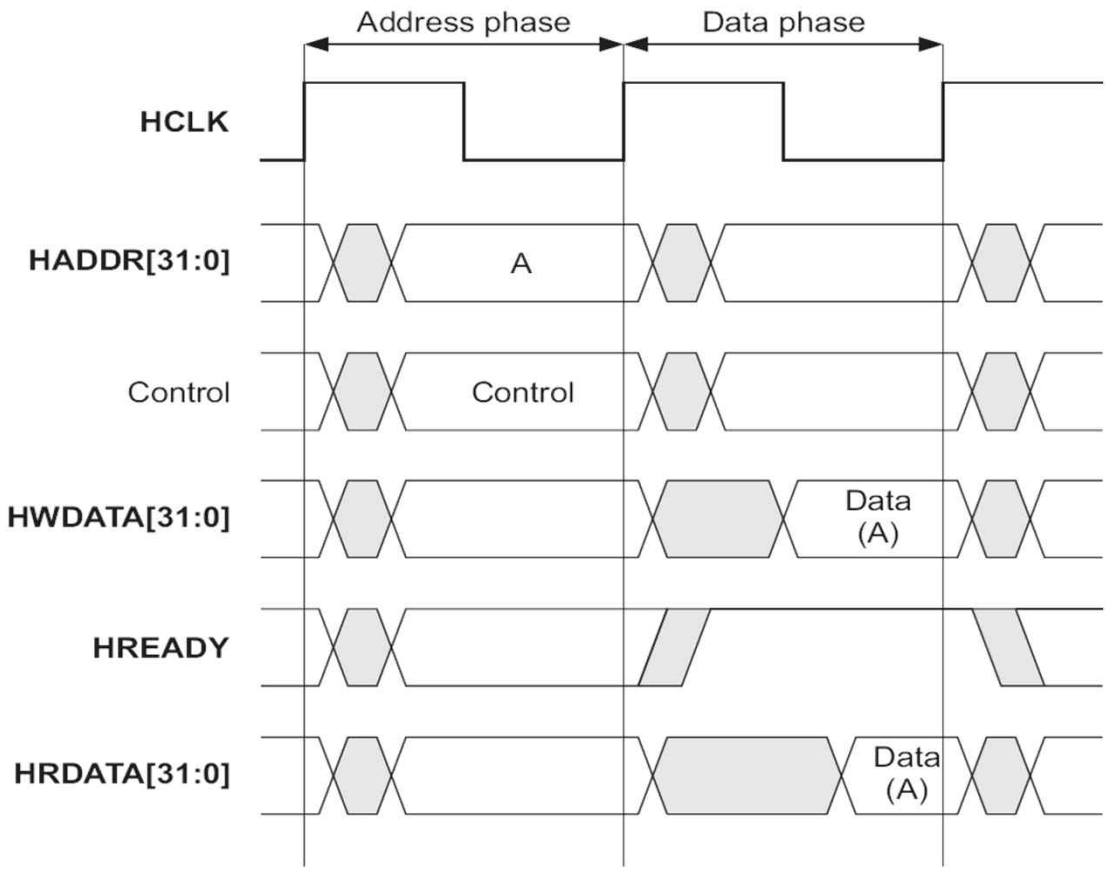
AMBA AHB Overview

- High performance
- Pipelined operation
- Multiple bus masters
  - It is necessary to implement Arbiter
- Single cycle bus master handover
- Burst transfers
- Split transactions
- Single clock edge operation
  - Rising edge
- Non-tri-state implementation
  - Uni-direction vs Bi-direction

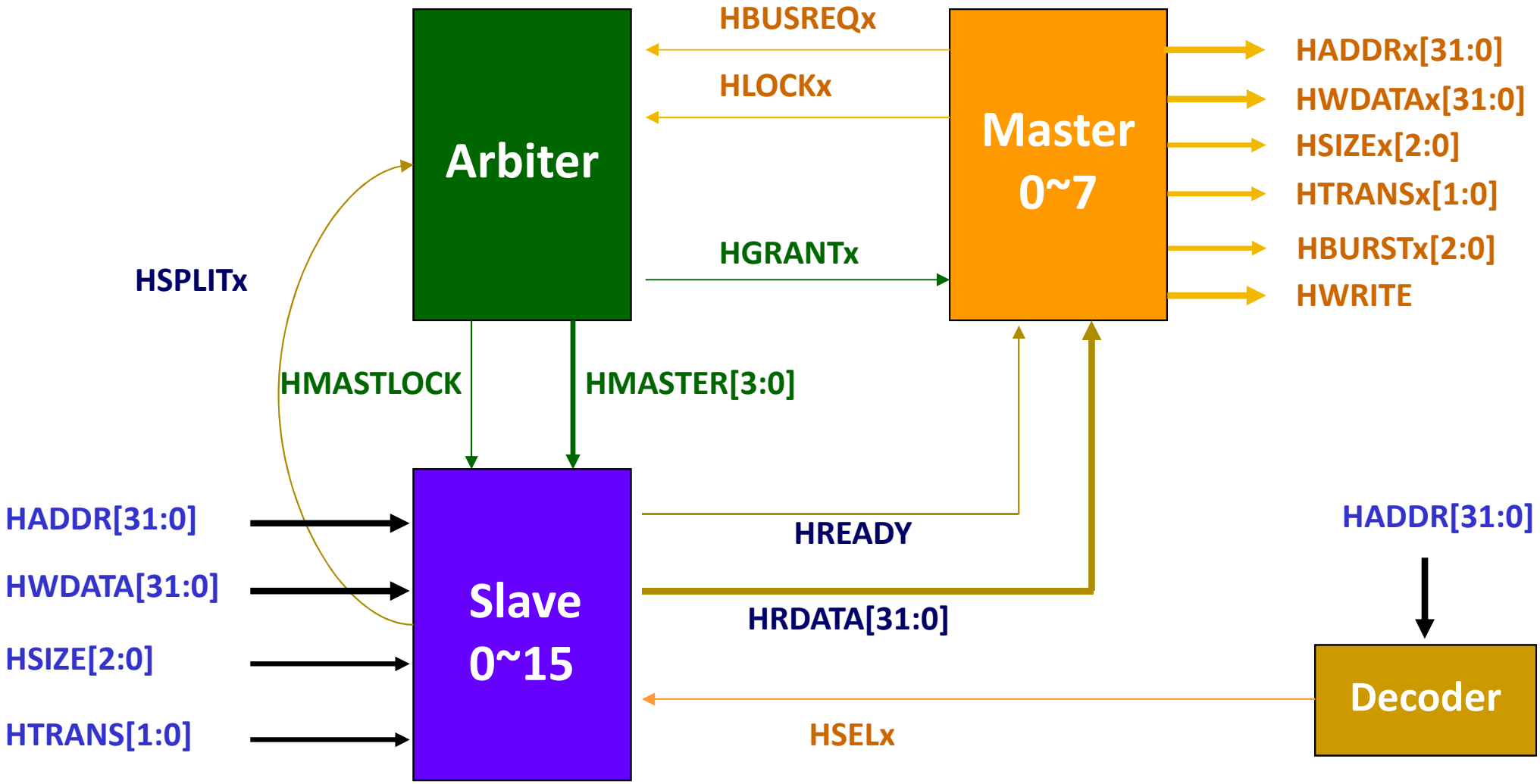


AMBA AHB Overview

- An AHB transfer consists of two distinct sections
  - The **address phase**, which lasts only a single cycle
  - The **data phase**, which may require several cycles
    - This is achieved using the HREADY signal



AMBA Signal Flow



AMBA AHB Signal List (1)

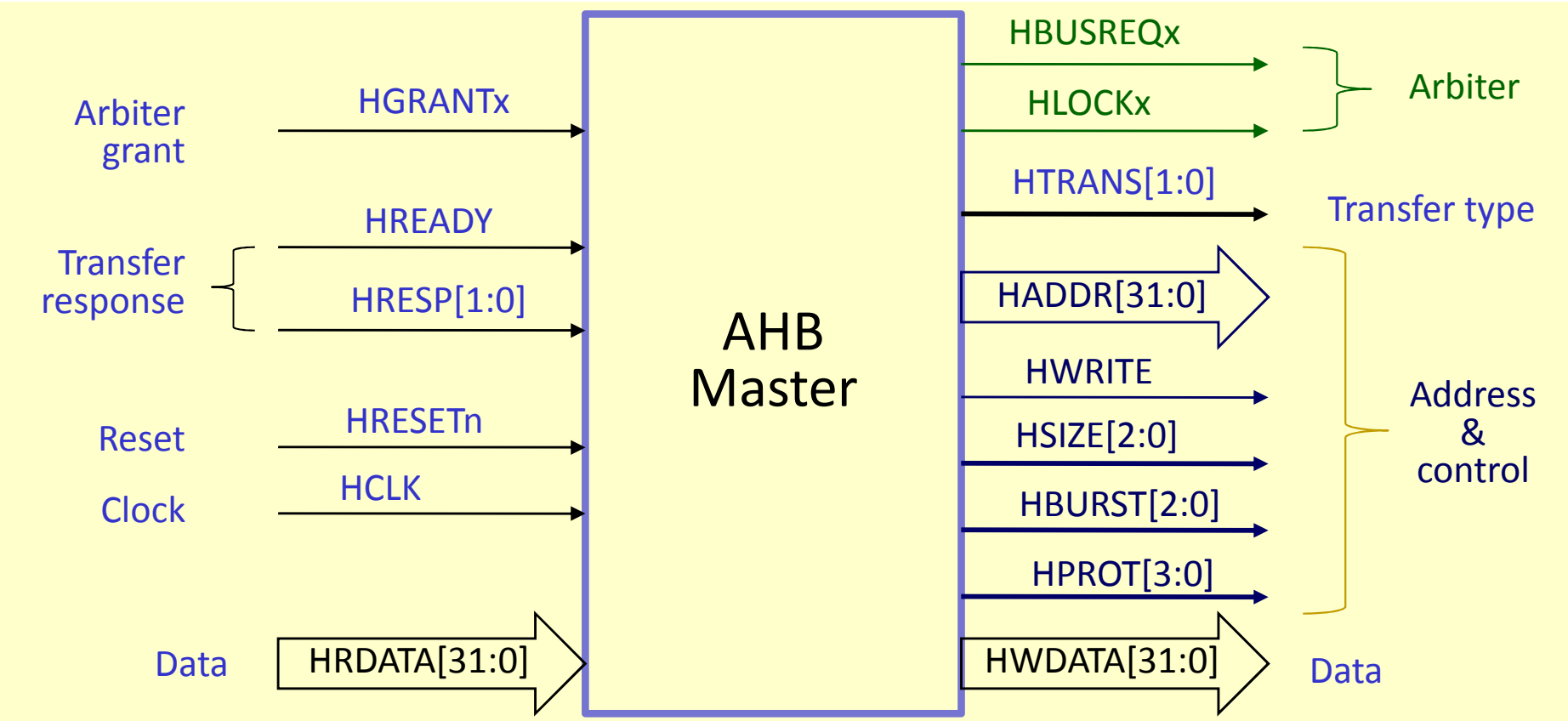
Name	Source	Description
HCLK	Clock source	Bus clock (rising edge)
HRESETn	Reset controller	Reset (Active low)
HADDR[31:0]	Master	Address bus
HTRANS[1:0]	Master	Transfer type (non-sequential, sequential, idle, busy)
HWRITE	Master	Transfer direction (high-write/low-read)
HSIZE[2:0]	Master	Transfer size[(8/16/32 bit), Max. 1024bit]
HBURST[2:0]	Master	Burst type (4/8/16 beat burst, incrementing or wrapping)
HPROT[3:0]	Master	Protection control (opcode fetch or data access) (privileged mode access, user mode access)
HWDATA[31:0]	Master	Write data bus
HSELx	Decoder	Slave select
HRDATA[31:0]	Slave	Read data bus
HREADY	Slave	Transfer done (high-done)
HRESP[1:0]	Slave	Transfer response (okay, error, retry, split)

AMBA AHB Signal List (2)

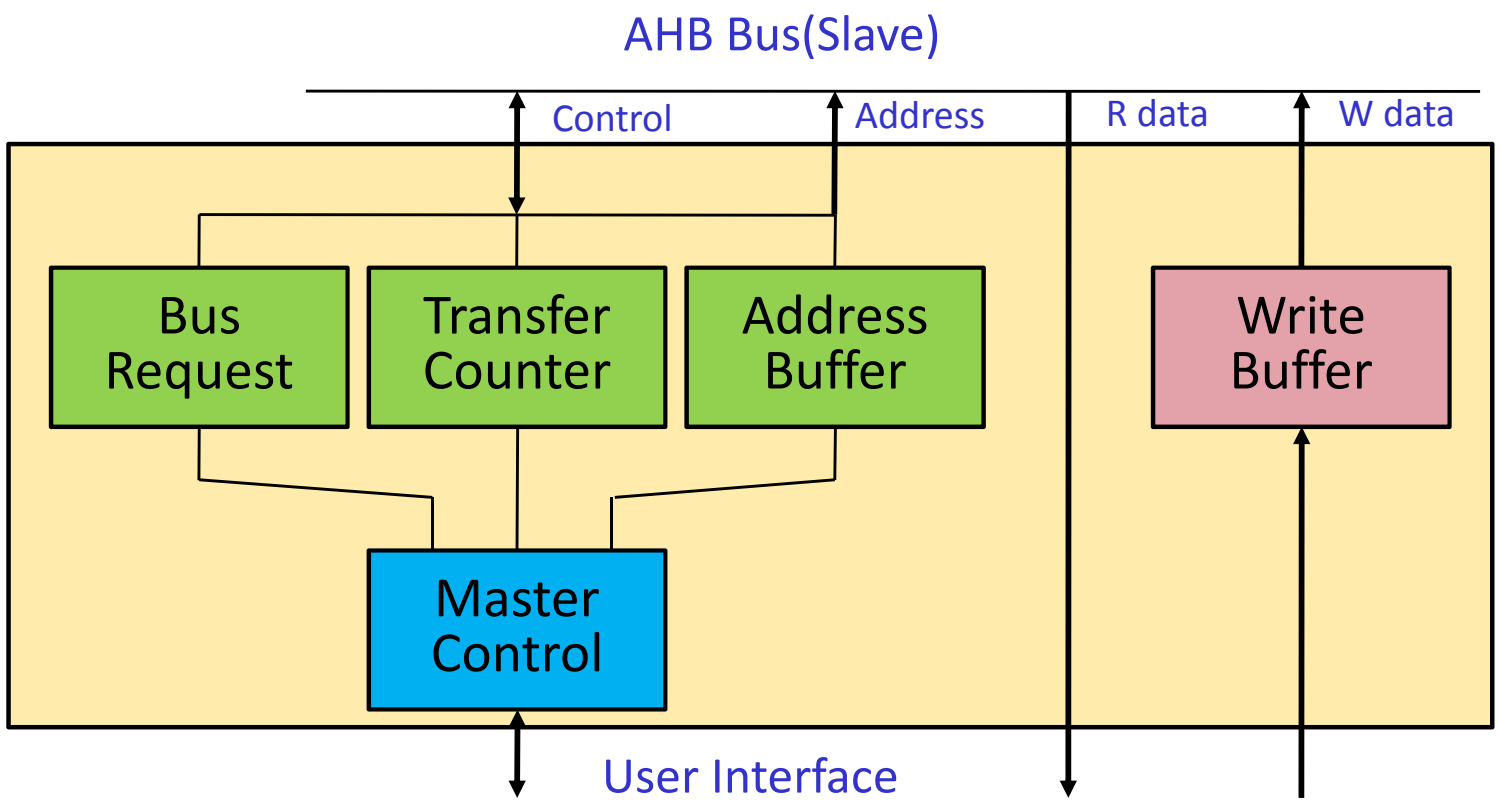
Name	Source	Description
HBUSREQx	Master	Bus request (Max. 16 bus master)
HLOCKx	Master	Locked transfers (Exclusive use of bus when high)
HGRANTx	Arbiter	Bus grant
HMASTER[3:0]	Arbiter	Master number
HMASTLOCK	Arbiter	Locked sequence (The same timing with HMASTER signal)
HSPLITx[15:0]	Slave	Split completion request

AMBA Diagram

- AHB Master
  - An AHB master is able to initiate read and write operation by providing an address and control information
  - Common AHB master
    - Processor, DSP, DMA controller, etc



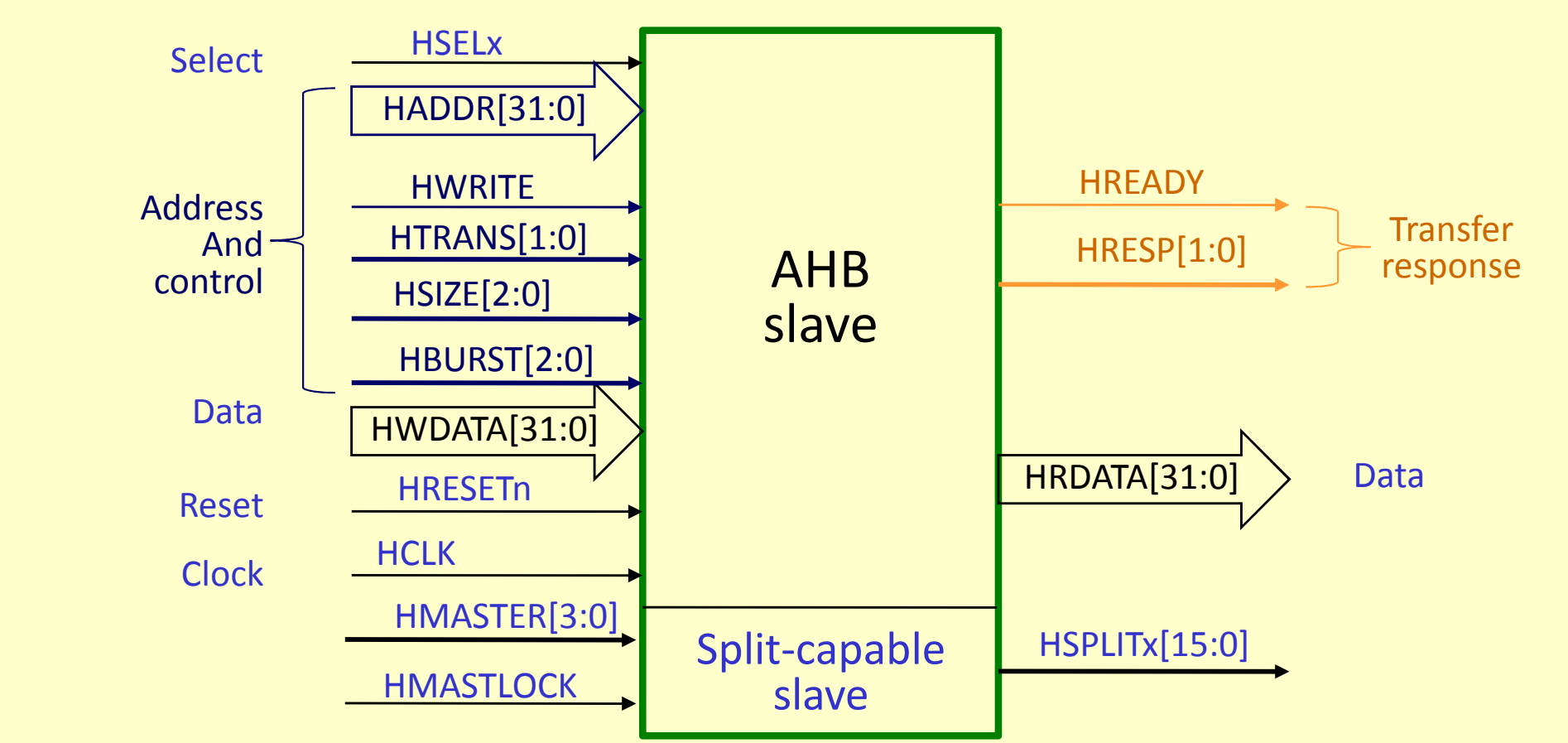
AHB Master Block Diagram



- The AHB master is an interface unit that allows user logic to initiate a data transfer on the AHB
- The bus master is optimized to interface with DMA and PCI bus bridge functions to initiate data transfer on the AHB

AMBA Diagram

- AHB Slave
  - An AHB slave processes read and write operations issued by an AHB master
  - Common AHB slave
    - Memory controller, Peripherals



Control Signal (1/2)

- HWRITE – Transfer direction
  - HIGH : Write using HWDAT[31:0]
  - LOW : Read using HRDATA[31:0]
- HSIZE[2:0] – Transfer size

HSIZE[2:0]	Size	Description
000	8 bits	Byte
001	16 bits	half word
010	32 bits	Word
011	64 bits	
100	128 bits	4-word line
101	256 bits	8-word line
110	512 bits	
111	1024 bits	

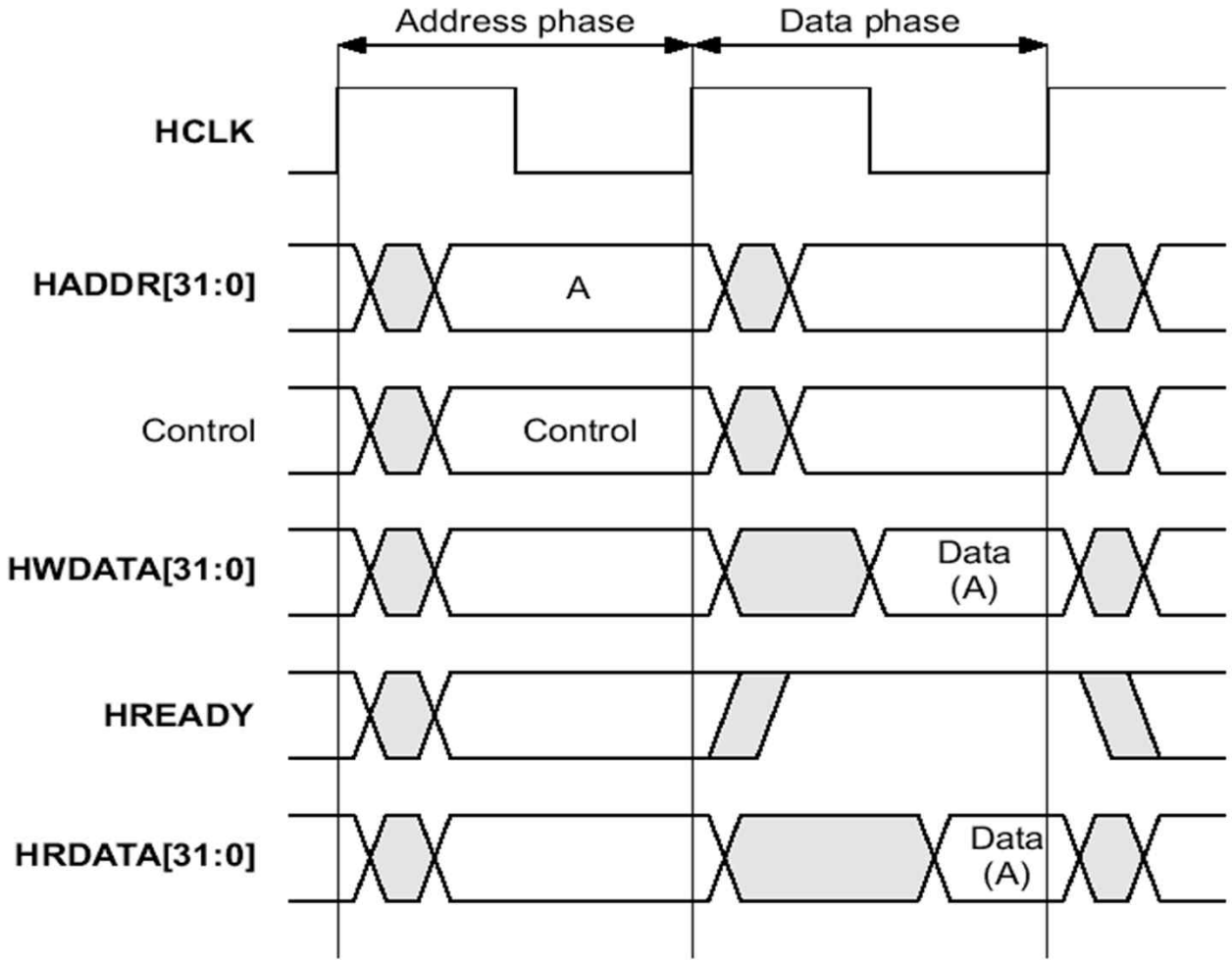
Control Signal (2/2)

- HPROT[3:0] – Protection control (Indicating opcode fetch, data access, privileged mode access, and user mode)

HPROT[3] cacheable	HPROT[2] bufferable	HPROT[1] privileged	HPROT[0] Data/opcode	Description
-	-	-	0	Opcode fetch
-	-	-	1	Data access
-	-	0	-	User access
-	-	1	-	Privileged access
-	0	-	-	Not bufferable
-	1	-	-	Bufferable
0	-	-	-	Not cacheable
1	-	-	-	Cacheable

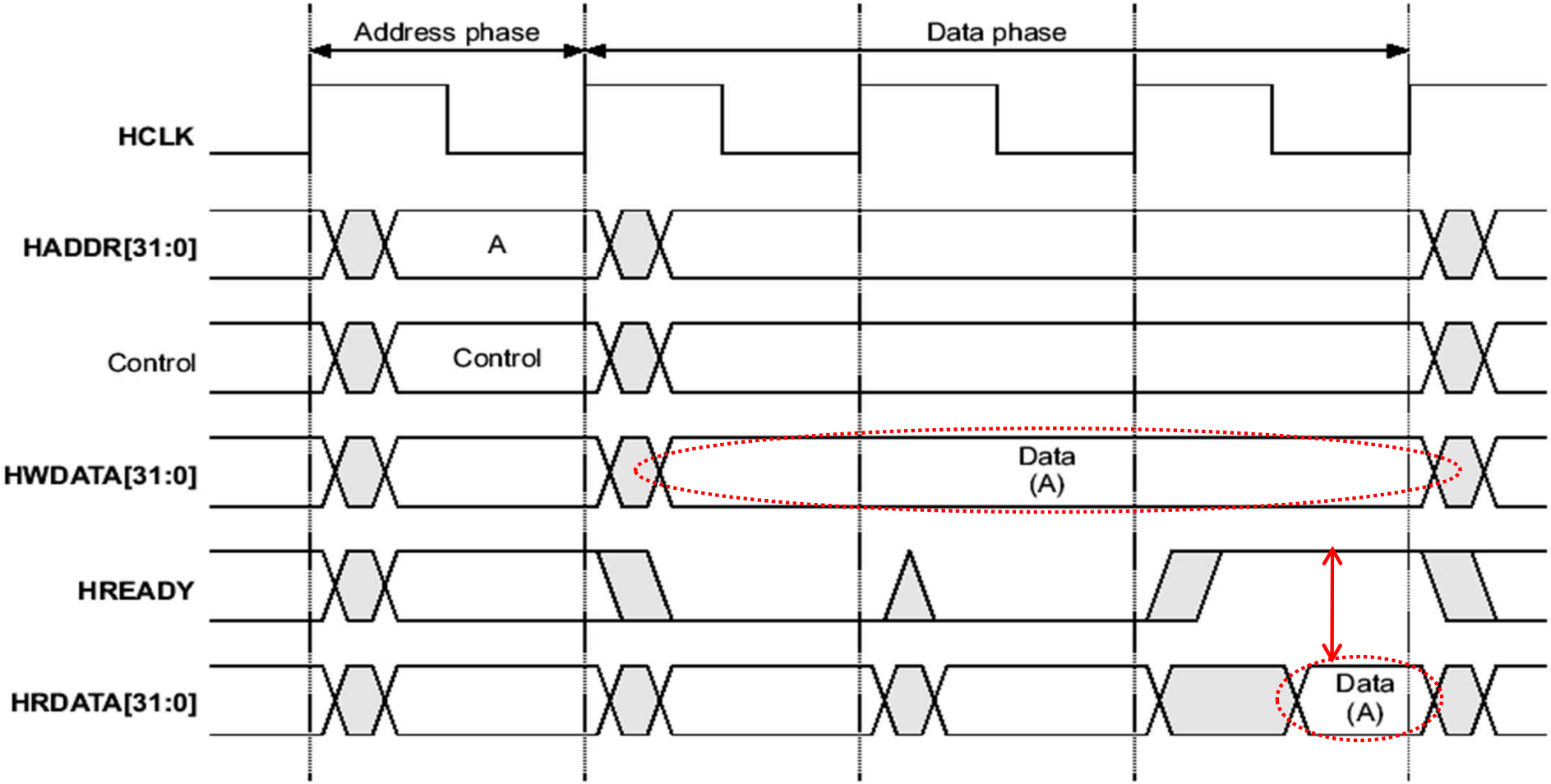
Basic Transfer (1/10)

- Address phase: Only single cycle
- Data phase can be extended to multi-cycle by using HREADY signal



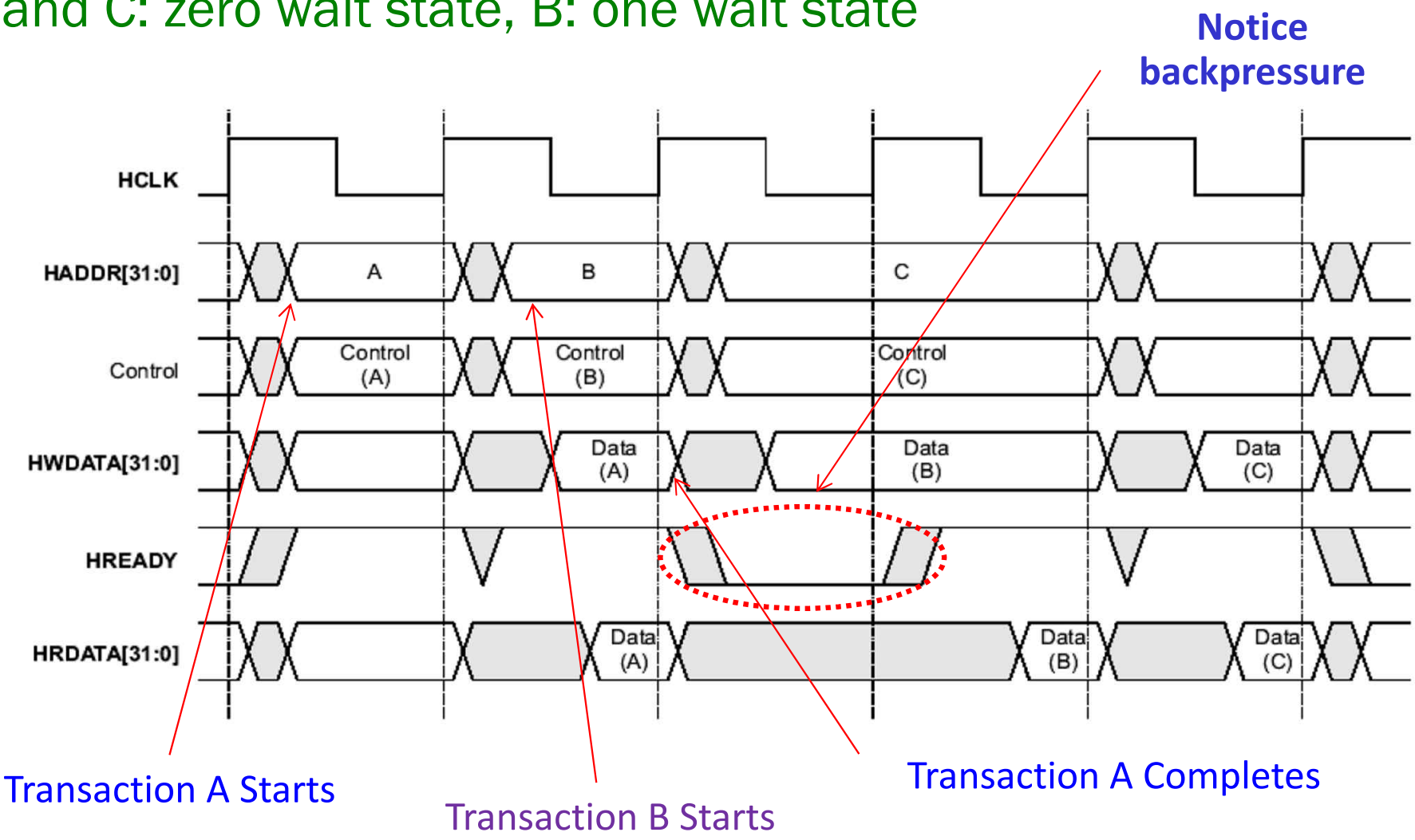
Basic Transfer (2/10)

- Example: data transfer cycle extension between objects with different operating clock frequencies



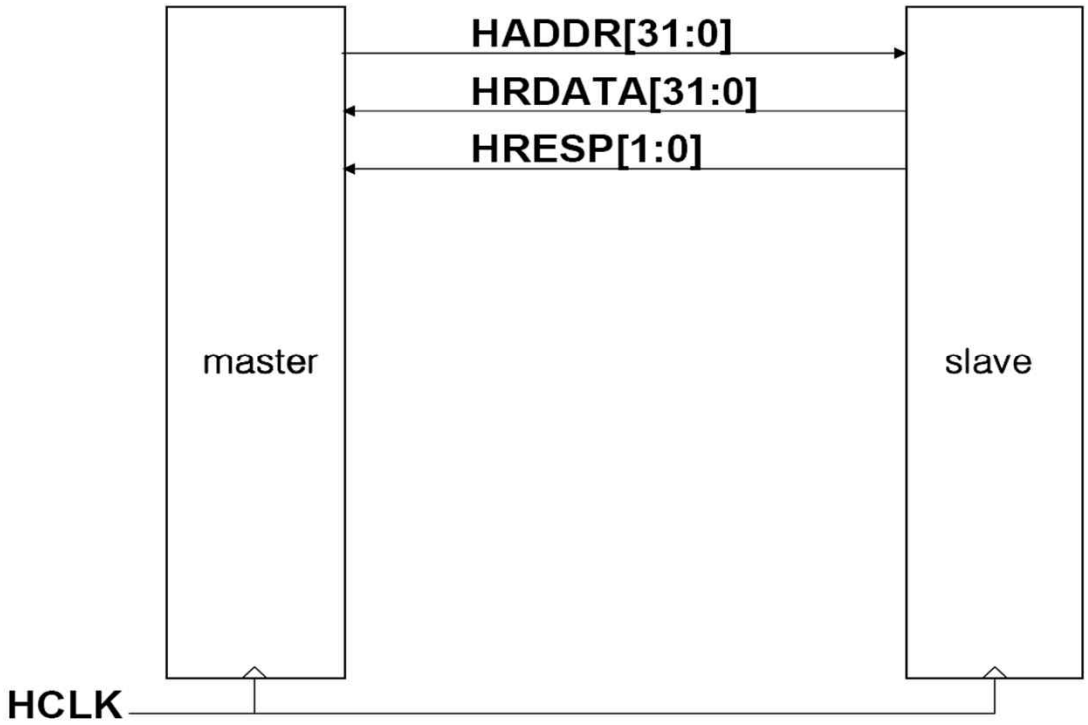
Basic Transfer (3/10)

- Multiple data transfer with non-contiguous address
- A and C: zero wait state, B: one wait state



Basic Transfer (4/10)

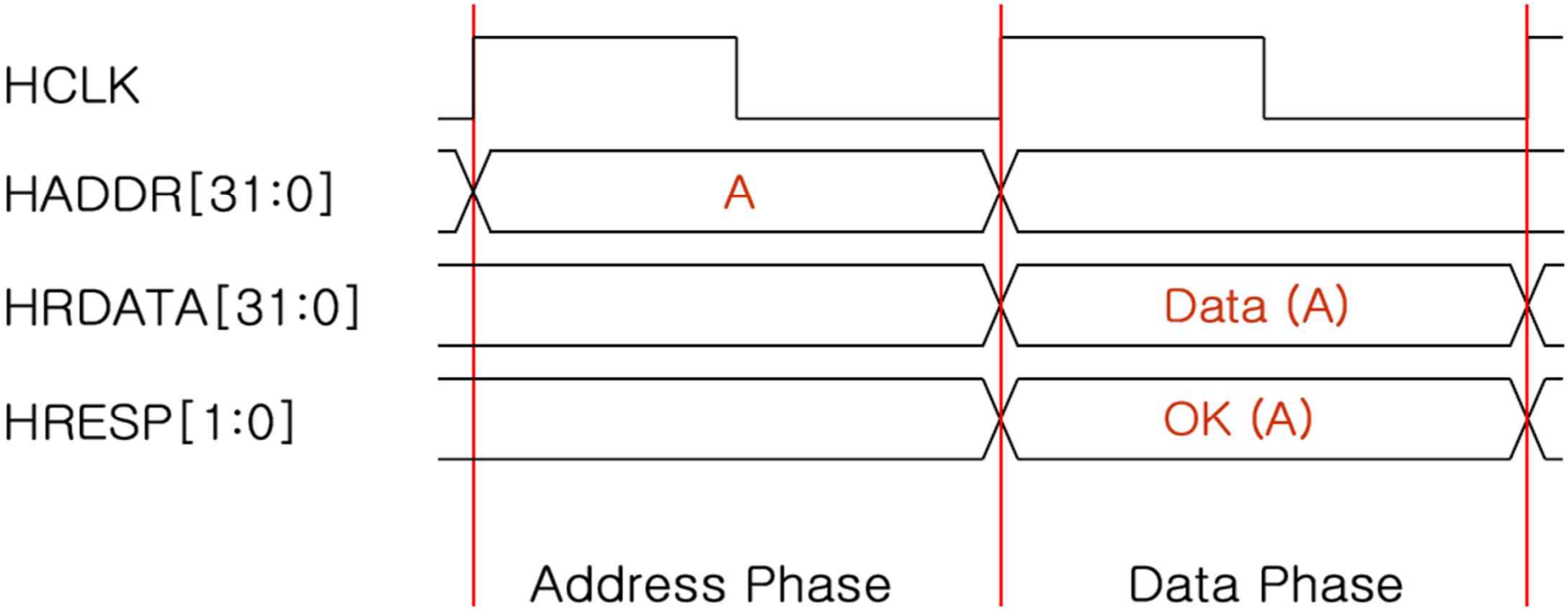
- Basic Signals for Read Txn



Name	Source	Description
HADDR[31:0]	Master	Address bus
HRDATA[31:0]	Slave	Read data bus
HRESP	Slave	Transfer response (okay, error, retry, split)

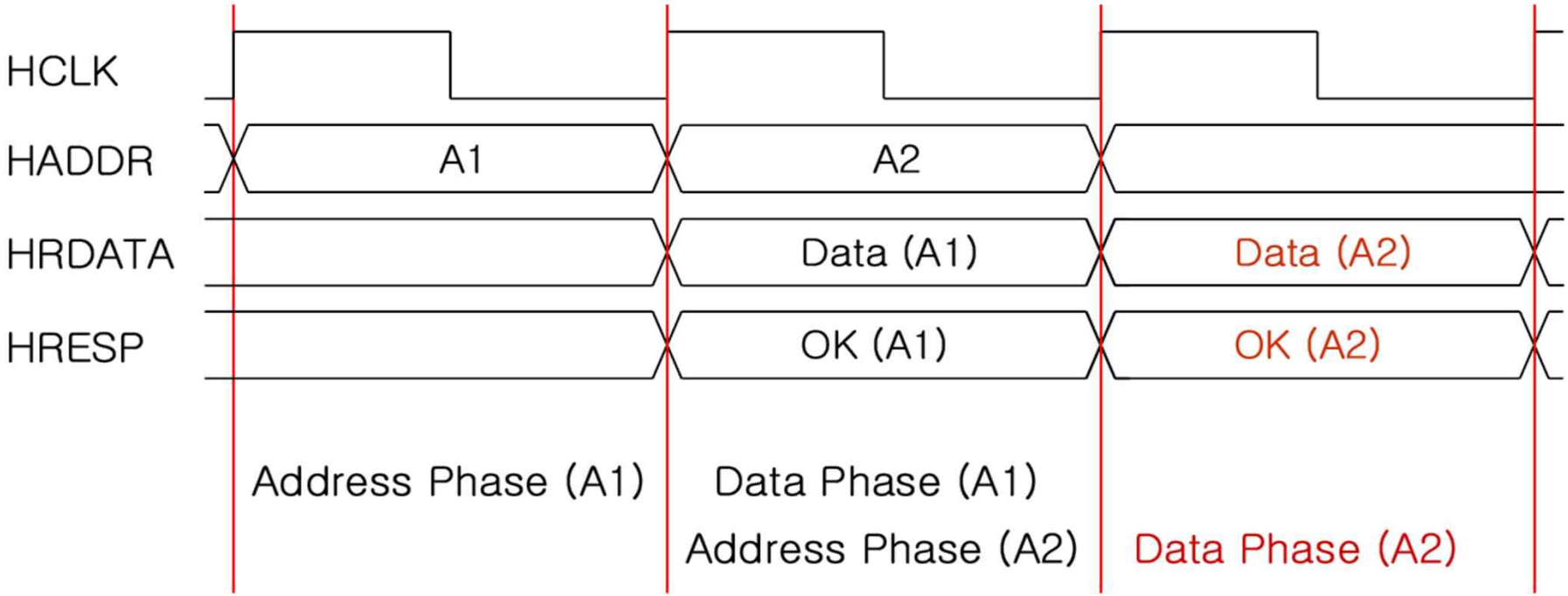
Basic Transfer (5/10)

- Operation for a Read Txn



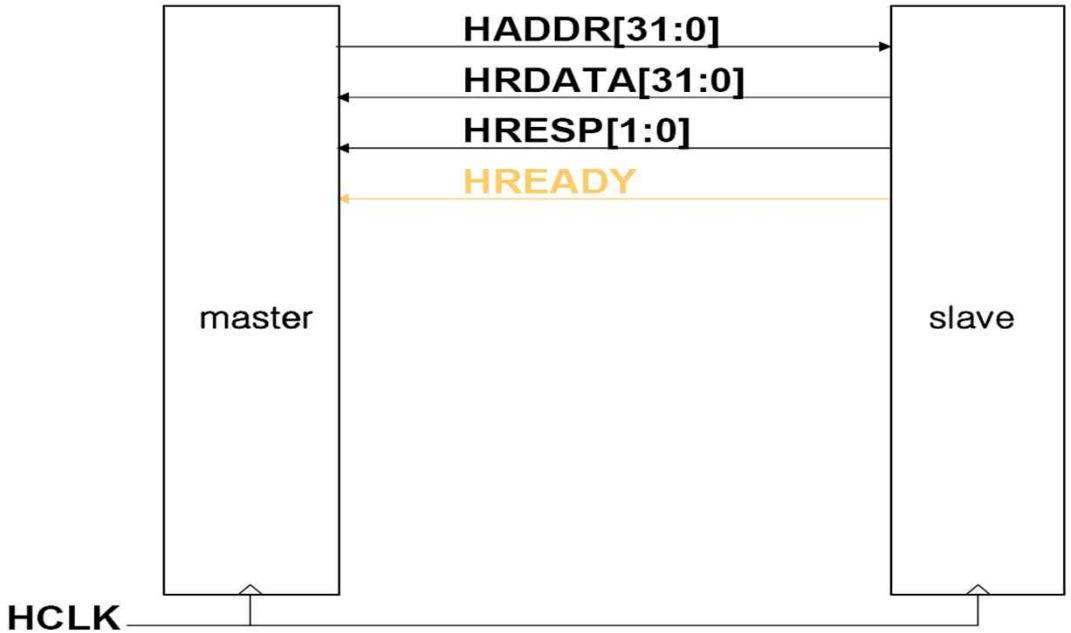
Basic Transfer (6/10)

- Pipelined Operation



Basic Transfer (7/10)

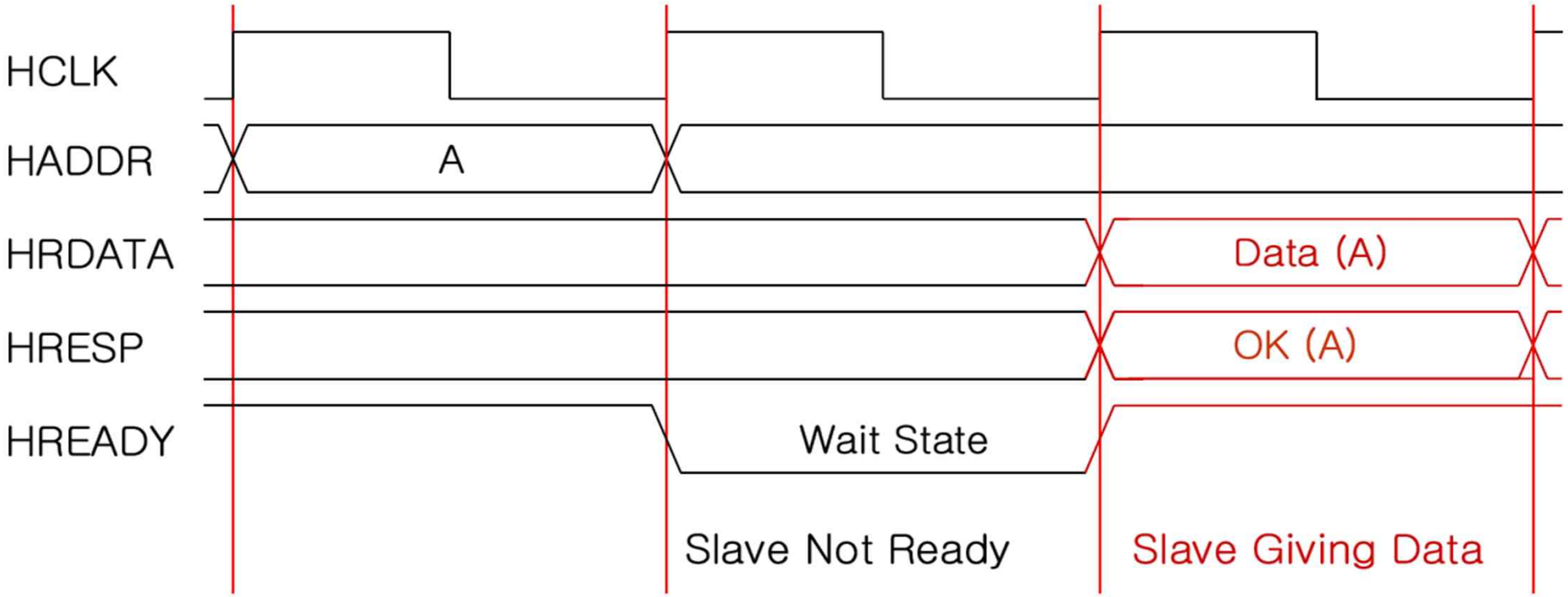
- HREADY for a Slow Slave



Name	Source	Description
HADDR[31:0]	Master	Address bus
HRDATA[31:0]	Slave	Read data bus
HREADY	Slave	Transfer done (high-done)
HRESP	Slave	Transfer response (okay, error, retry, split)

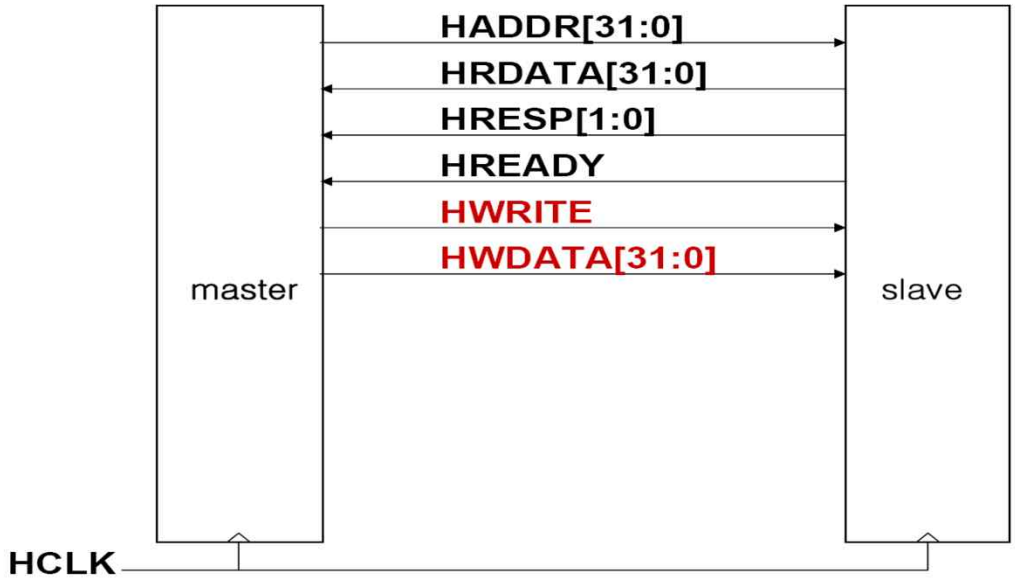
Basic Transfer (8/10)

- Wait State Insertion



Basic Transfer (9/10)

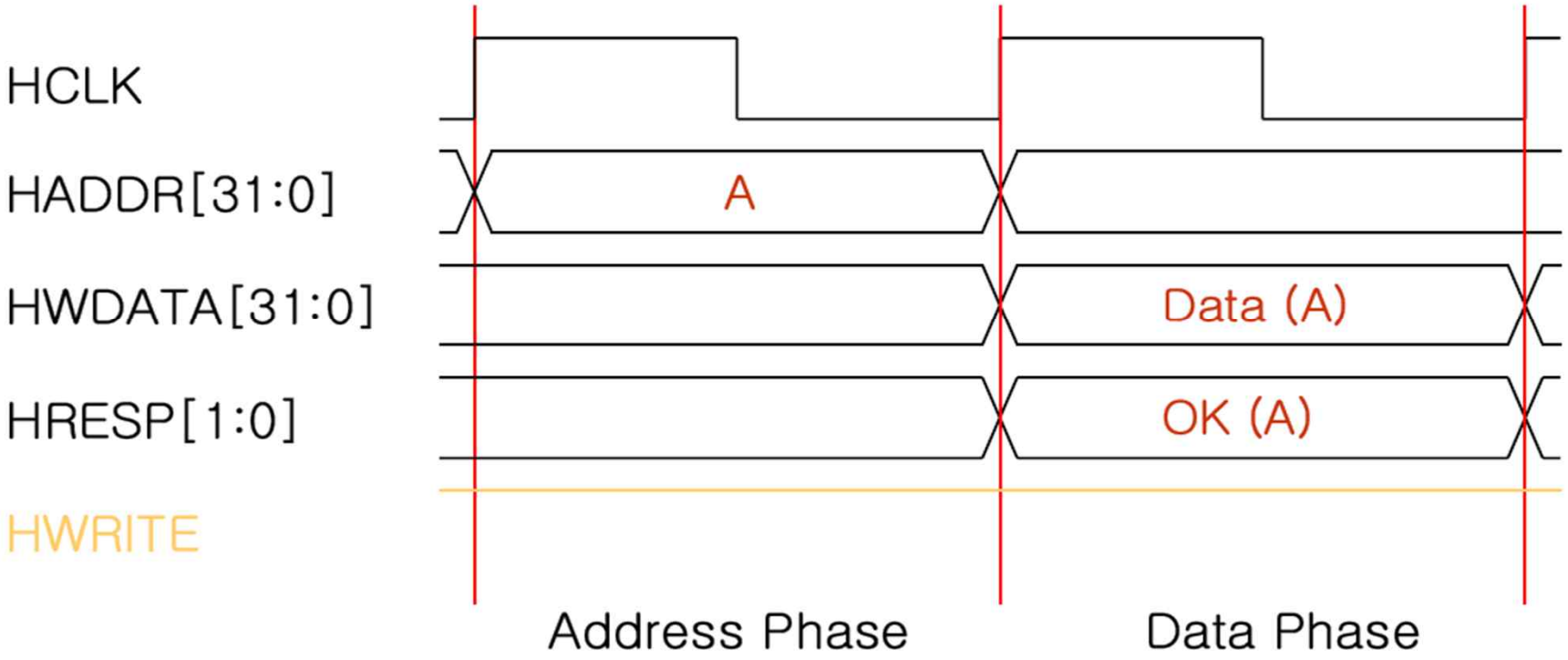
- Operation for Write Txn



Name	Source	Description
HADDR[31:0]	Master	Address bus
HWRITE	Master	Transfer direction (high-write/low-read)
HWDATA[31:0]	Master	Write data bus
HRDATA[31:0]	Slave	Read data bus
HREADY	Slave	Transfer done (high-done)
HRESP	Slave	Transfer response (okay, error, retry, split)

Basic Transfer (10/10)

- Operation for a Write Txn



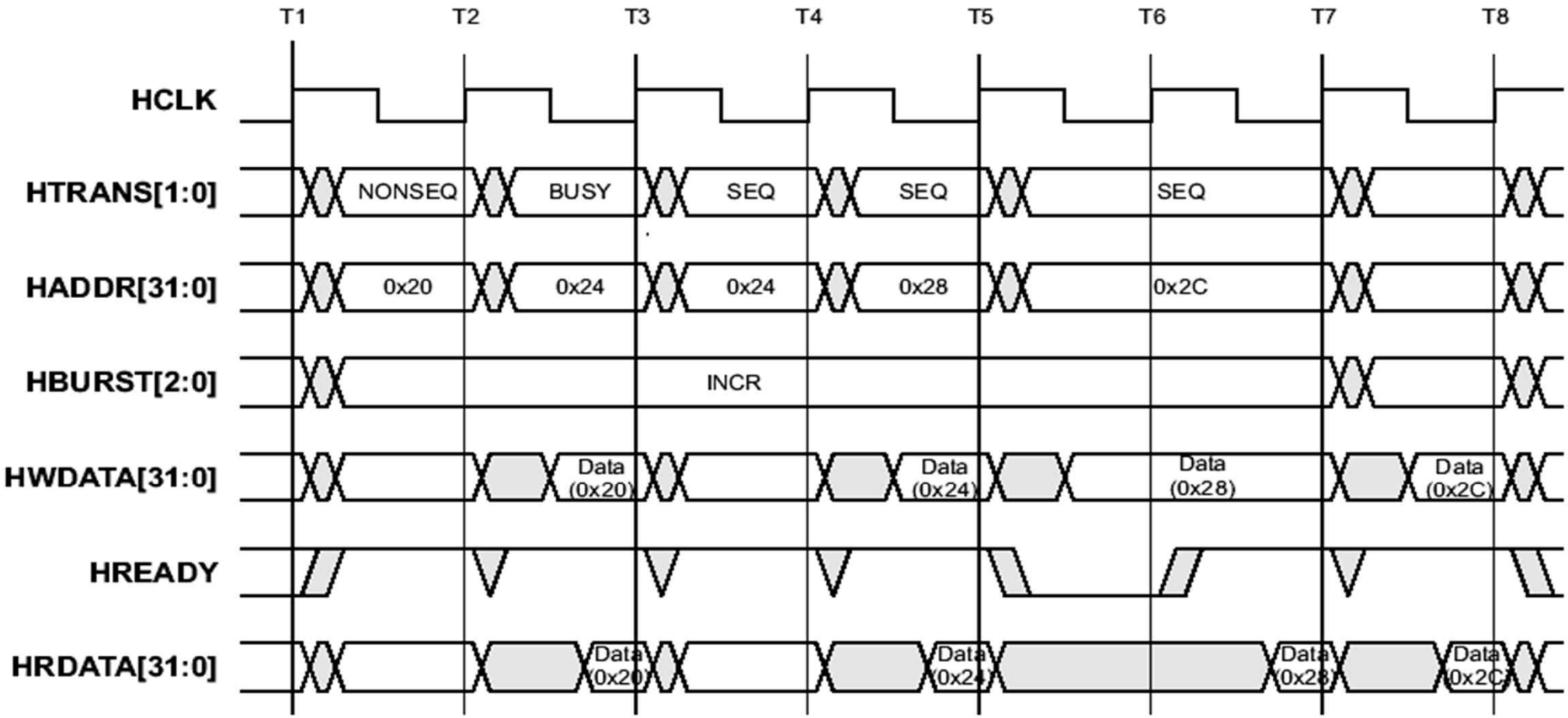
Transfer Type (1/2)

- All data transfer are processed with HTRANS[1:0]

HTRANS[1:0]	Type	Description
00	IDLE	Indicates that no data transfer is required. The IDLE transfer type is used when a bus master is granted the bus, but does not wish to perform a data transfer. Slaves must always provide a zero wait state OKAY response to IDLE transfers and the transfer should be ignored by the slave.
01	BUSY	The BUSY transfer type allows bus masters to insert IDLE cycles in the middle of bursts of transfers. This transfer type indicates that the bus master is continuing with a burst of transfers, but the next transfer cannot take place immediately. When a master uses the BUSY transfer type the address and control signals must reflect the next transfer in the burst. The transfer should be ignored by the slave. Slaves must always provide a zero wait state OKAY response, in the same way that they respond to IDLE transfers.
10	NONSEQ	Indicates the first transfer of a burst or a single transfer. The address and control signals are unrelated to the previous transfer. Single transfers on the bus are treated as bursts of one and therefore the transfer type is NONSEQUENTIAL.
11	SEQ	The remaining transfers in a burst are SEQUENTIAL and the address is related to the previous transfer. The control information is identical to the previous transfer. The address is equal to the address of the previous transfer plus the size (in bytes). In the case of a wrapping burst the address of the transfer wraps at the address boundary equal to the size (in bytes) multiplied by the number of beats in the transfer (either 4, 8 or 16).

Transfer Type (2/2)

- Transfer type example



Burst Operation

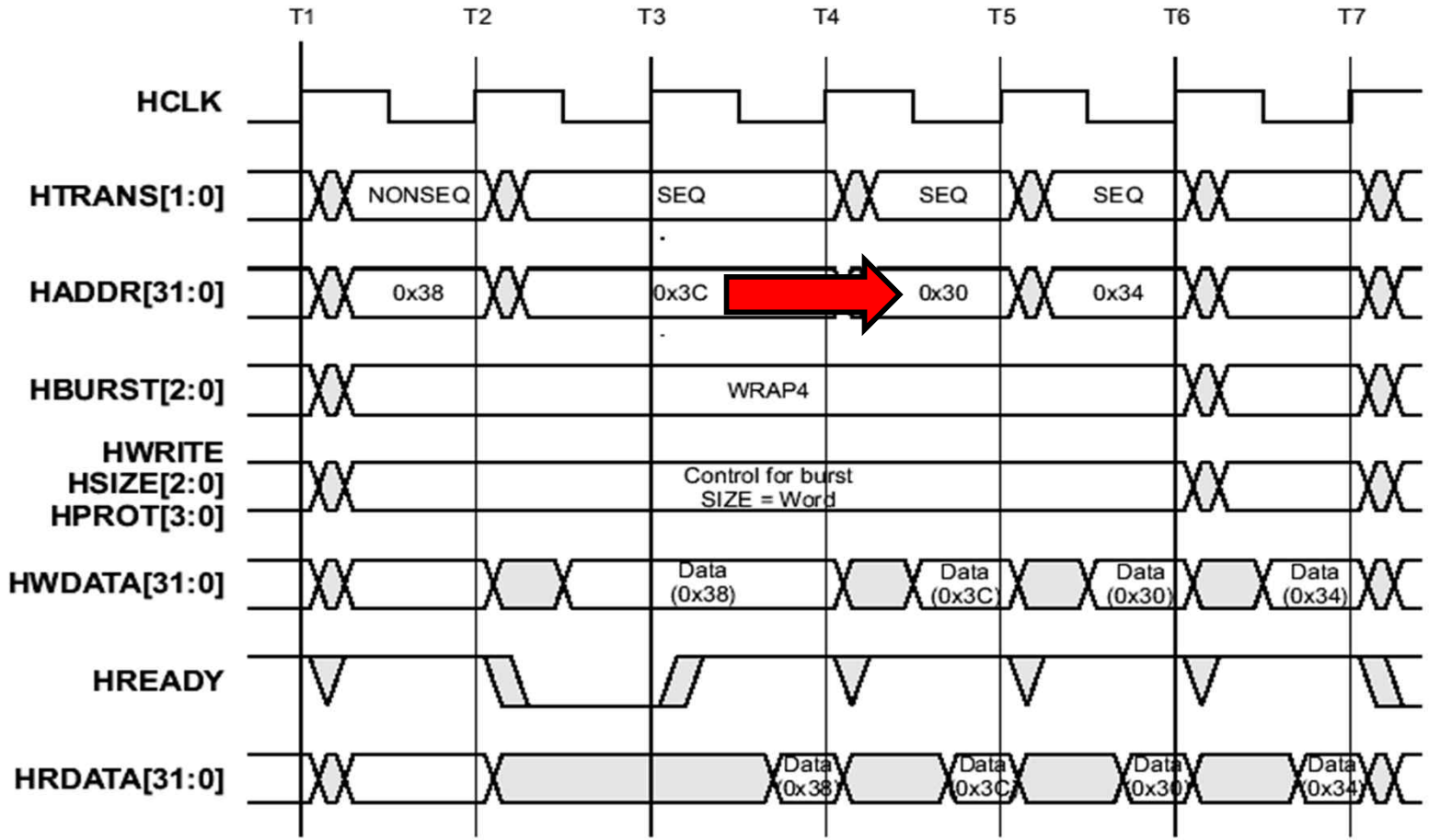
Burst signal encoding

HBURST[2:0]	Type	Description
000	SINGLE	Single transfer
001	INCR	Incrementing burst of unspecified length
010	WRAP4	4-beat wrapping burst
011	INCR4	4-beat incrementing burst
100	WRAP8	8-beat wrapping burst
101	INCR8	8-beat incrementing burst
110	WRAP16	16-beat wrapping burst
111	INCR16	16-beat incrementing burst

Burst Mode Example (1/5)

- The address wraps if it reaches the address boundary
  - In this case, the address boundary is 0x30 ~ 0x3C

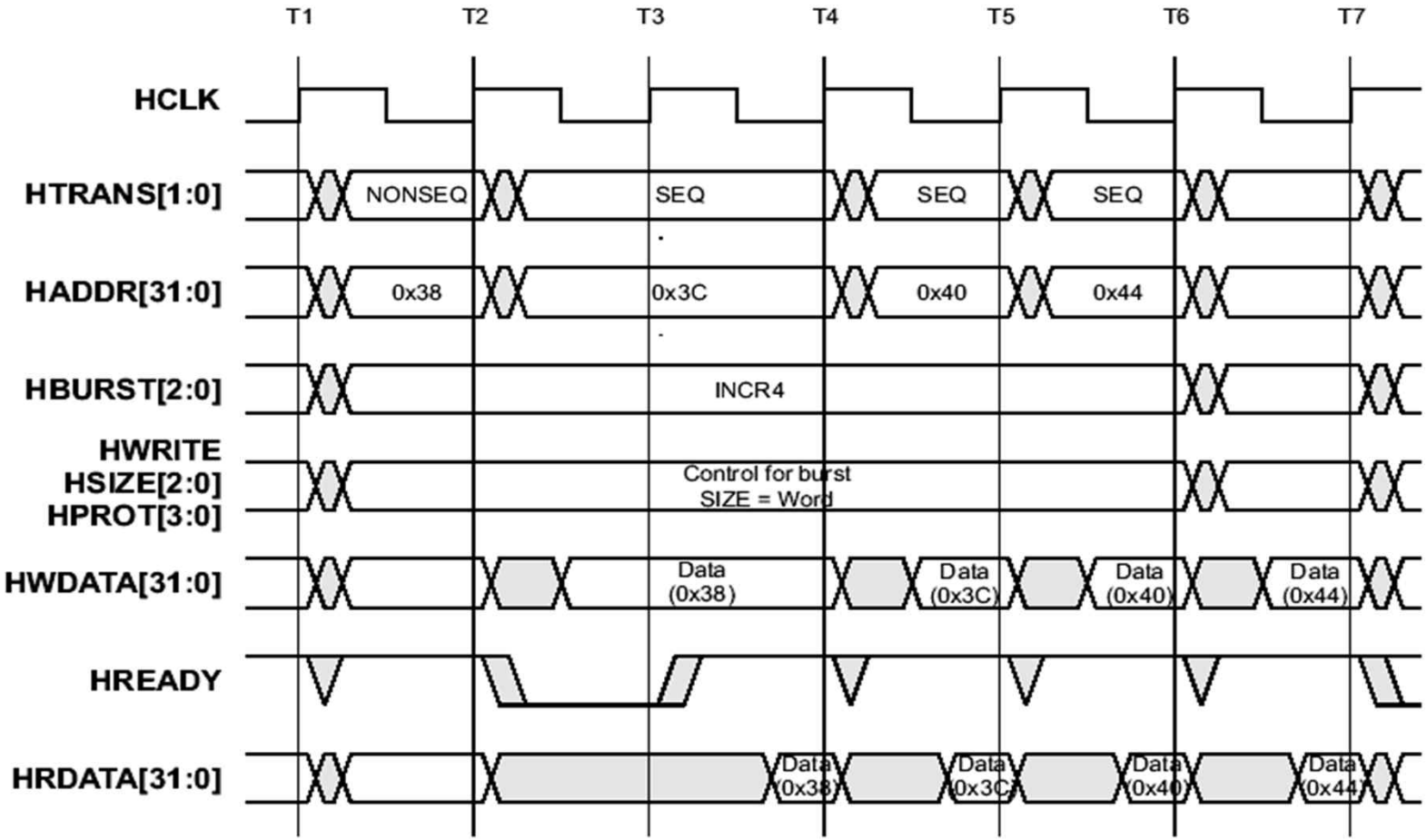
Four-beat wrapping burst



Burst Mode Example (2/5)

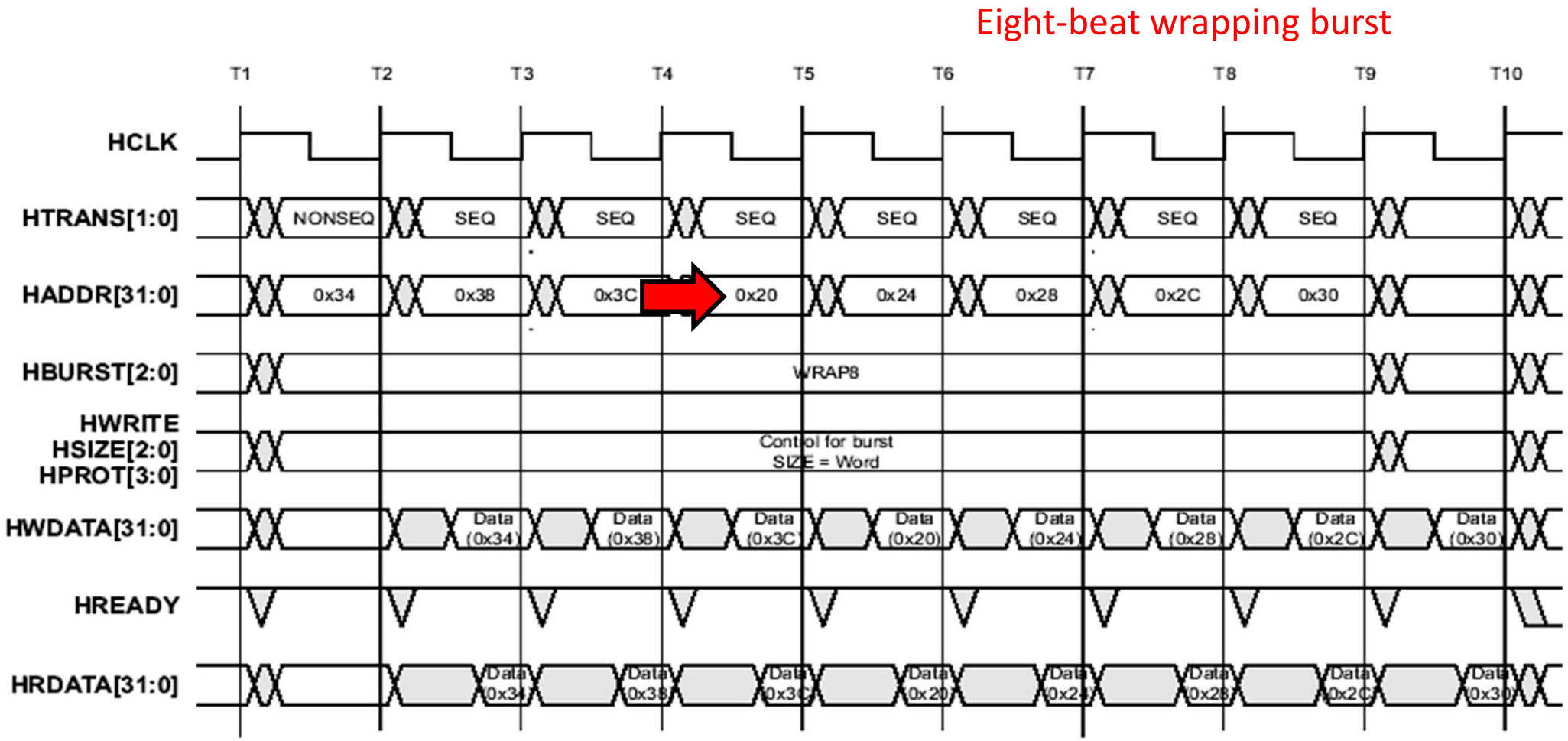
- Sequentially increasing address
  - In this case, the address increases by 4 because SIZE = Word

Four-beat incrementing burst



Burst Mode Example (3/5)

- The address wrap if it reaches the address boundary
  - In this case, the address boundary is 0x20 ~ 0x3C



Answer to the Question

- In the case of a wrapping burst the address of the transfer wraps at the address boundary equal to the size (in bytes) multiplied by the number of beats in the transfer (either 4, 8 or 16).

  So if 'Eight-beat wrapping burst' with size (specified in HSIZE[2:0]) of 4Byte (Word=32bit) has 32 Byte(=8x4) address boundary. '32 Byte address boundary' means that only 5 LSB bits out of HADDR[31:0], that is, HADDR[4:0] can be changed.

  Therefore if the start address is 0x34, then next address changes as follows:

  0x34 → 0x38 → 0x3C → 0x20 → 0x24 → 0x28 → 0x2C → 0x30.

- The question in the middle of today's lecture is: If the start address begins with 0x24, how the address bus will be changed? The answer is:

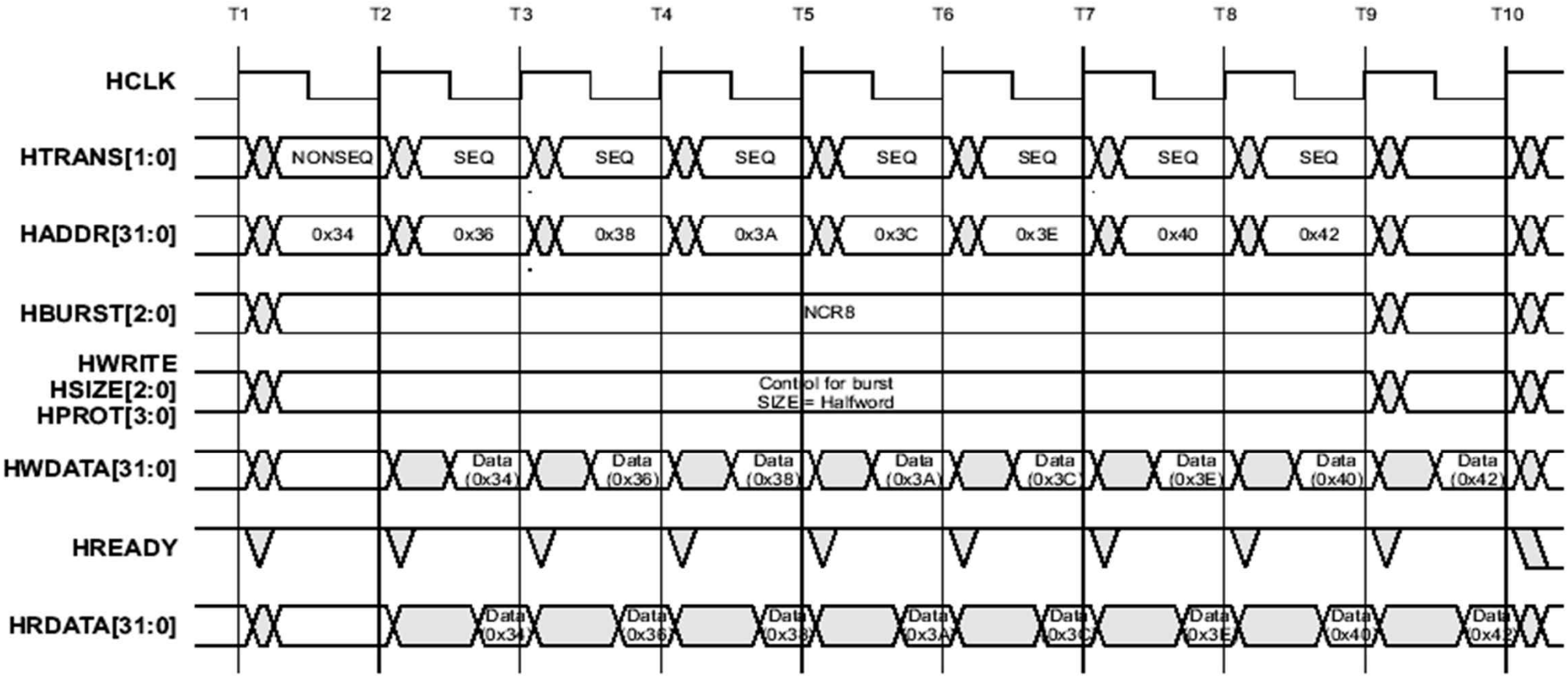
  0x24 → 0x28 → 0x2C → 0x30 → 0x34 → 0x38 → 0x3C → 0x20.

  Because only 5 LSB bits are allowed to be changed due to the fact that address boundary is 32 Byte.

Burst Mode Example (4/5)

- Sequentially increasing address
  - In this case, the address increases by 2 because SIZE = halfword

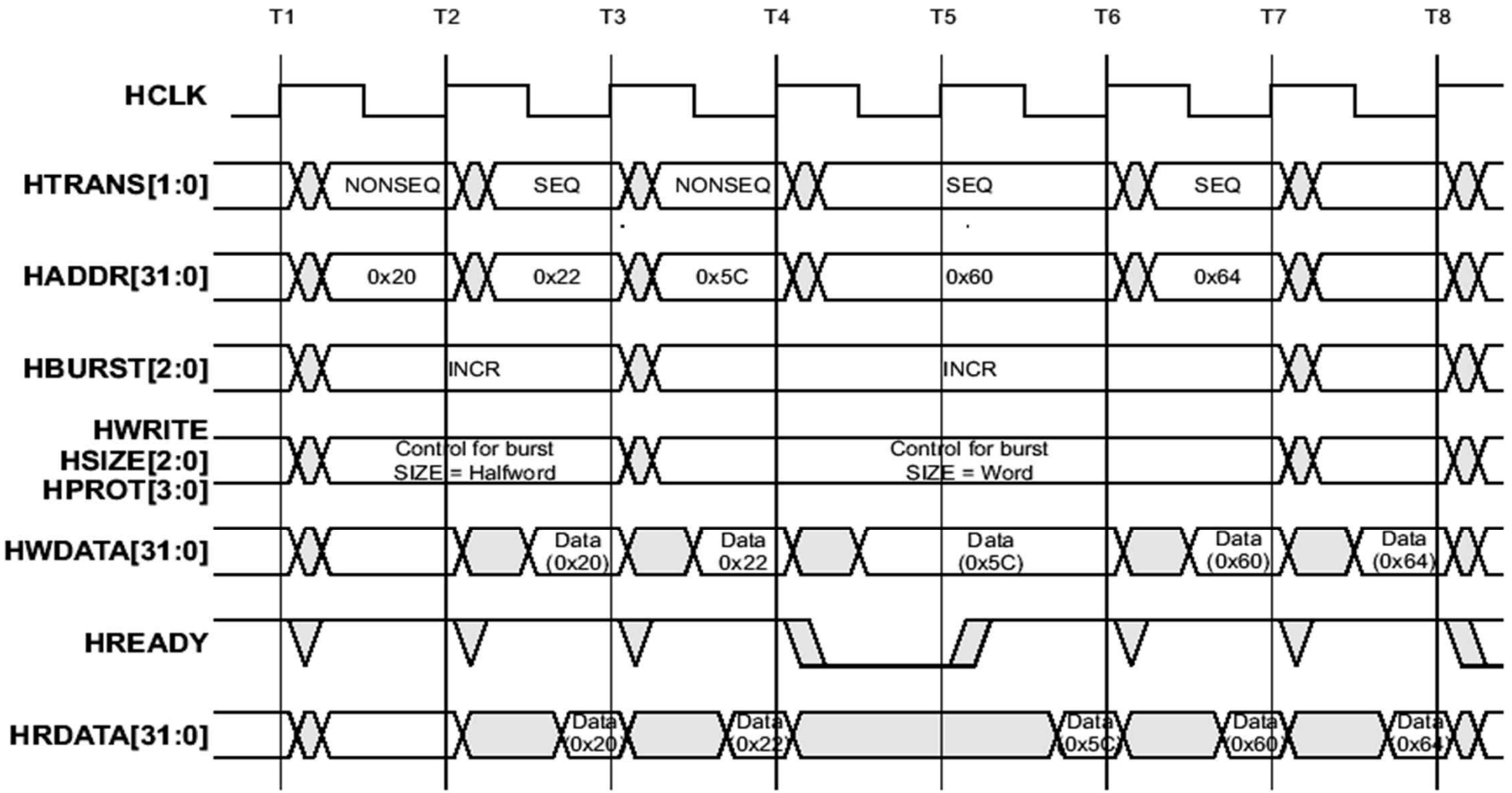
Eight-beat incrementing burst



Burst Mode Example (5/5)

- Incrementing burst of unspecified length

Undefined-length bursts

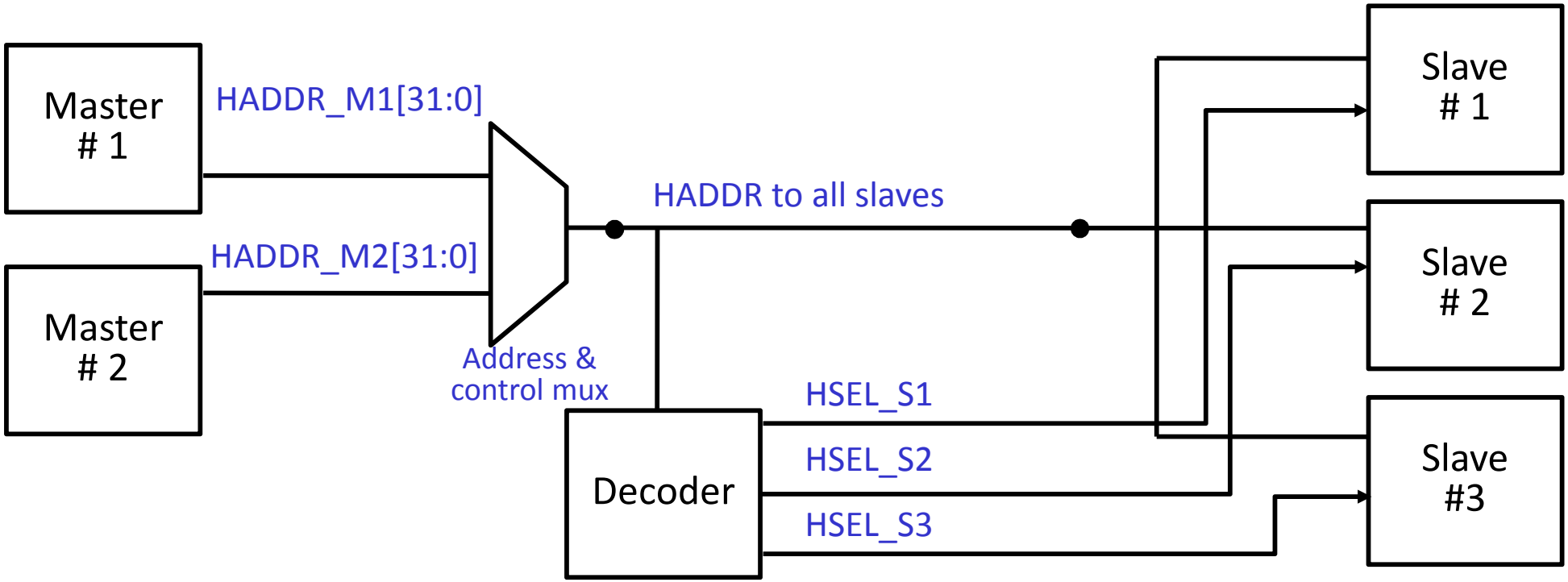


Early Burst Termination

- There are certain circumstances when a burst will not be allowed to complete and therefore it is important that any slave design which makes use of the burst information can take the correct course of action if the burst is terminated early
- The slave can determine when a burst has terminated early by monitoring the HTRANS signals and ensuring that after the start of the burst every transfer is labeled as SEQUENTIAL or BUSY
- If a NONSEQUENTIAL or IDLE transfer occurs then this indicates that a new burst has started and therefore the previous one must have been terminated
- If a bus master cannot complete a burst because it loses ownership of the bus then it must rebuild the burst appropriately when it next gains access to the bus
  - For example, if a master has only completed one beat of a four-beat burst then it must use an undefined-length burst to perform the remaining three transfers

Address Decoding

- Address decoding Process-



- Decoder – used to provide a select signal, HSELx, for each slave on the bus
- Select signal – Implemented by high-order address signals
- Address boundary – The minimum address space that can be allocated to a single slave is 1KB

Slave Transfer Responses

- The role of slaves
  - Whenever a slave is accessed it must provide a response which indicates the status of the transfer
  - The HREADY signal is used to extend the transfer and this works in combination with the response signals, HRESP[1:0], which provide the status of the transfer
- The slave can complete the transfer in a number of ways:
  - complete the transfer immediately
  - insert one or more wait states to allow time to complete the transfer
  - signal an error to indicate that the transfer has failed
  - delay the completion of the transfer, but allow the master and slave to back off the bus, leaving it available for other transfers
- Transfer response
  - OKAY with HREADY indicates the transfer has completed successfully
  - ERROR response: Occurs when accessing wrong addresses
  - RETRY: Indicates transfer has not yet completed, so the bus master should retry the transfer
  - SPLIT: The transfer has not yet completed successfully, so the bus master must retry the transfer when it is next granted access to the bus

Data Buses (1/2)

- AHB does not specify the required endianness, so, it is important that all masters and slaves on the bus are of the same endianness.

- Active byte lanes for a 32-bit little-endian data bus -

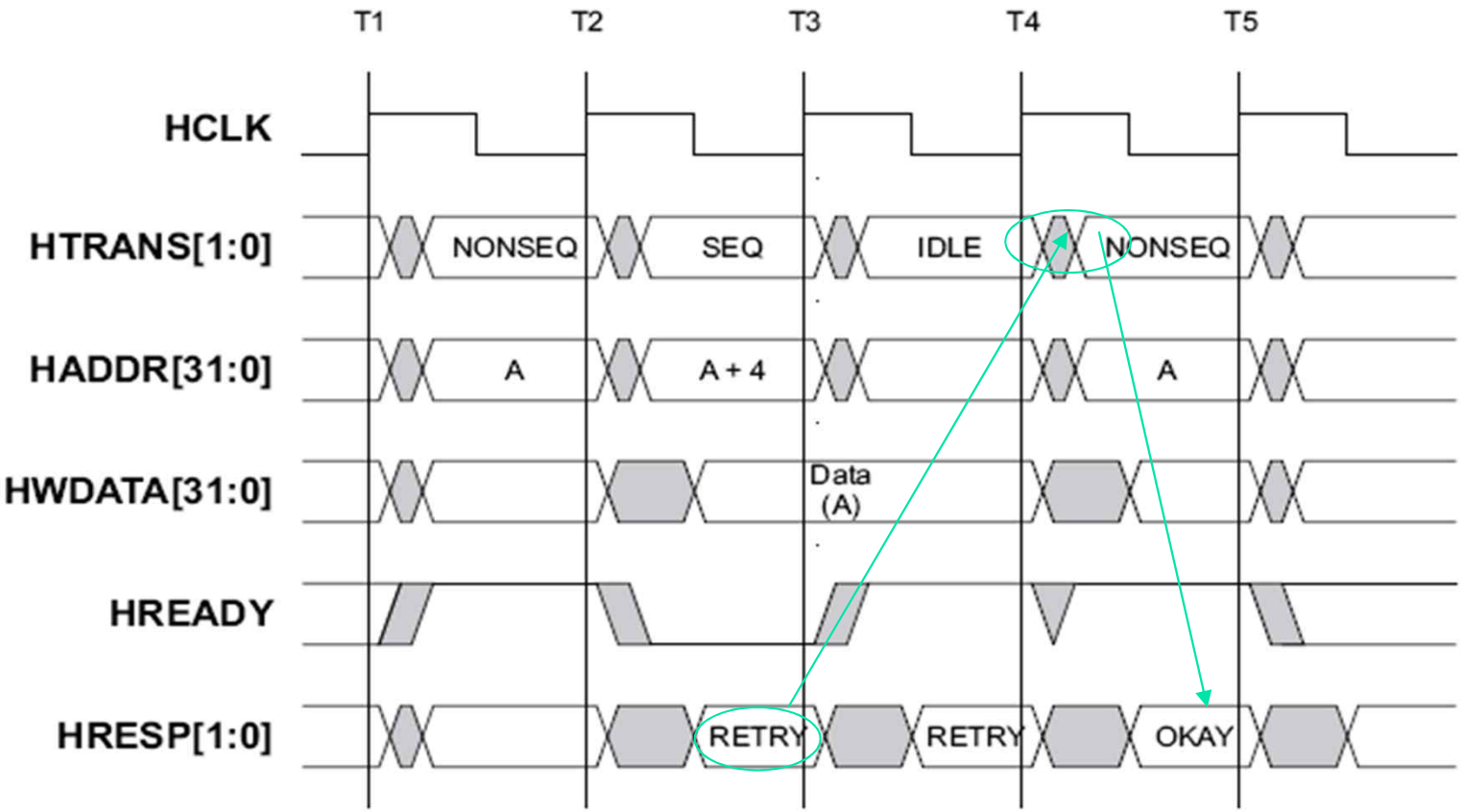
Transfer size	Address offset	DATA [31:24]	DATA [23:16]	DATA [15:8]	DATA [7:0]
Word	0	o	o	o	o
Halfword	0	-	-	o	o
Halfword	2	o	o	-	-
Byte	0	-	-	-	o
Byte	1	-	-	o	-
Byte	2	-	o	-	-
Byte	3	o	-	-	-

Data Buses (2/2)

- Active byte lanes for a 32-bit big-endian data bus -

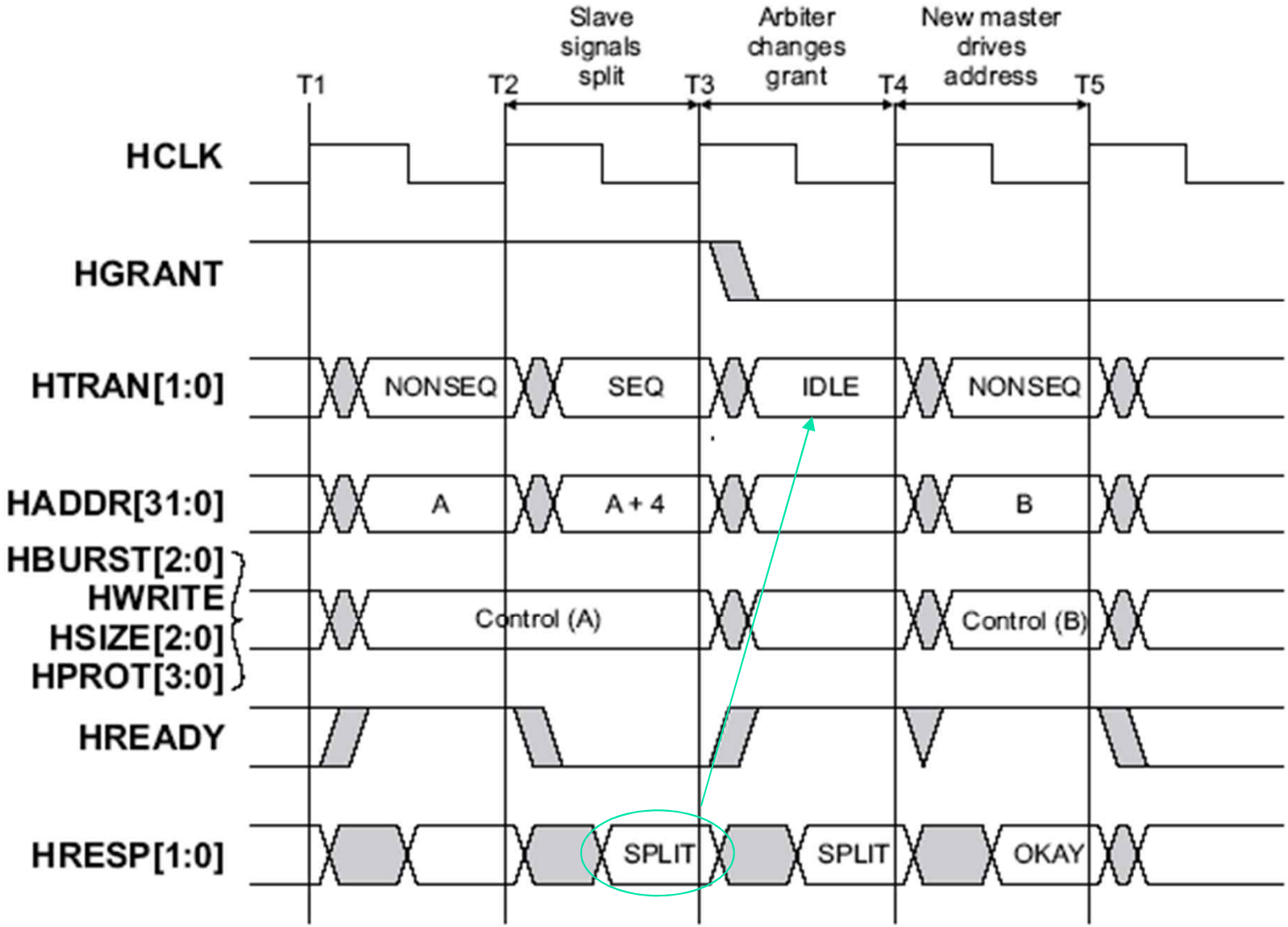
Transfer size	Address offset	DATA [31:24]	DATA [23:16]	DATA [15:8]	DATA [7:0]
Word	0	o	o	o	o
Halfword	0	o	o	-	-
Halfword	2			o	o
Byte	0	o	-	-	-
Byte	1	-	o	-	-
Byte	2	-	-	o	-
Byte	3	-	-	-	o

Slave Transfer Response : Retry



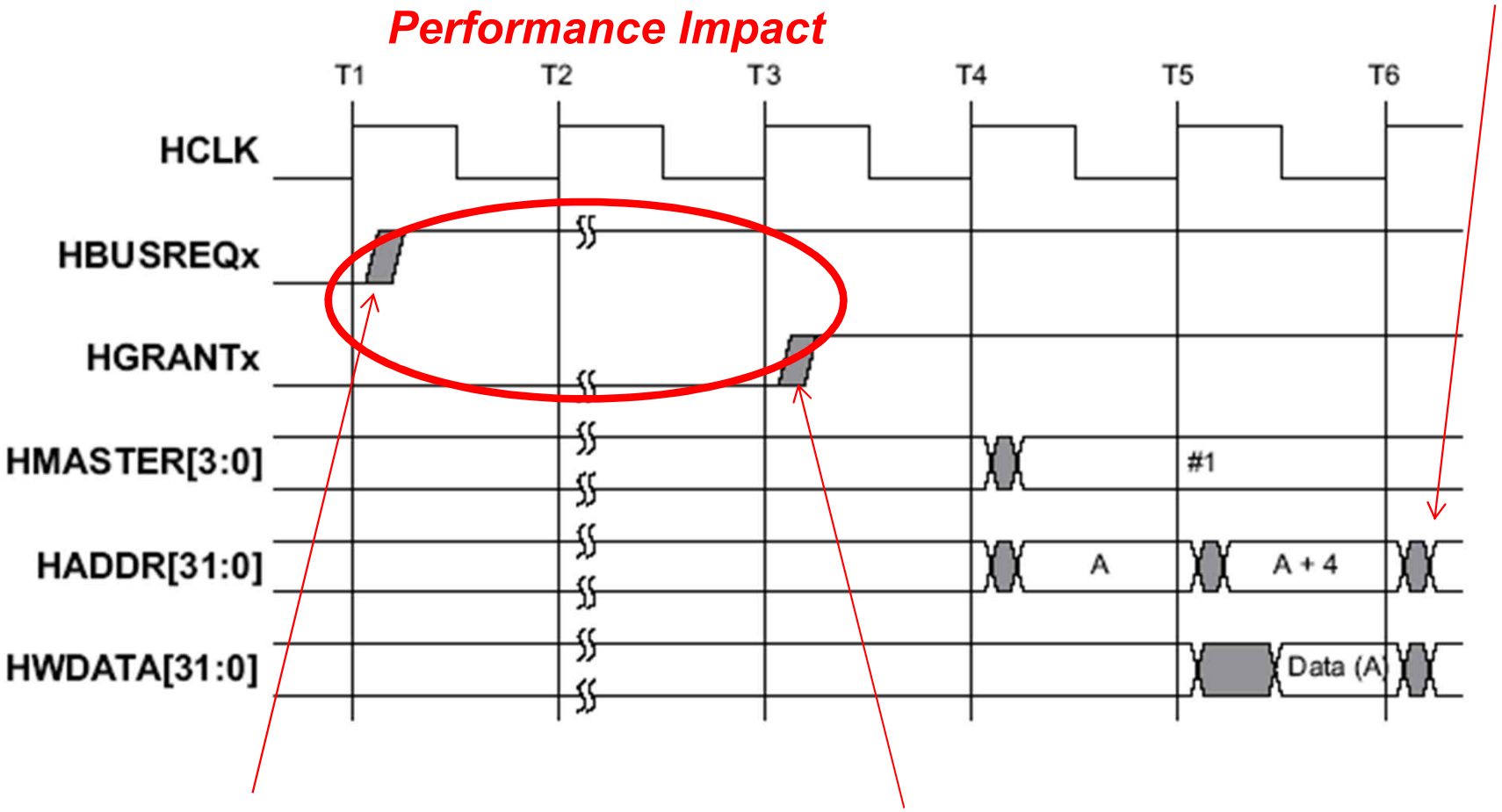
- okay response : 1 cycle
- error, split, retry : more than 2 cycles

Slave Transfer Response : Split



Arbitration : Granting Bus Access

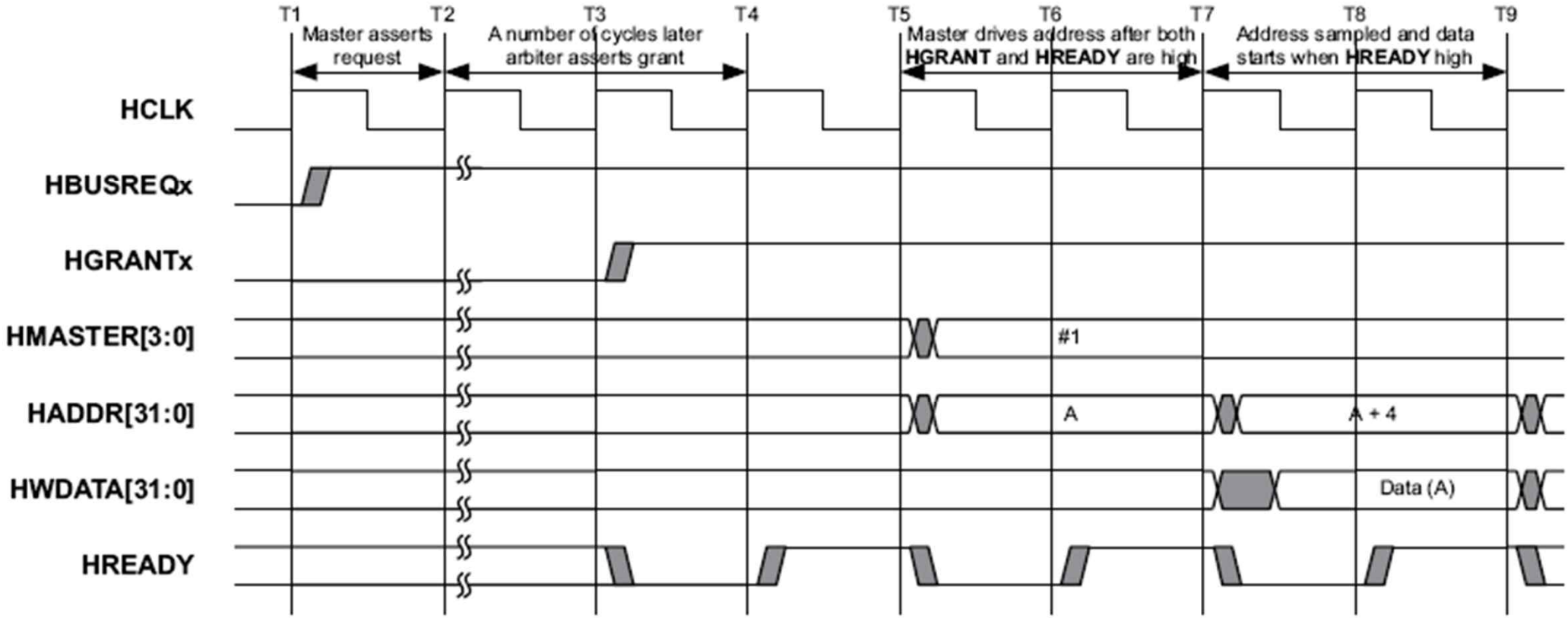
Then the transaction proceeds



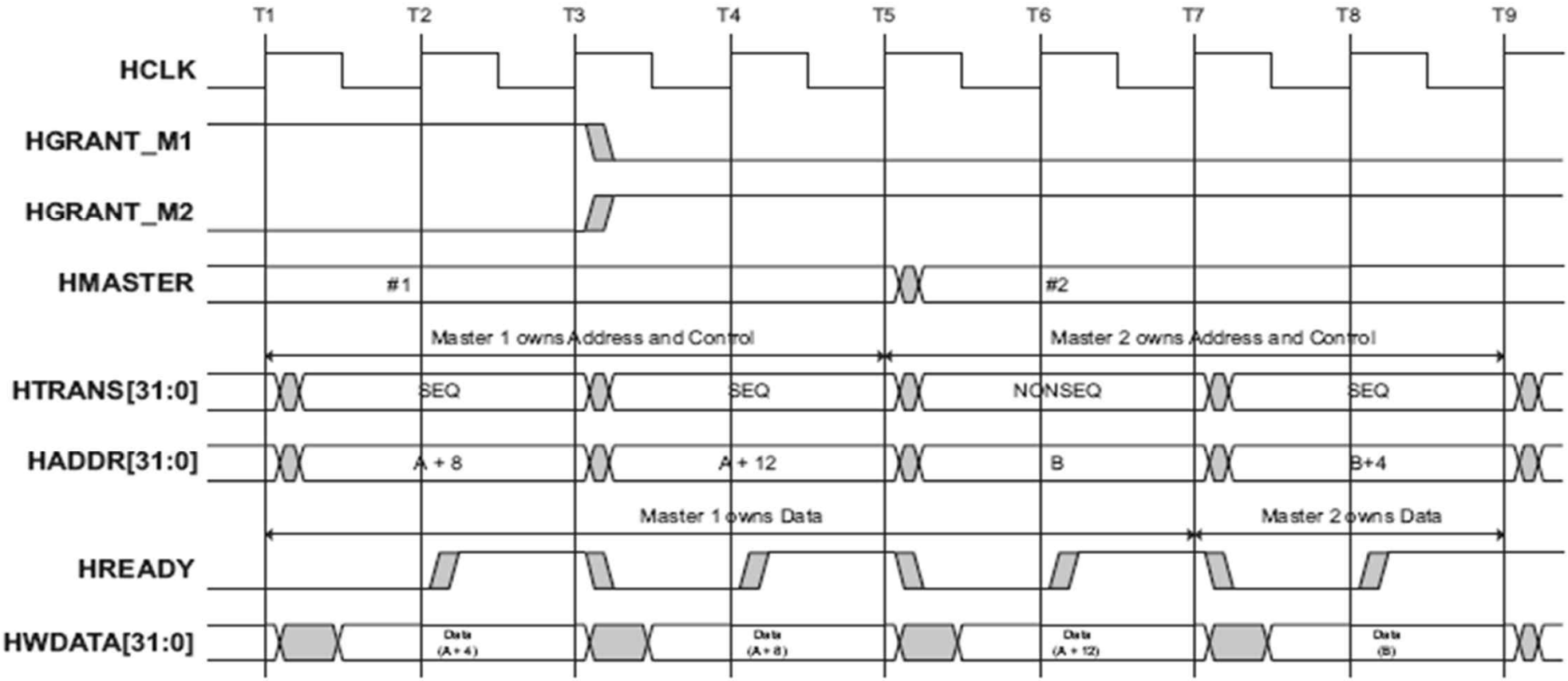
Before a transaction a master makes a request to the central arbiter

Eventually the request is granted

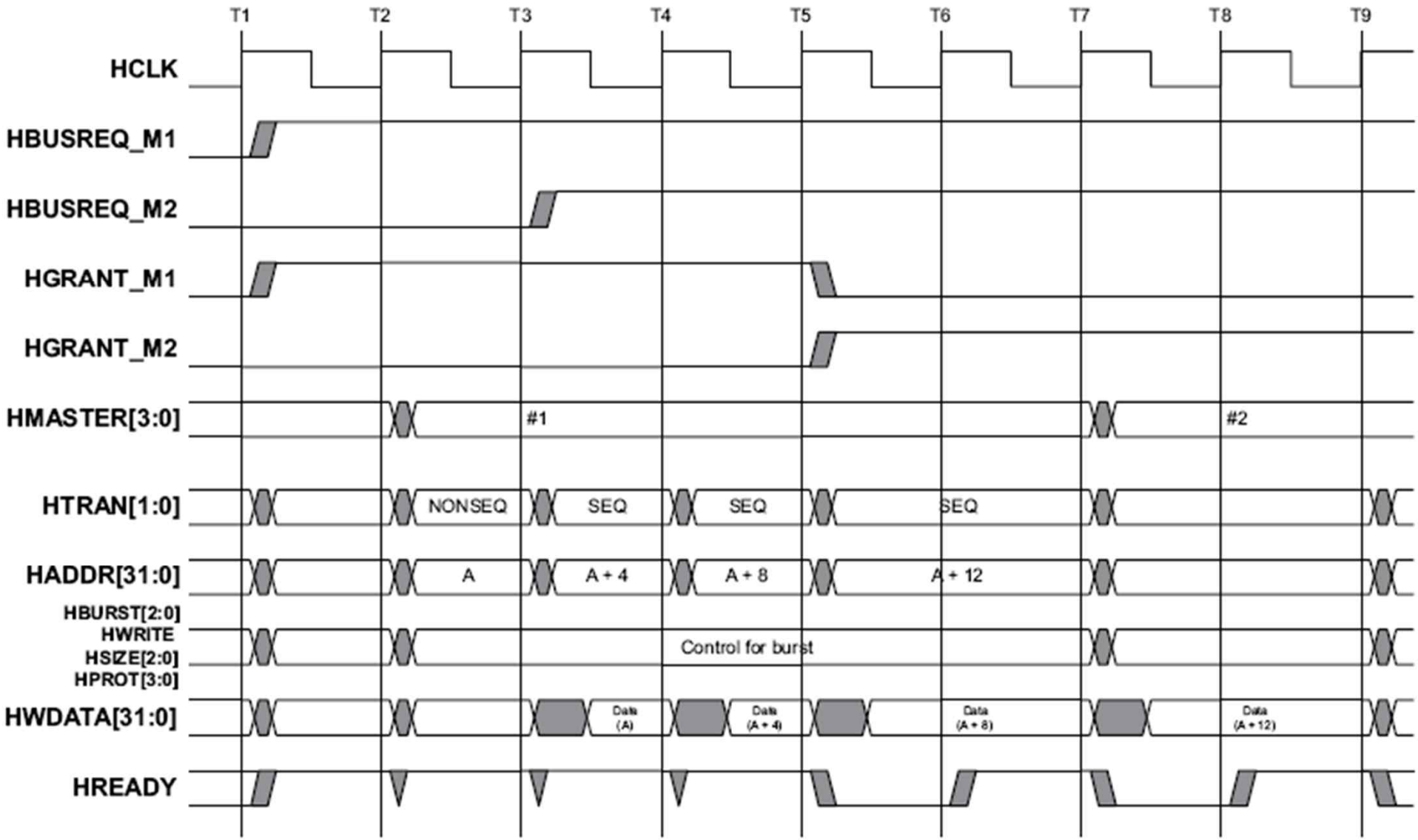
Arbitration : Granting Bus Access with Wait States



Arbitration : Data Bus Ownership

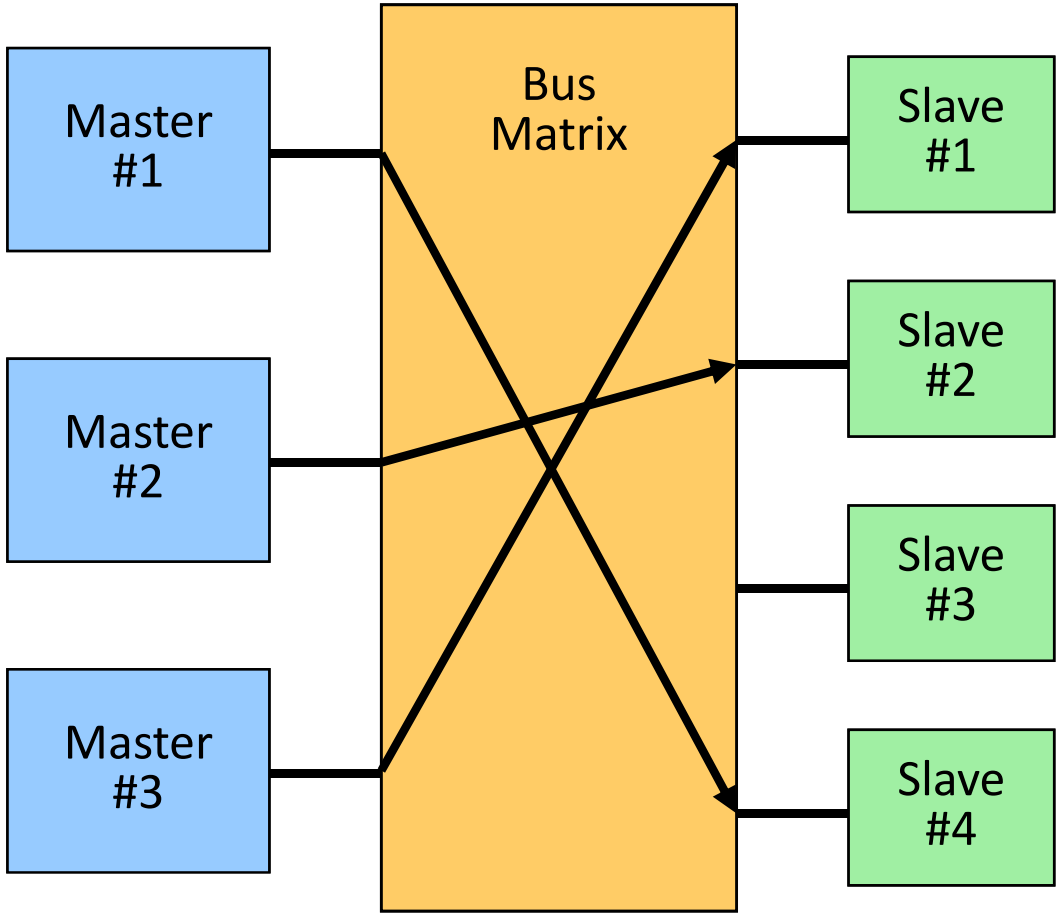


Arbitration : Handover after Burst



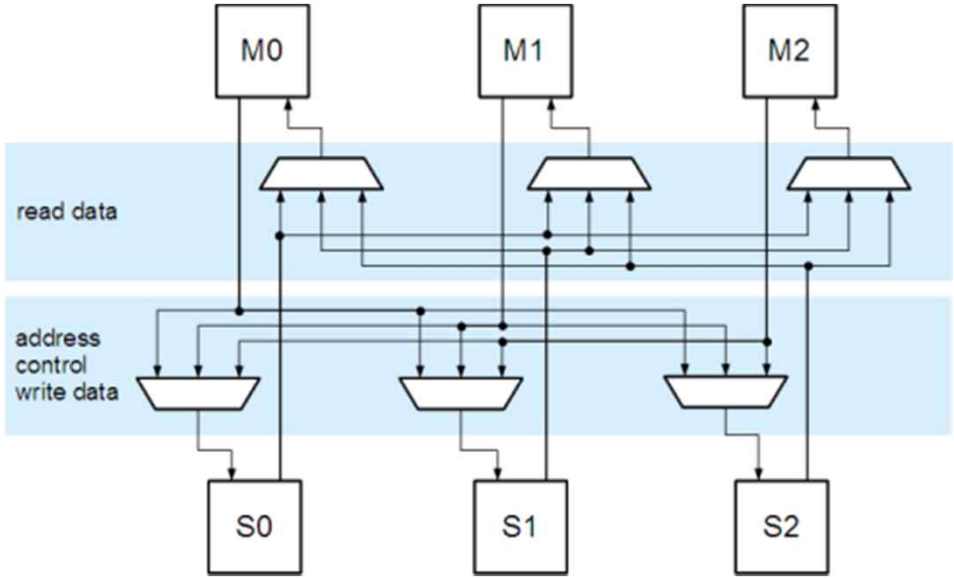
Multi-layer Bus

- Parallel paths between masters and slaves
- Key Advantages
  - Increased bandwidth
  - Design flexibility
- Uses the same interface protocol



In Brief: Multi-layer AHB and AHB-Lite (1/2)

- ARM SoC World shifts towards crossbar switched interconnects, in the form of multi-layer busses
  - Each layer of the bus is an independent single master AHB system
- Instead of a rather complex monolithic multiplexing scheme,
  - A multi-layer AHB bus architecture with M masters and S slaves is structured as M ×1:S muxes plus S × M:1 slave muxes all connected to separate arbitration and decoding logic

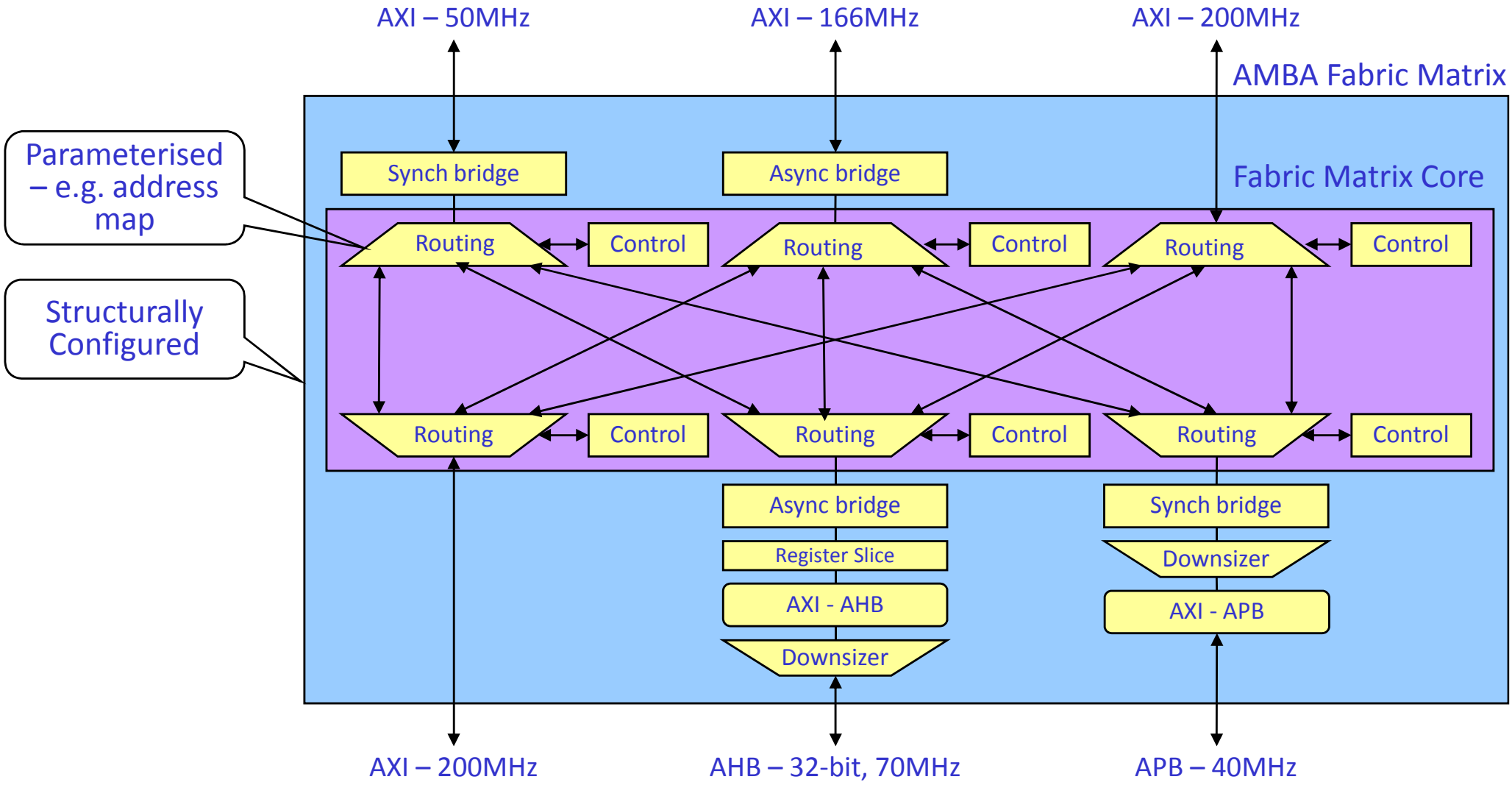


In Brief: Multi-layer AHB and AHB-Lite (2/2)

- Multiple masters can talk to multiple slaves concurrently, as long as no two masters don't try to access the same slave at the same time
- All arbitration and protocol complexity moves into the fabric
  - The interface implementation becomes simpler as a number of unneeded signals, most notably HGRANT and HBUSREQ, can be removed along with their associated protocol
- Although not a necessary consequence of the multi-layer architecture, getting rid of the unpopular SPLIT and RETRY handshaking mechanism was another advantage
  - With the advent of AMBA3 this AHB subset has been standardized upon as AHB-Lite
    - AHB-Lite is a subset of the full AHB specification for designs where only a single bus master is used. This can either be a simple single-master system, or a multi-layer AHB-Lite system where there is only one AHB master per layer

AMBA 3.0 AXI Bus

AMBA3.0: Typical AXI Interconnect



Why update the AMBA Protocol?

- Future SoC designs demand:
  - Increasing levels of system complexity
  - Increasing performance demands
  - Increasing clock speed requirements
  - Reduction of system power consumption
- Whilst retaining existing AMBA strengths:
  - Modular design
  - Component re-use
  - Simple to understand interface

Why AXI instead of Multi-layer AHB?

- AHB is transfer-oriented
  - With each transfer, an address will be submitted and a single data item will be written to or read from the selected slave
  - All transfers will be initiated by the master, thus the master will be stalled if the slave cannot respond immediately to a transfer request
  - Each master can have only one outstanding transaction
- Sequential accesses (bursts) consist of consecutive transfers which indicate their relationship by asserting HTRANS/HBURST accordingly
- Although AHB systems are multiplexed and thus have independent read and write data busses, they cannot operate in full-duplex mode

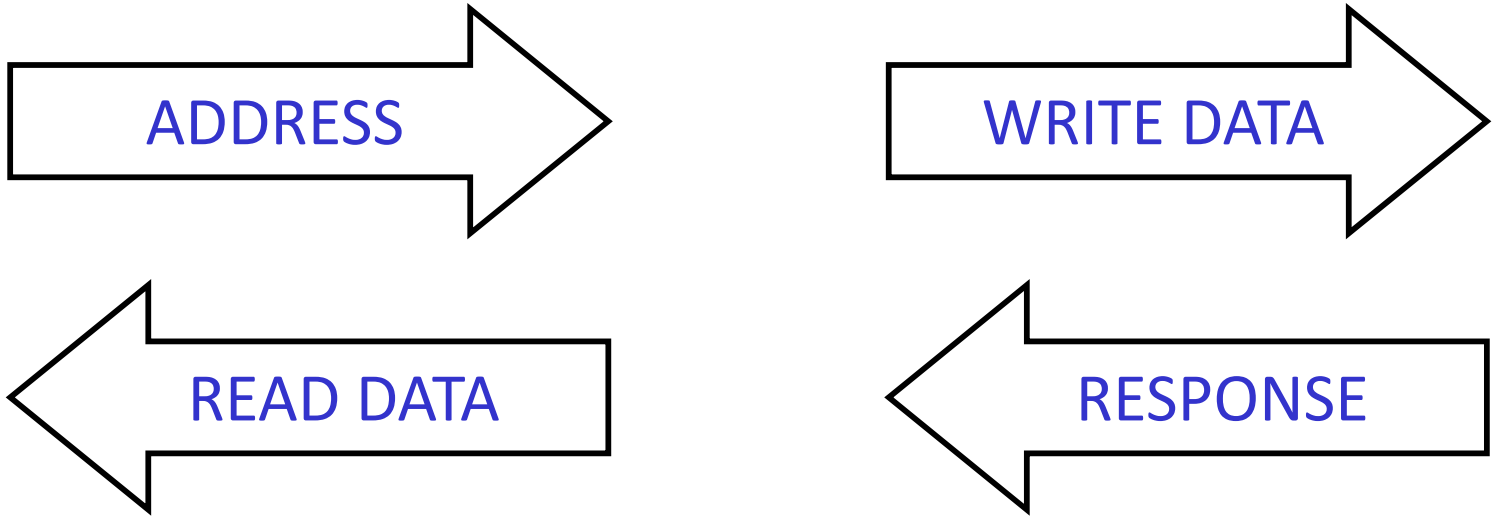
AMBA3 AXI Protocol

[Source: AXI Spec]

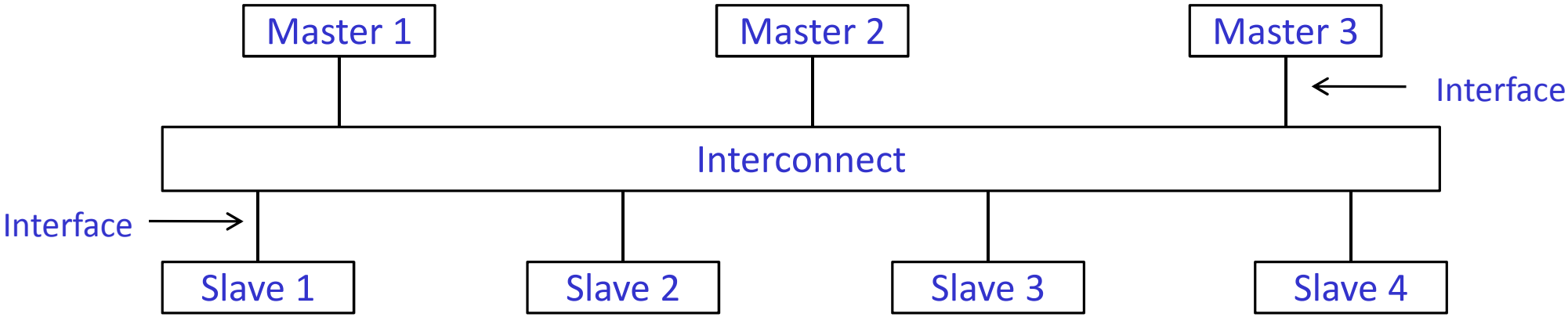
- Advanced eXtensible Interface (AXI)
  - separate address/control and data phases
  - support for unaligned data transfers, using byte strobes
  - uses burst-based transactions with only the start address issued
  - separate read and write data channels, that can provide low-cost Direct Memory Access (DMA)
  - support for issuing multiple outstanding addresses
  - support for out-of-order transaction completion
  - permits easy addition of register stages to provide timing closure
- Five channels
  - Read: Address read (AR) and read (R) data channels
  - Write: Address write (AW), write (W) data, and response (R) channels

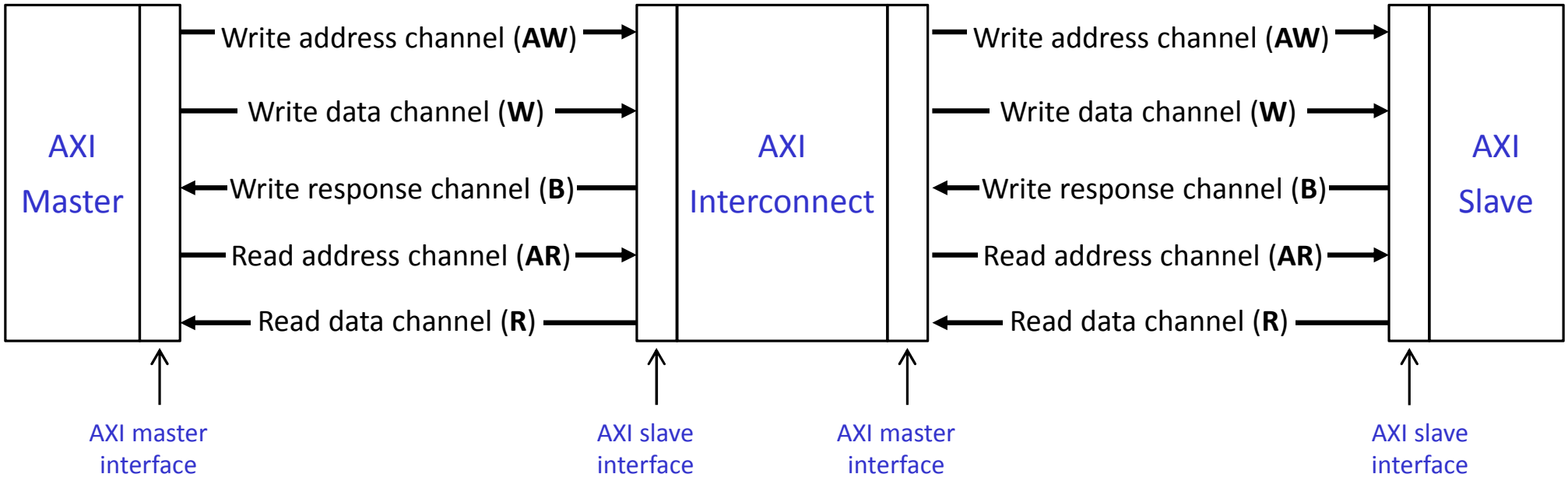
Channel Architecture

- Four groups of signals
  - Address “A” signal name prefix
  - Read “R” signal name prefix
  - Write “W” signal name prefix
  - Write Response“B” signal name prefix

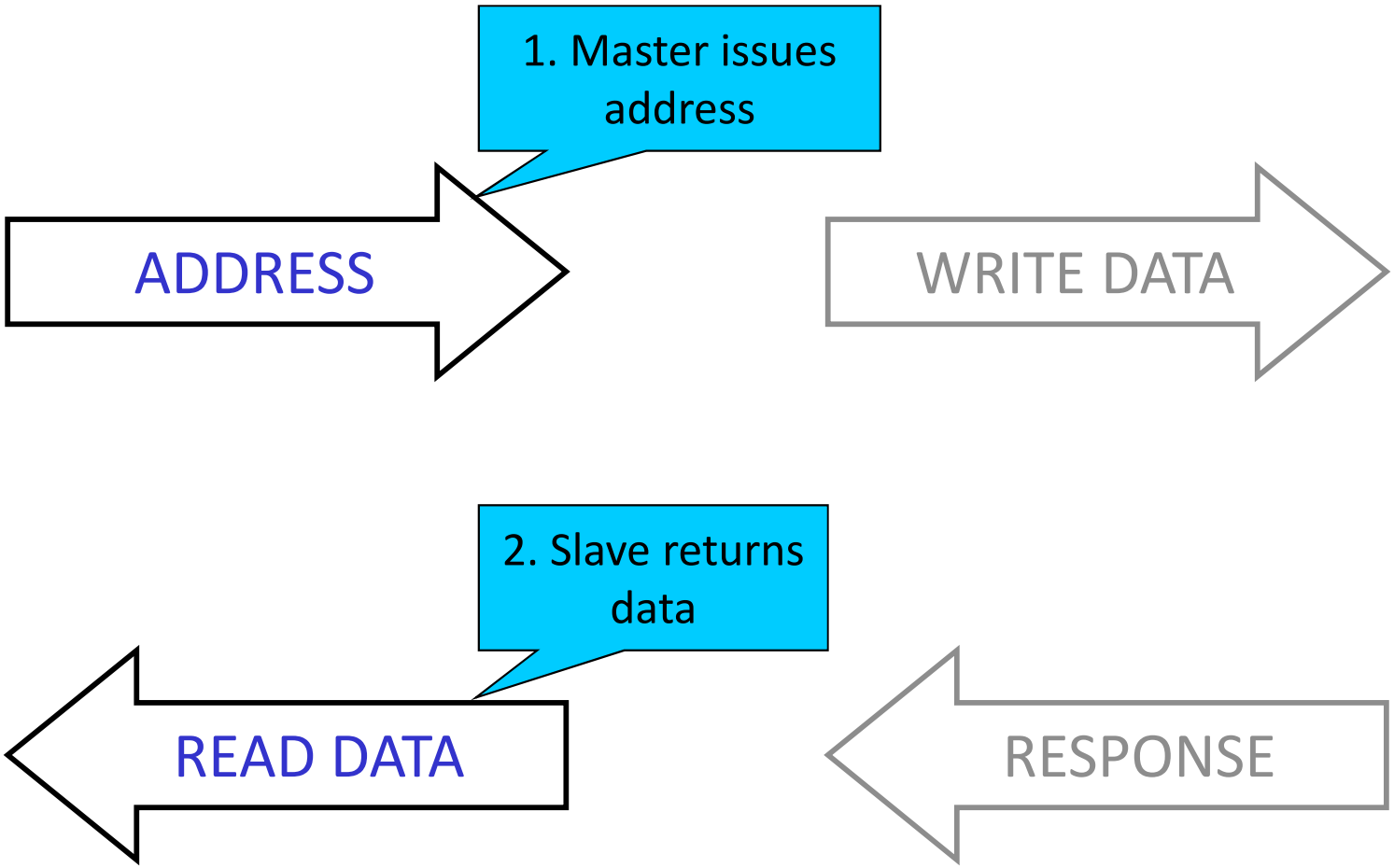


Interconnect, Interface & Channel

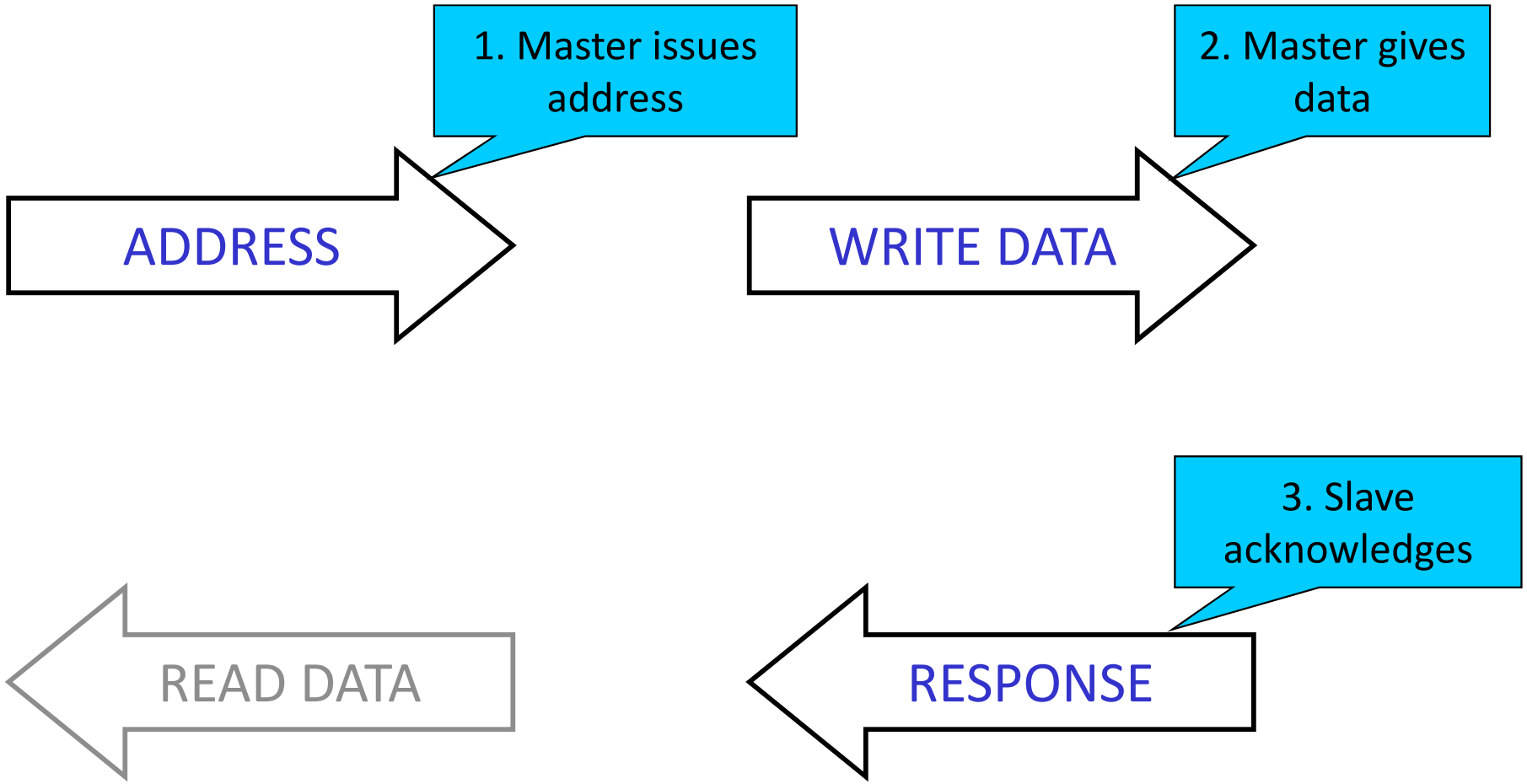




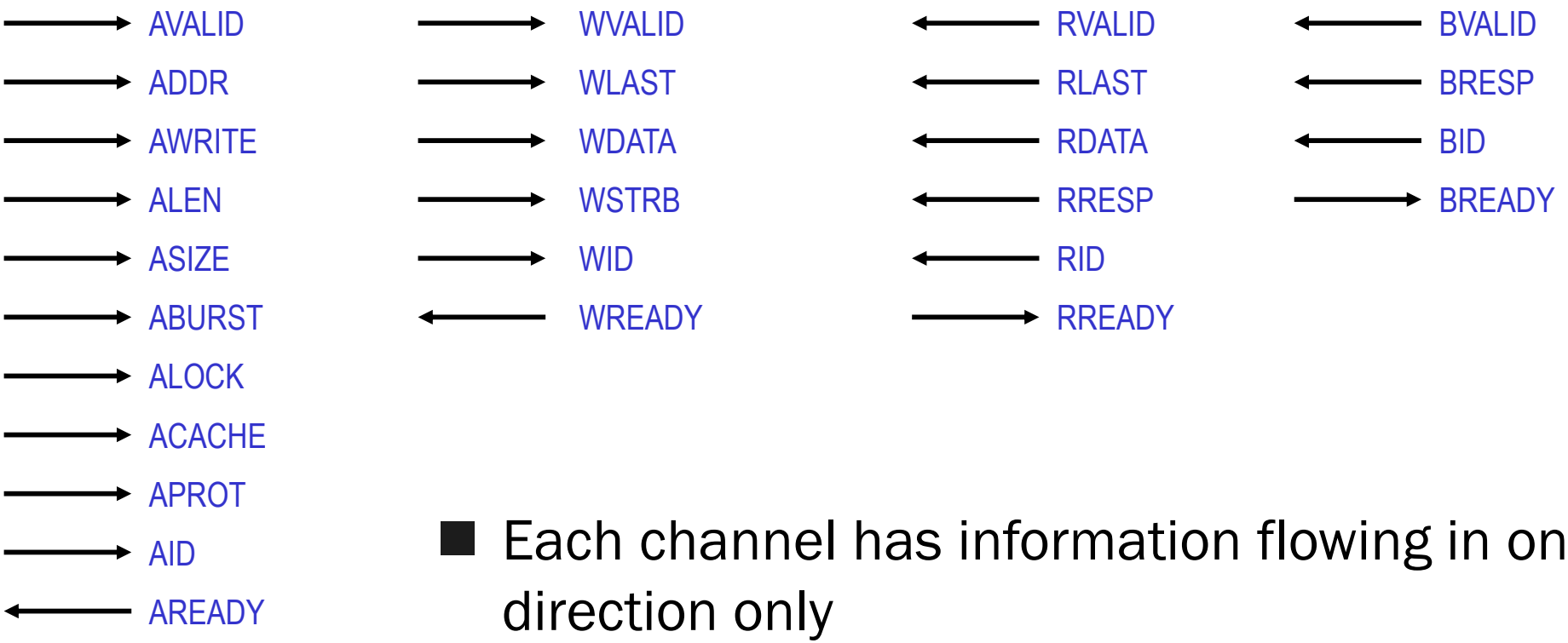
To read ...


1. Master issues address
ADDRESS
WRITE DATA
2. Slave returns data
READ DATA
RESPONSE

To write ...


1. Master issues address
2. Master gives data
ADDRESS
WRITE DATA
3. Slave acknowledges
READ DATA
RESPONSE

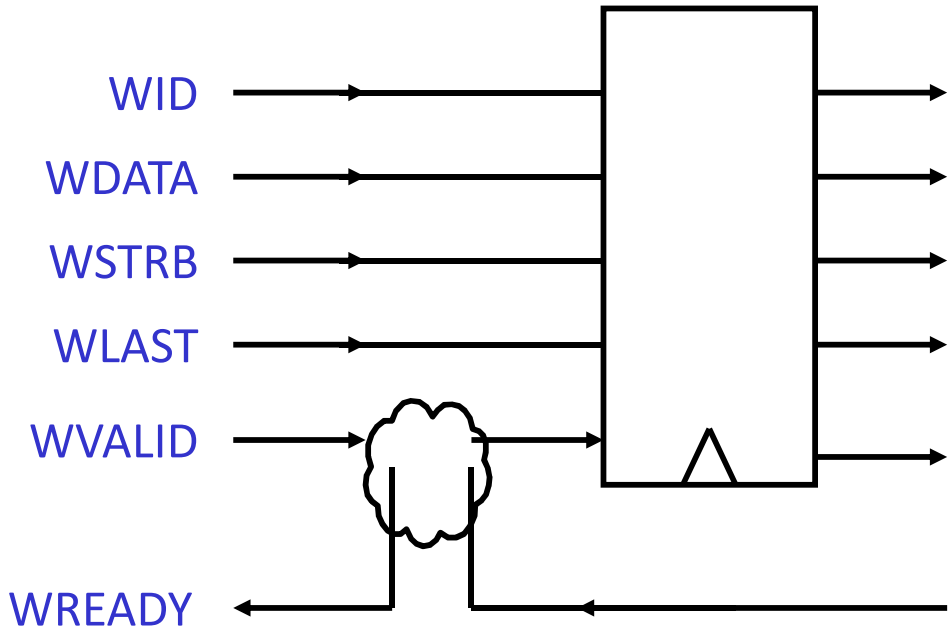
Channels - One way flow



- Each channel has information flowing in one direction only
- READY is the only return signal

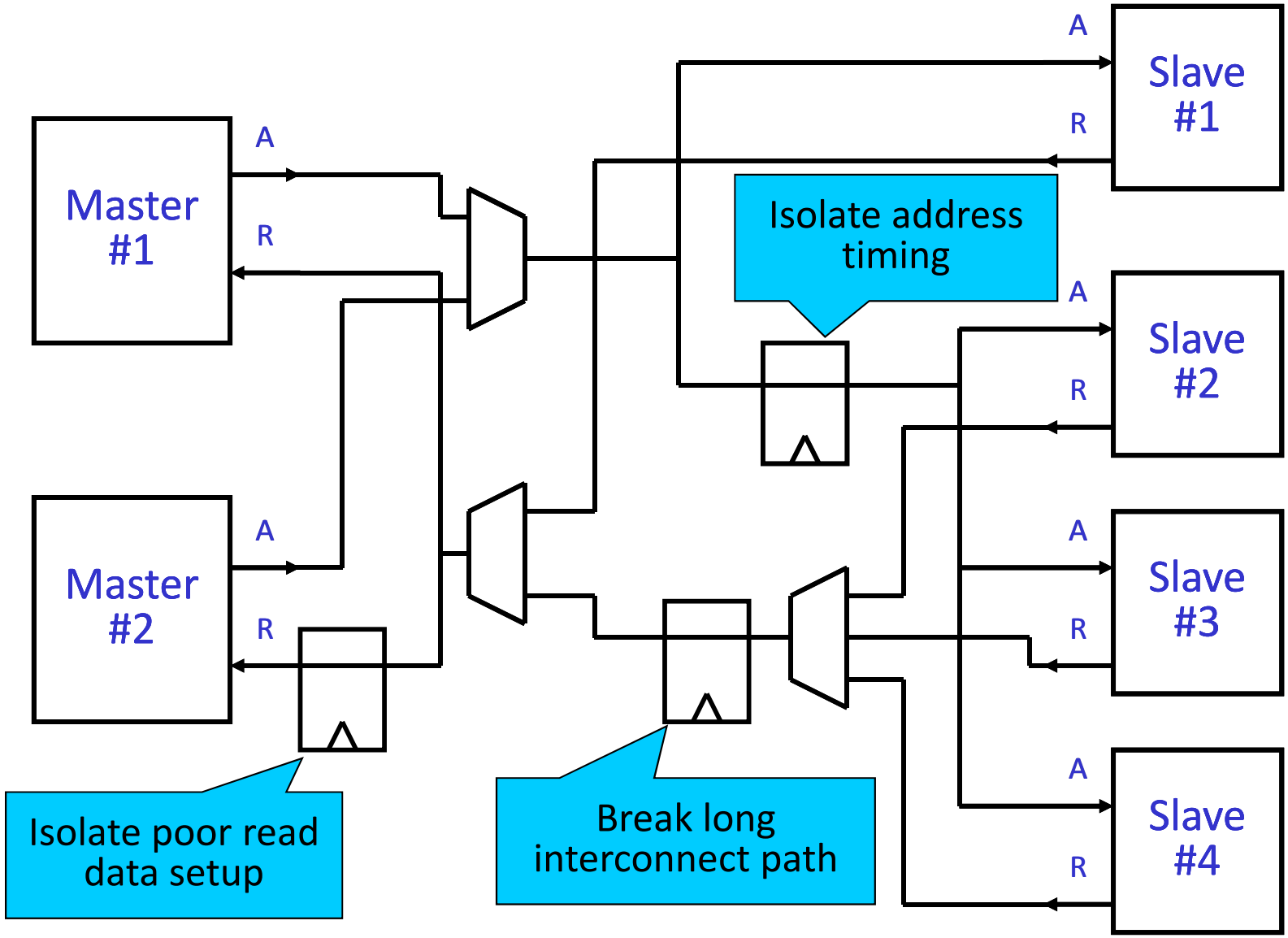
Register Slices for Max Frequency

- Register slices can be applied across any channel
- Allows maximum frequency of operation by matching channel latency to channel delay



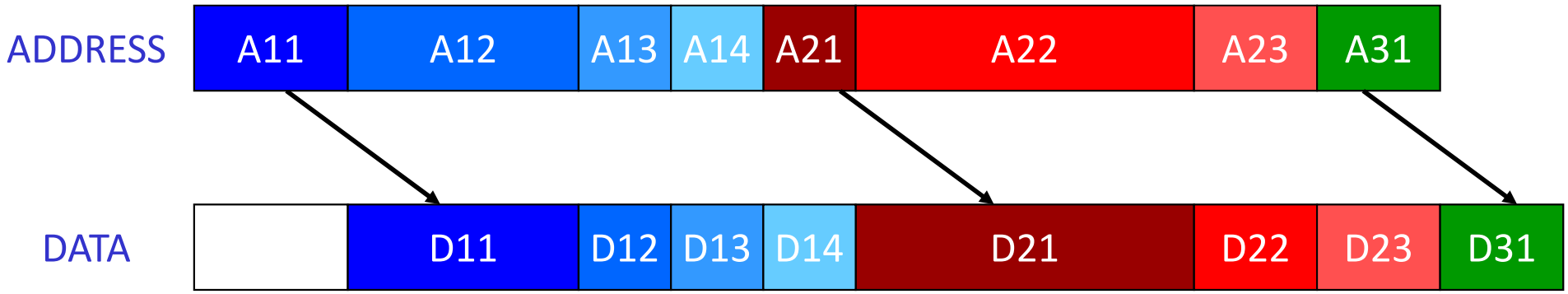
- Allows system topology to be matched to performance requirements

Example Register Slices

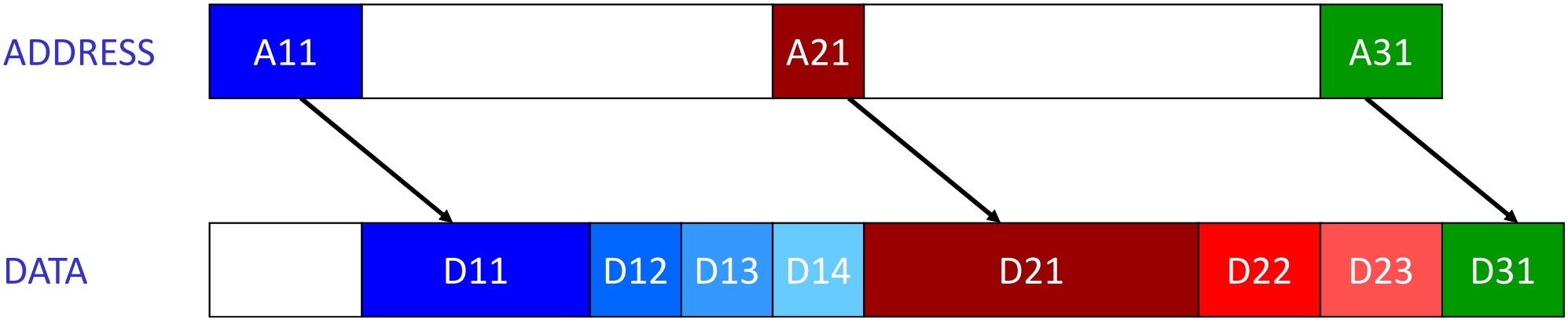


AMBA 2.0 AHB Burst

- AHB Burst
  - Address and Data are locked together
  - Single pipeline stage
  - HREADY controls intervals of address and data

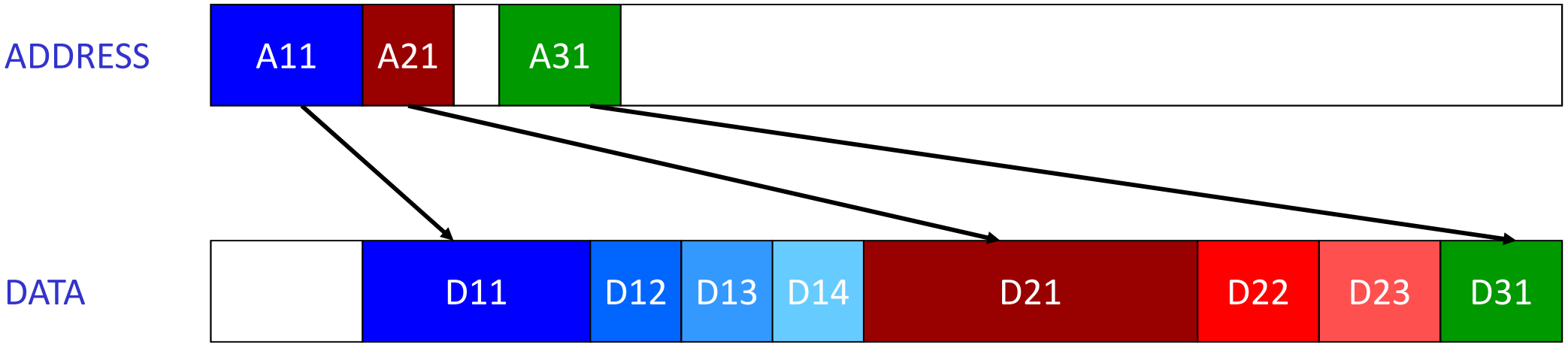


AXI - One Address for Burst



- One Address for entire burst
- Separation of address and data channel
  - Master provides the start address of burst
  - Slave needs to generate the remaining addresses based on burst type (FIXED, INCR, WRAP)

AXI - Outstanding Transactions

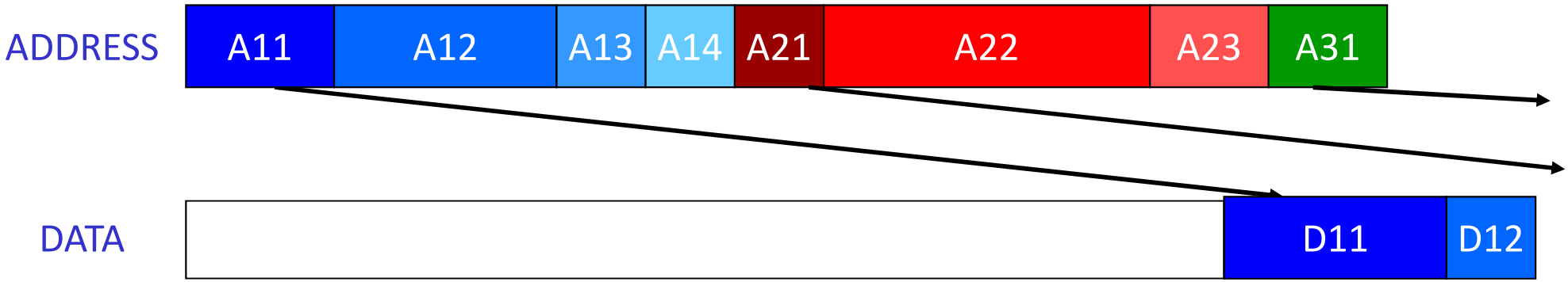


- Benefit of split transaction: multiple outstanding requests
- Parameters for multiple outstanding requests
  - Master I/F: Issuing capability → # of outstanding request that the master can generate
  - Slave I/F: Acceptance capability → # of outstanding request that the slave can accommodate

Out of Order Interface

- Each transaction has an ID attached
  - Channels have ID signals - AID, RID, etc.
- Transactions with the same ID must be ordered
- Requires bus-level monitoring to ensure correct ordering on each ID
  - Masters can issue multiple ordered addresses

AMBA 2.0 AHB Burst - Slow slave



- With AHB
  - If one slave is very slow, all data is held up.

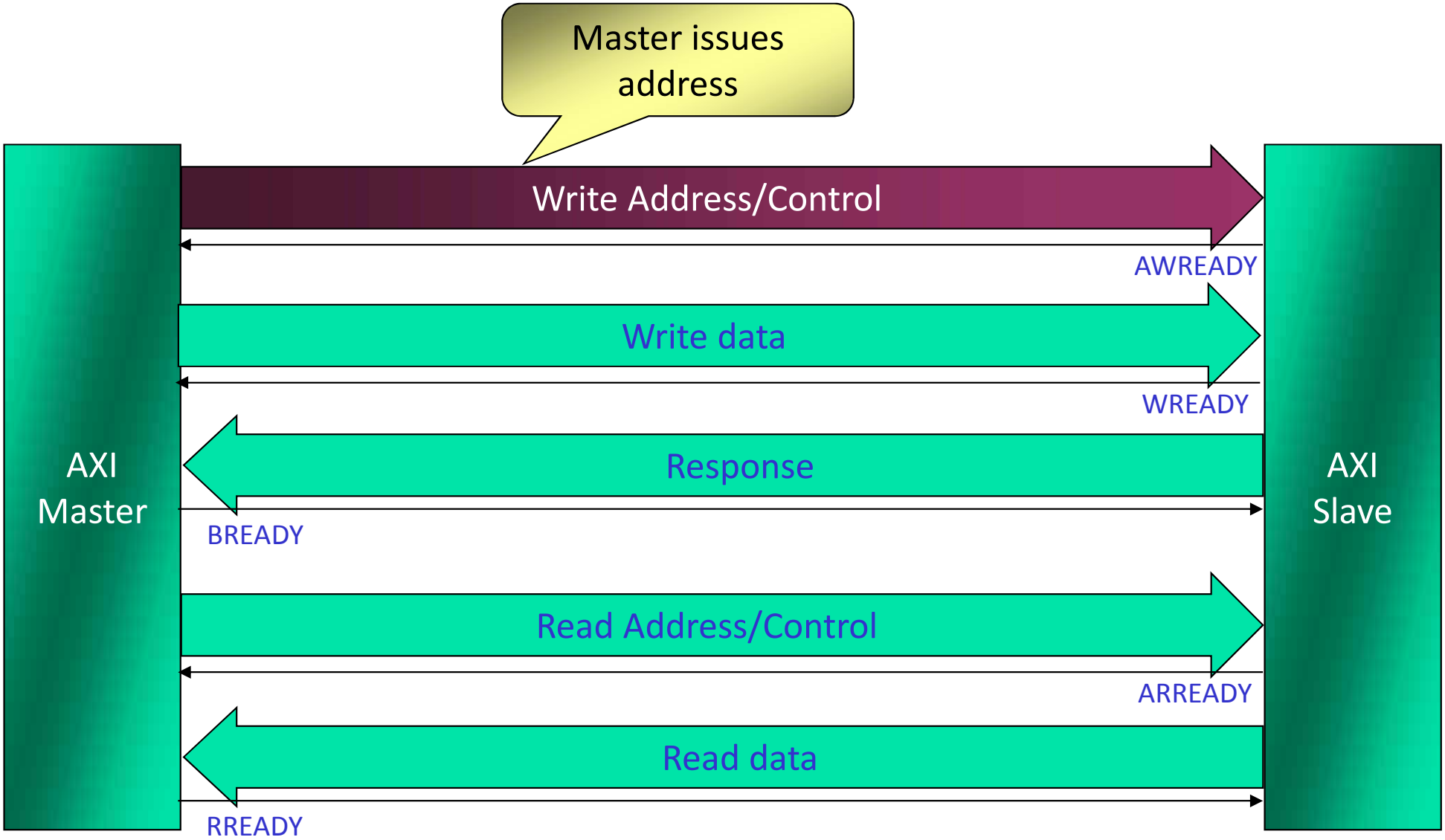
Separate Read / Write Channels

- AMBA AXI allows for independent read and write transactions.

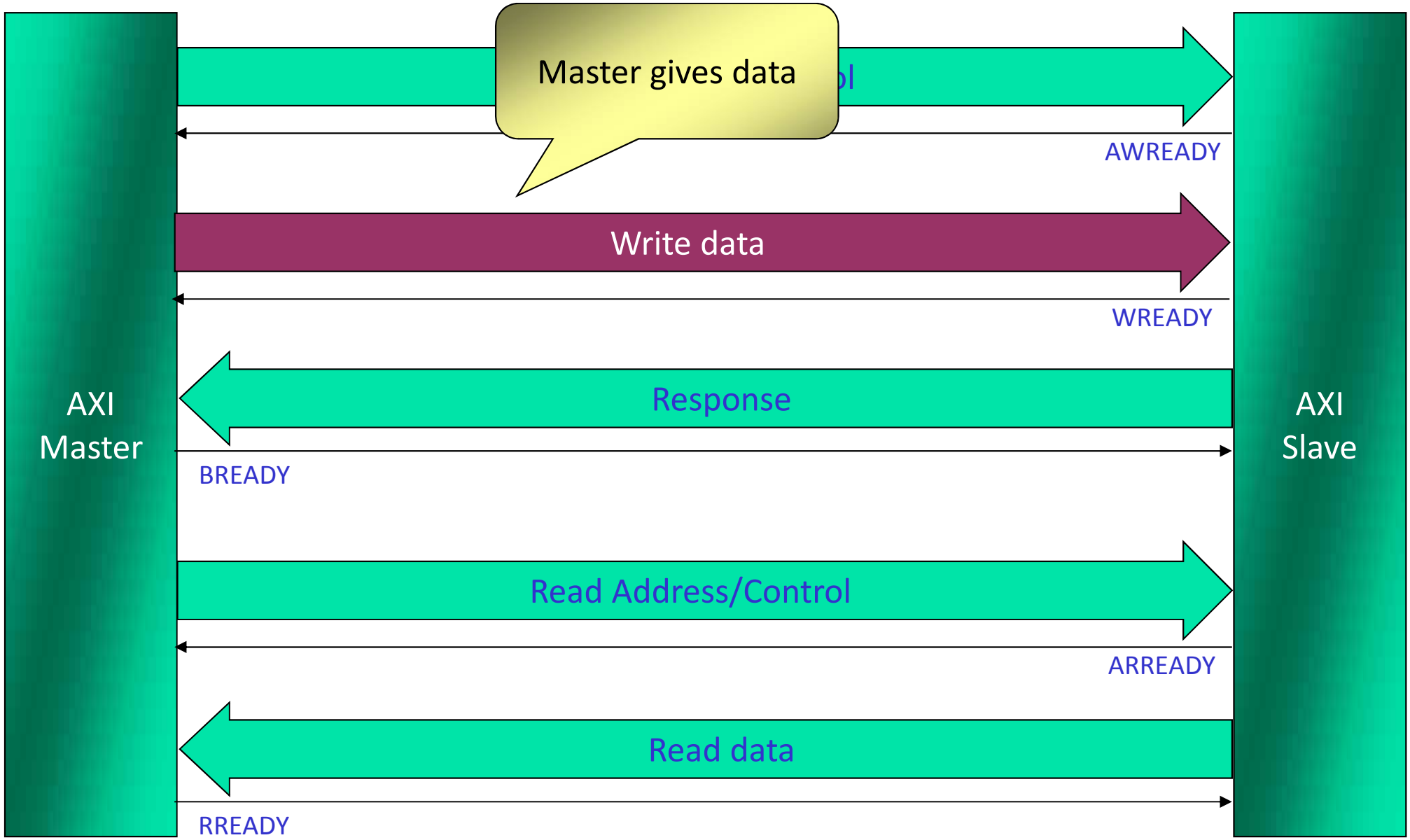
Split Transaction

- Address, data, and response are handled separately.

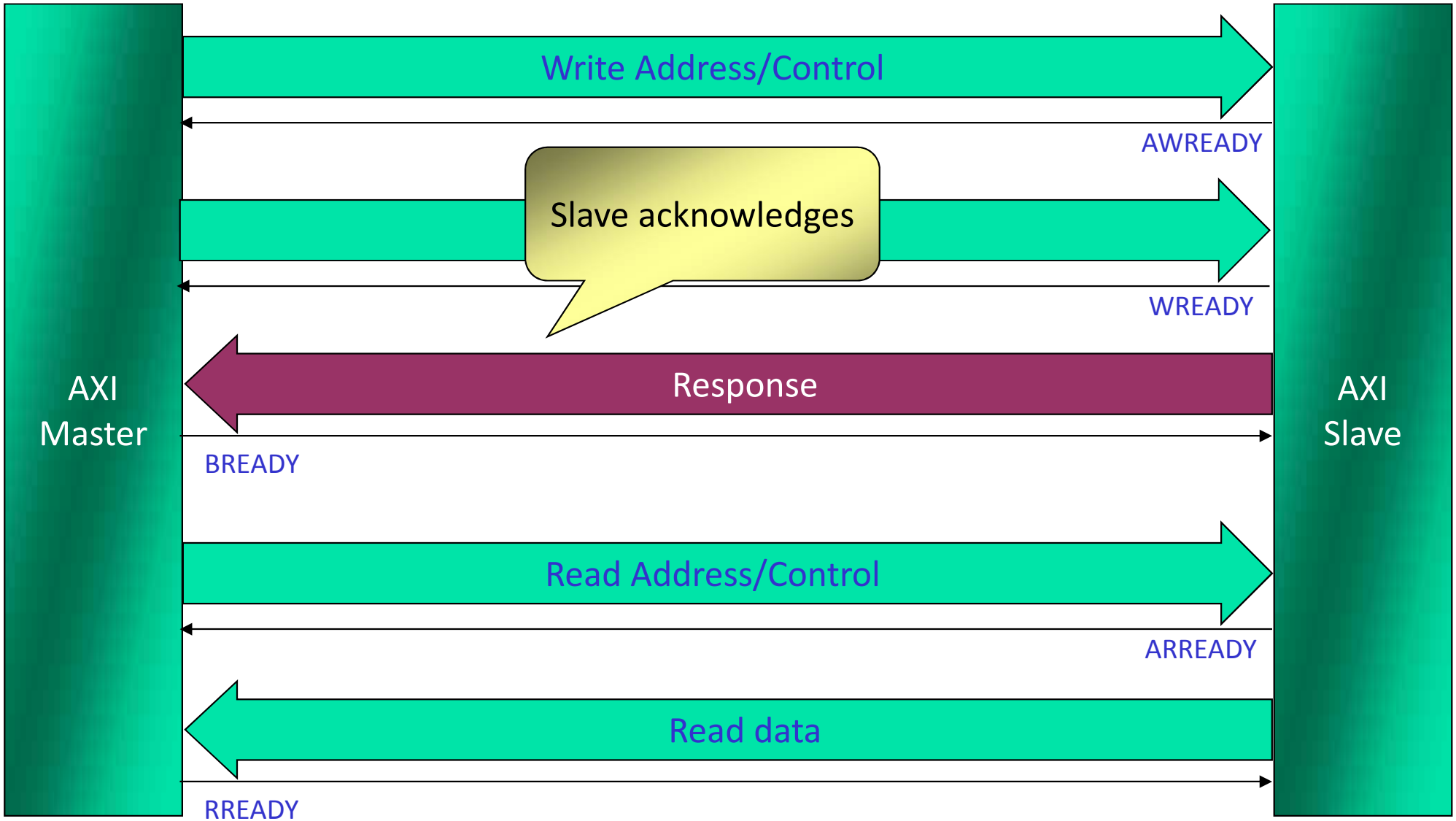
Split Transaction: Write (1/3)



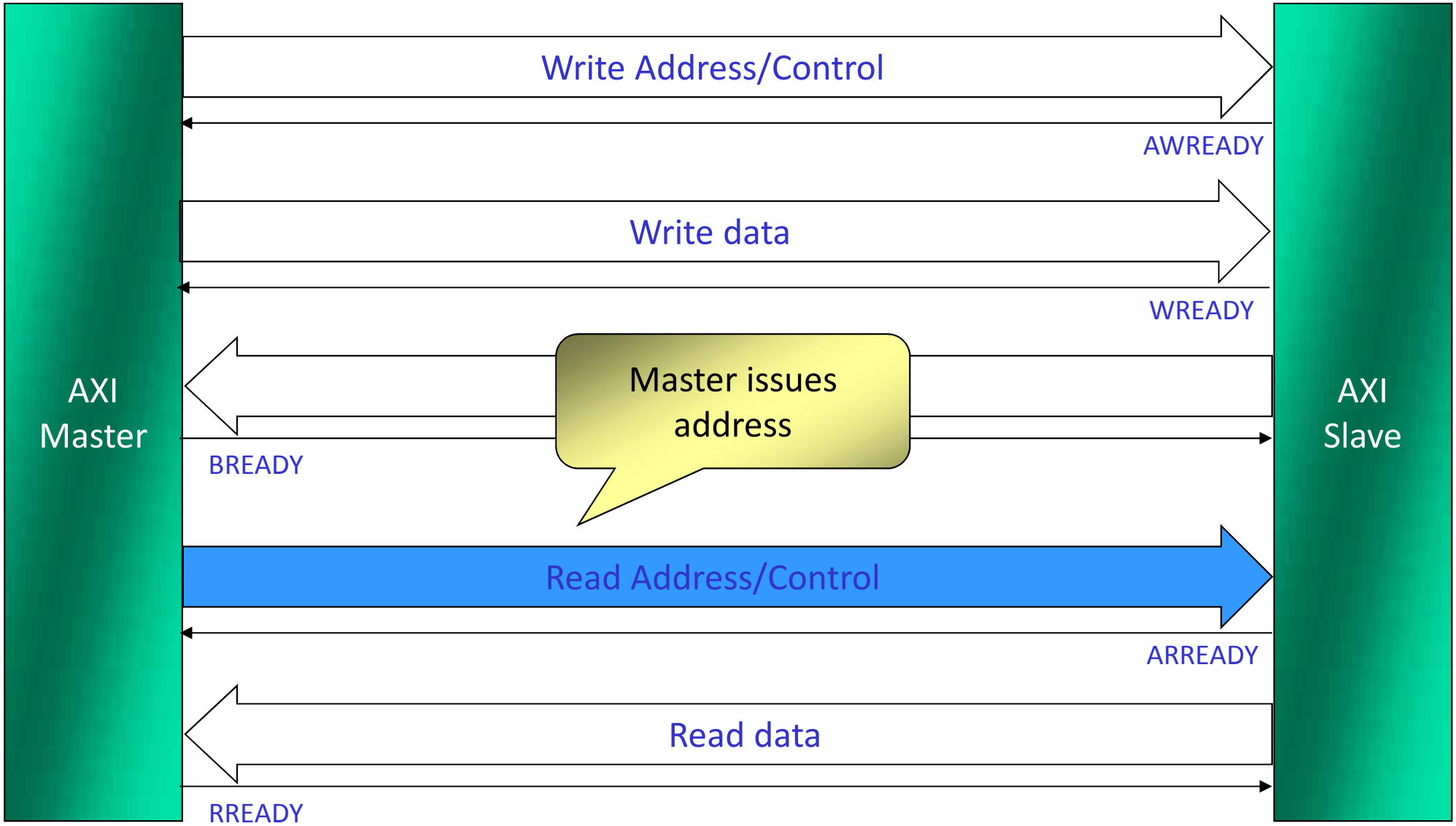
Split Transaction: Write (2/3)



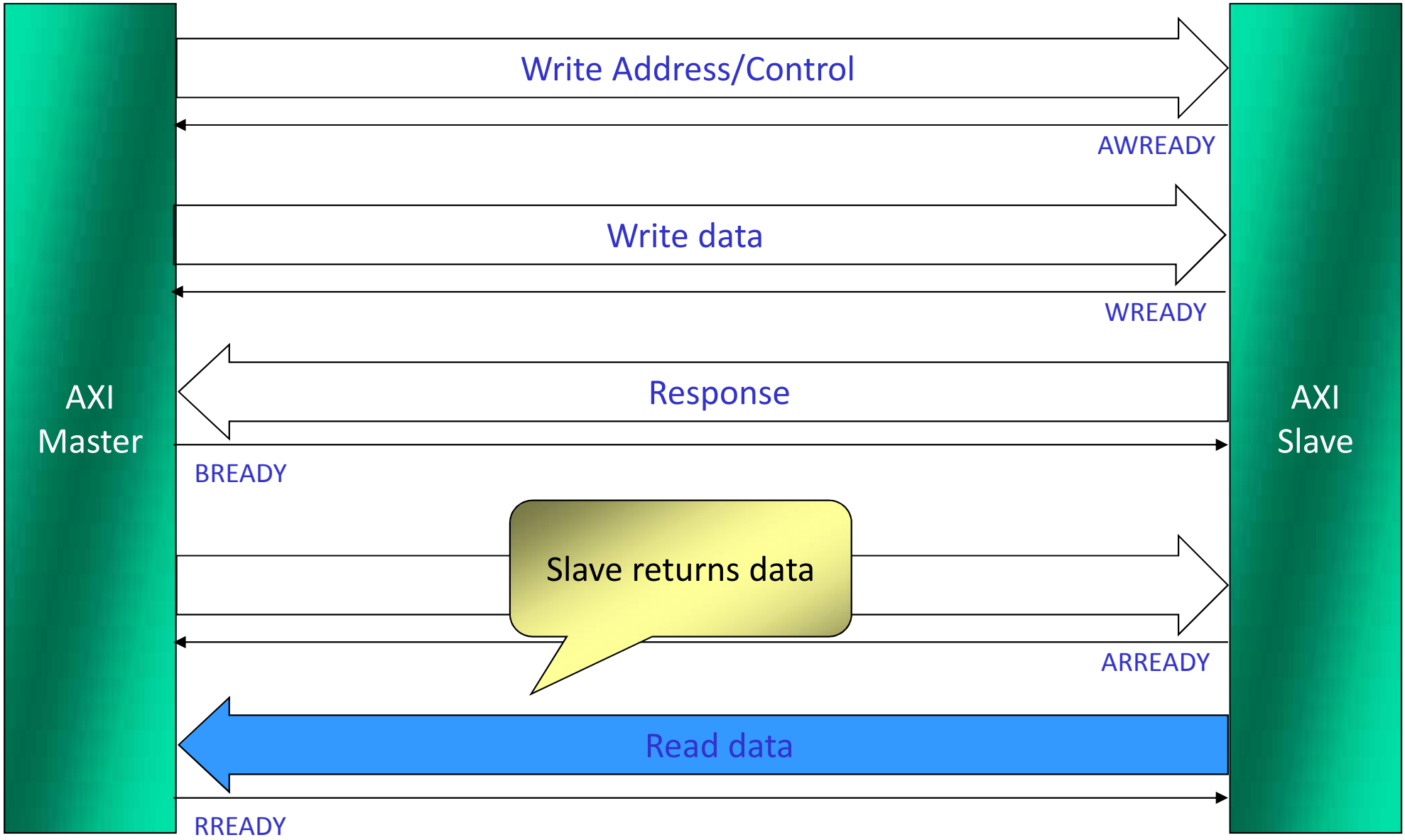
Split Transaction: Write (3/3)



Split Transaction: Read (1/2)

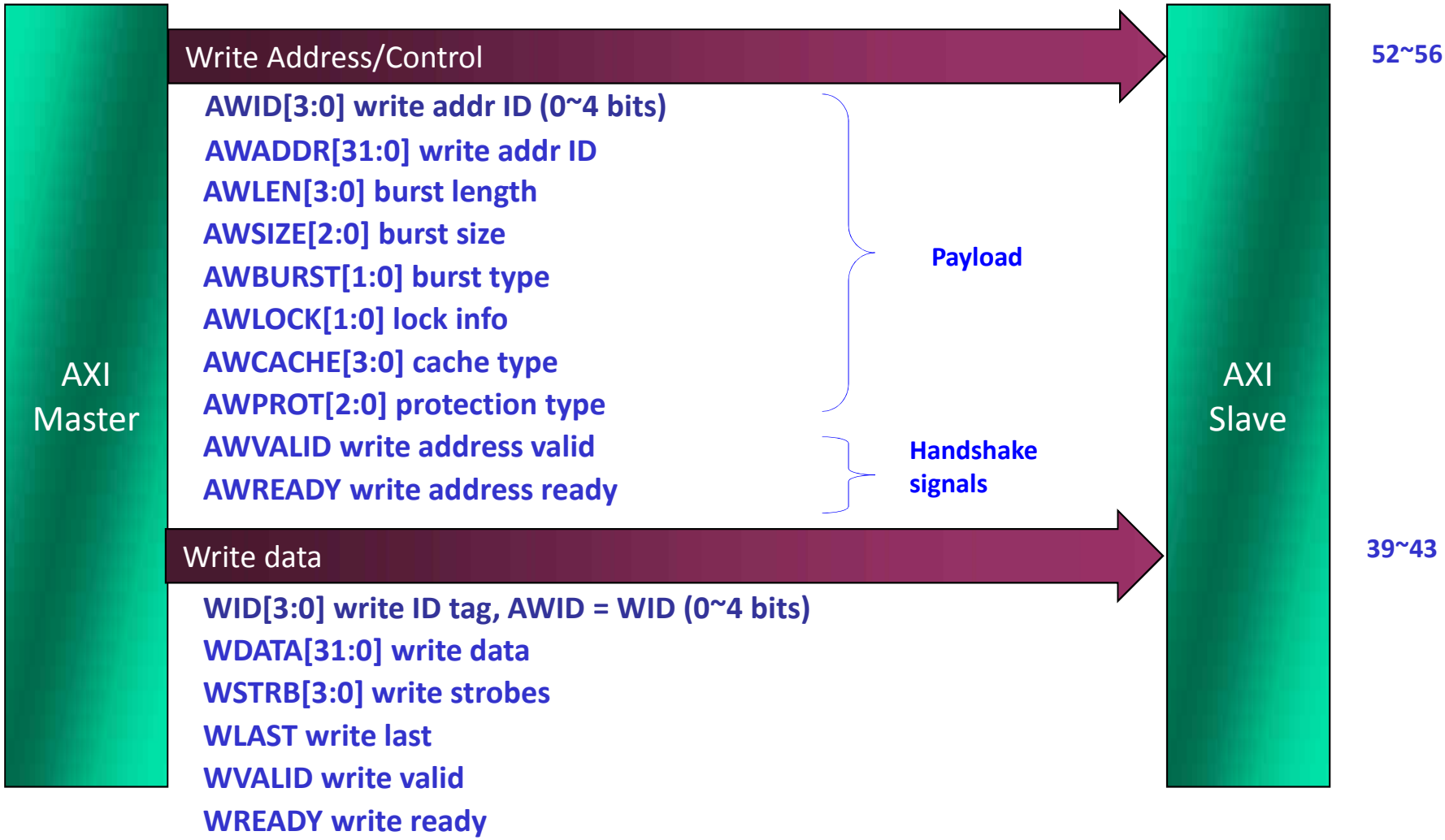


Split Transaction: Read (2/2)

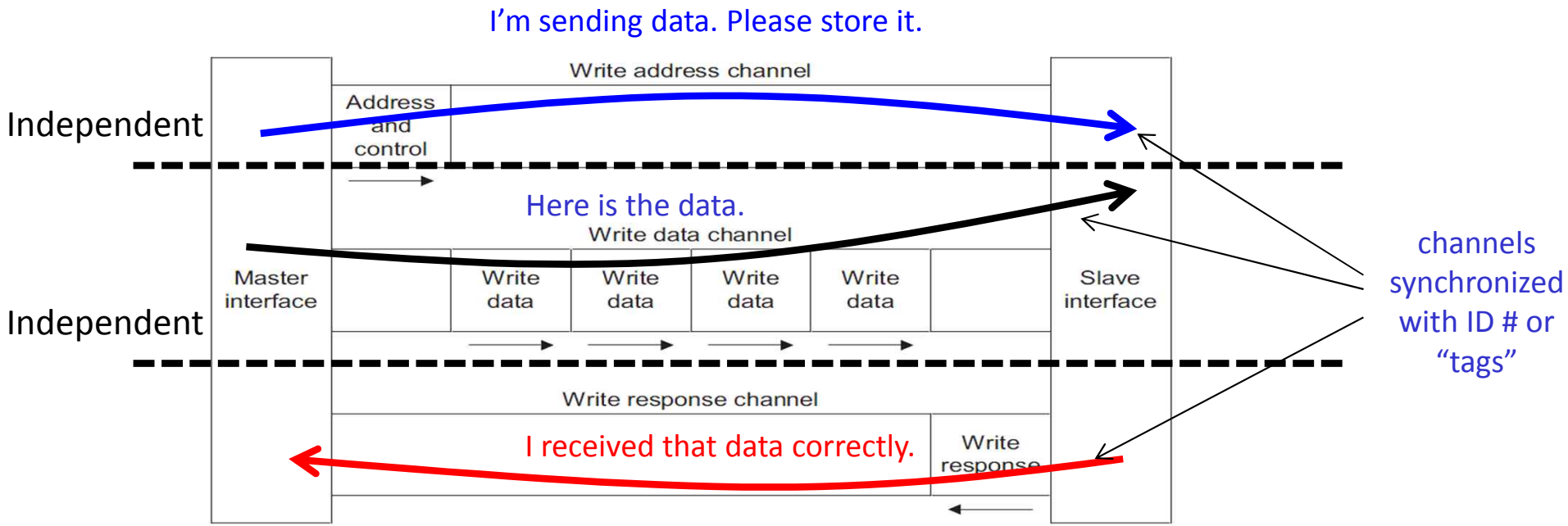


Wire Counts

- Address 32b, data 32b bus case: 184~204
  - AW: 52~56, W: 39~43, B: 4~8, AR: 52~56, R: 37~41

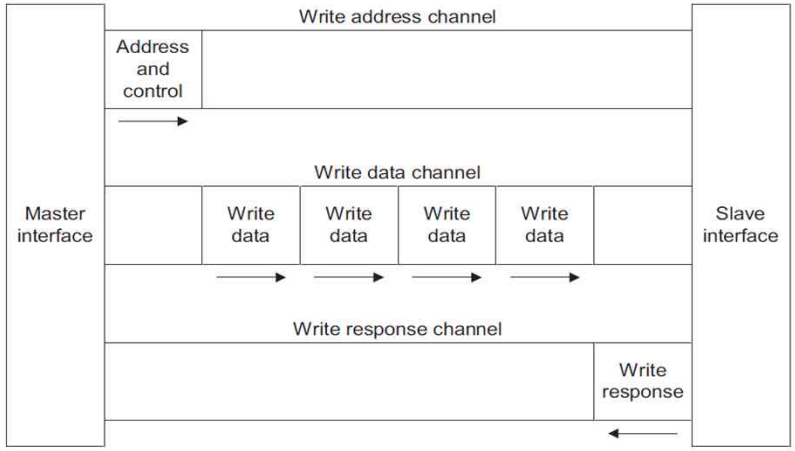


Write Address Channel & Signals



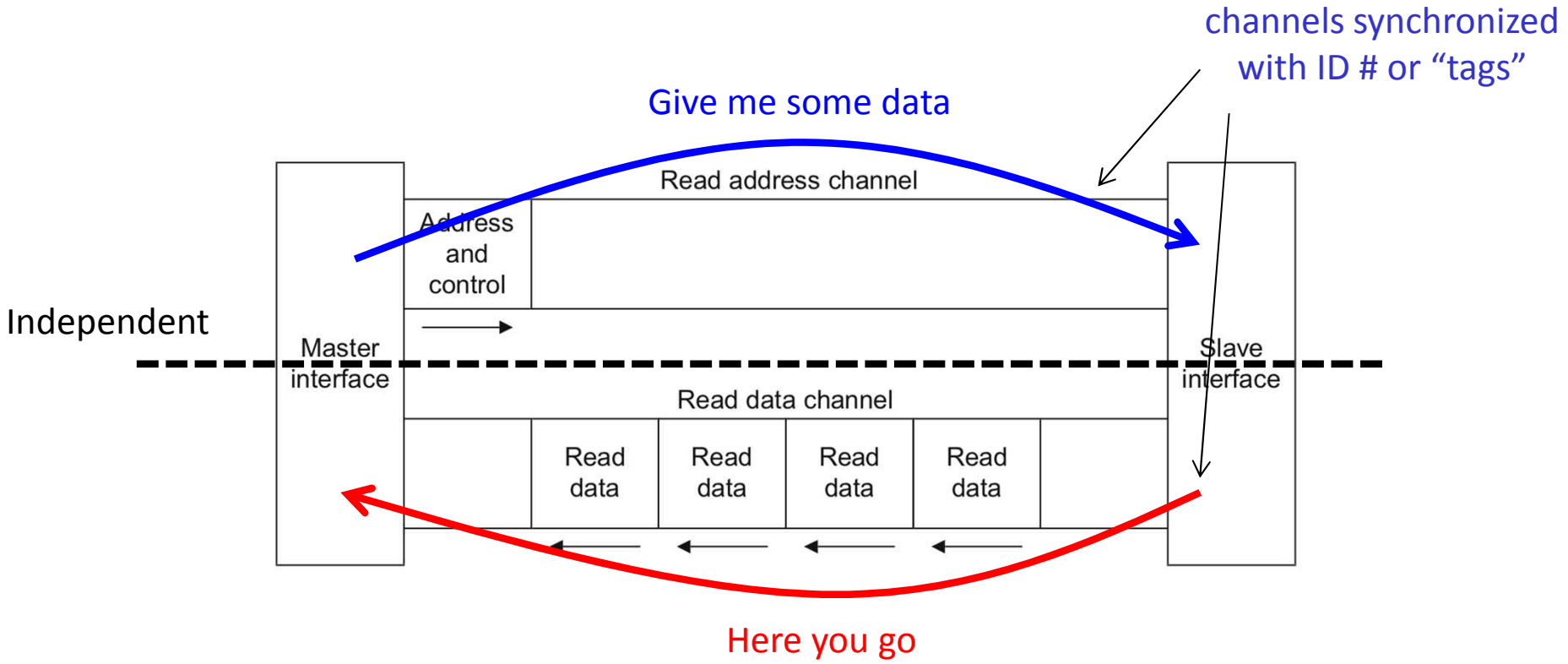
Signal	Source	Description
AWID[3:0]	Master	Transaction id
AWADDR[31:0]	Master	Start address
AWLEN[3:0]	Master	Burst length (1~16)
AWSIZE[2:0]	Master	Data width (1~1024)
AWBURST[1:0]	Master	Burst type (FIXED, INCR, WRAP)
AWLOCK[1:0] / AWCACHE[1:0]	Master / Master	Lock / cache type
AWPROT[2:0]	Master	Protection type
AWVALID / AWREADY	Master / Slave	Handshake control

Write Data & Response Channels



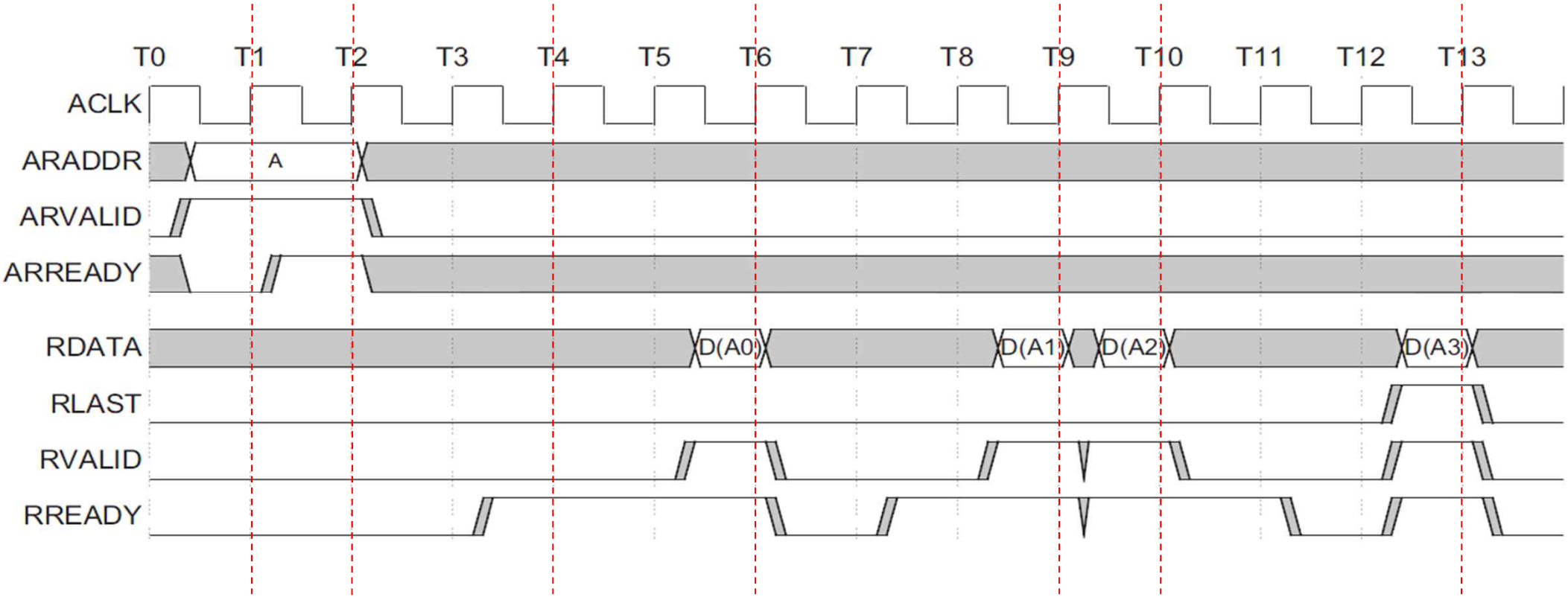
Signal (Write Data Ch.)	Source	Description
WID[3:0]	Master	Transaction id
WADDR[31:0]	Master	Start address
WSTRB[3:0]	Master	Write strobe per byte
WLAST	Master	Write last
WVALID / WREADY	Master / Slave	Handshake control
Signal (Write Response Ch.)	**Source**	**Description**
BID[3:0]	Master	Transaction id
BRESP[1:0]	Master	Write response
BVALID / BREADY	Master / Slave	Handshake control

Read Channels & Signals



- AW signals ~ AR signals
- R signals ~ W signals
  - RLAST is controlled by slave

Read Burst Operation



Read request is initiated

Read request is accepted

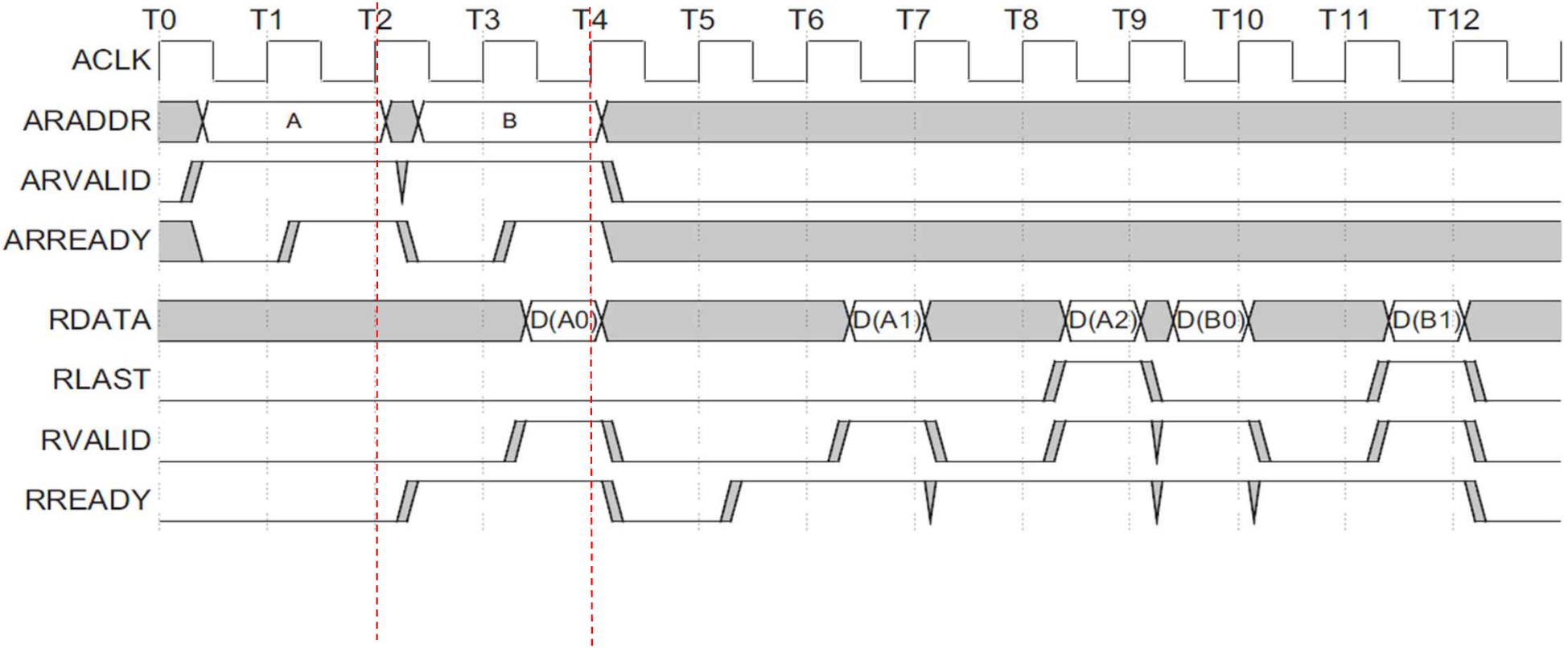
Data read is ready

1st data is transferred

The last data is transferred

Note: data transfer only when valid = ready = 1

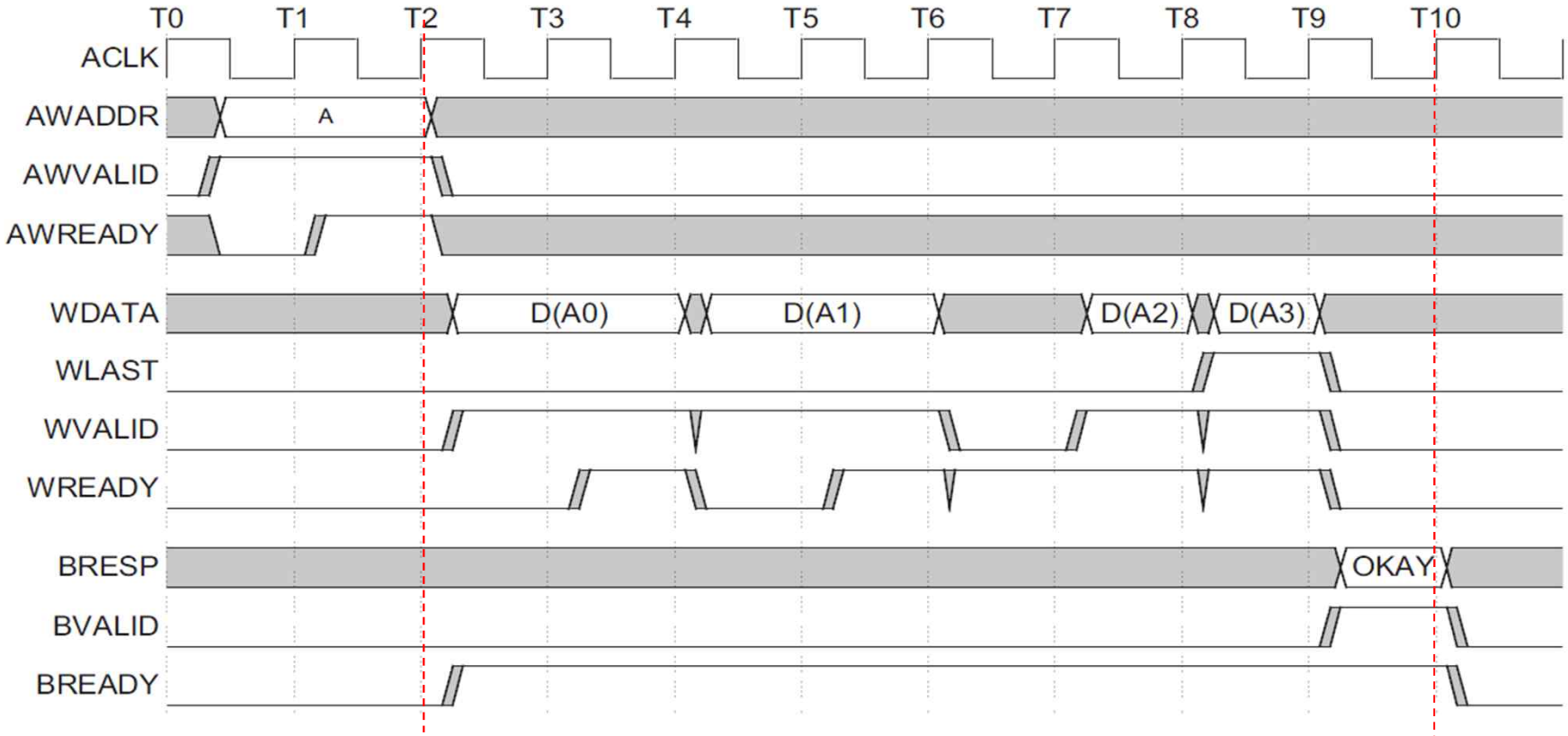
Overlapping Read Bursts



Read request A
is accepted

Read request B
is accepted via AR channel
while data A(0) is
transferred via R channel

Write Burst Operation



Write request A is accepted

Response completes write operation

Cache Support: ARCACHE[3:0] / AWCACHE[3:0]

- Bufferable bit (B): AWCACHE[0]
  - Write delay can be an arbitrary one
- Cacheable bit (C): AR(W)CACHE[1]
  - Read: prefetch or read cache is possible
  - Write: write merging is possible
- Read Allocate bit (RA): ARCACHE[2]
  - If read miss, fetch the data to cache
  - If C=low, RA=low
- Write Allocate bit (WA): AWCACHE[3]
  - If write miss, fetch the data to cache, and then write to the cache (and through the memory)
  - If C=low, WA=low

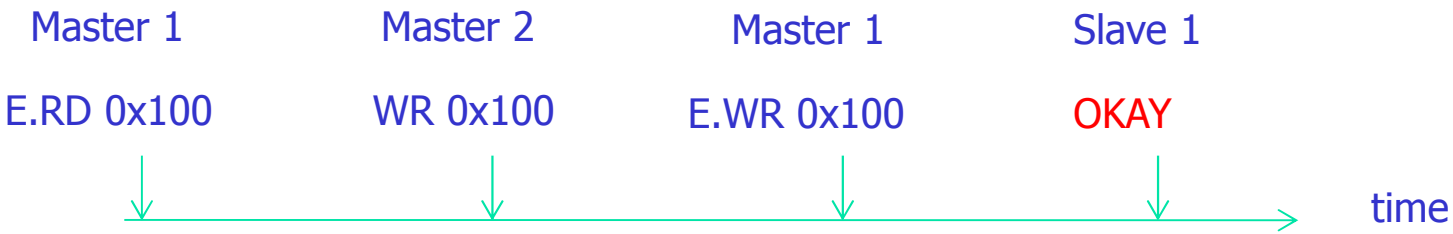
Protection Support

[Source: AXI Spec]

- Normal or privileged: AR(W)PROT[0]
  - High → privileged
- Secure or non-secure: AR(W)PROT[1]
  - Low → secure
- Instruction or data, AR(W)PROT[2]
  - High / low → instruction / data

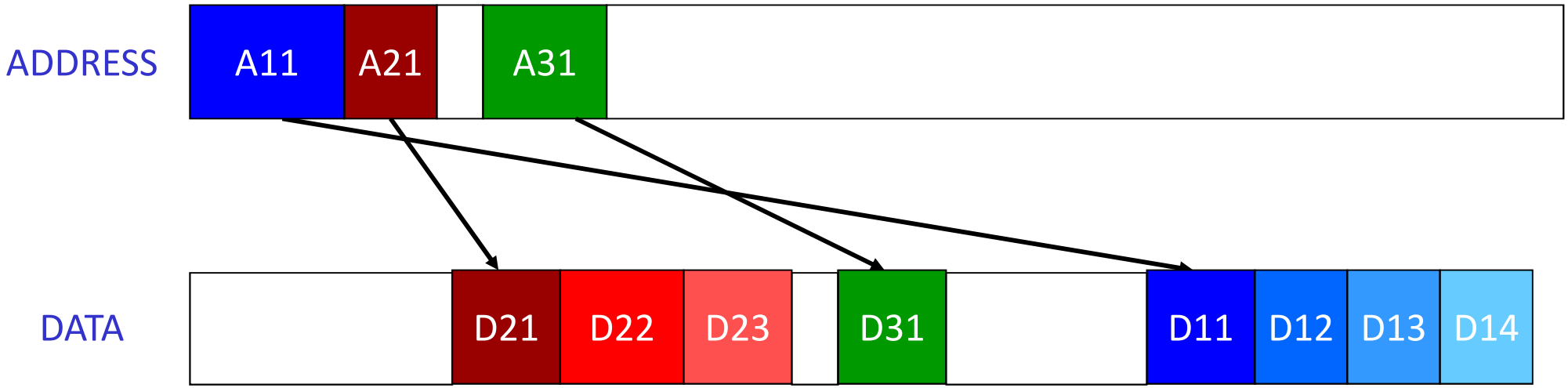
Atomic Access

- Normal access, AR(W)LOCK[1:0]=b00
- Exclusive access, b01
  - Exclusive read … Exclusive write
  - If no intervening write to the address region, EXOKAY response. If not, OKAY response.
  - Usually used for read-modify-write



- Locked access, b11
  - Start with b11, and end with b00
  - During the period, only the lock initiating master can access the address region (not the slave!!!)

Out-of-Order Transaction



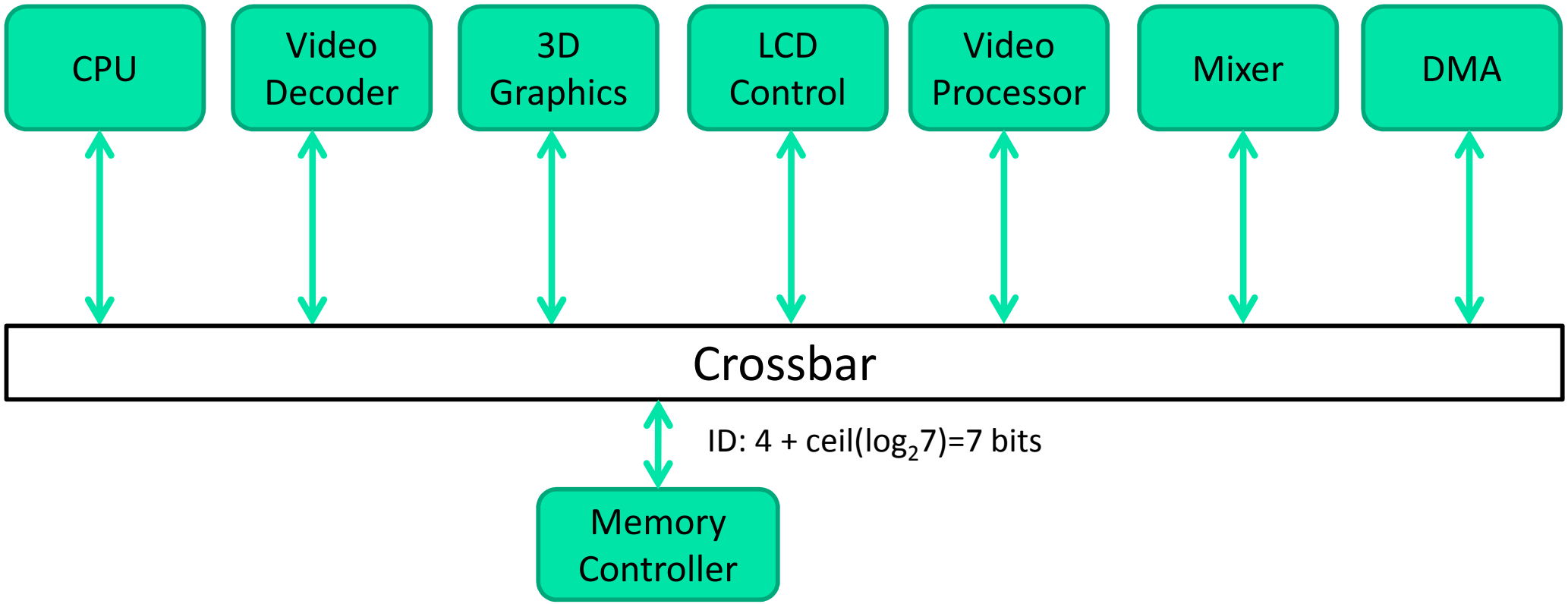
- Transaction ID is used to identify data transfer at all channels
  - ARID, AWID, RID, WID, & BID
  - Up to four bits
- Ordering by transaction ID
  - Master needs to finish data transfers with the same transaction ID in the order of request issue.
  - Slave can handle data transfers with different transaction IDs out-of-order

Ordering

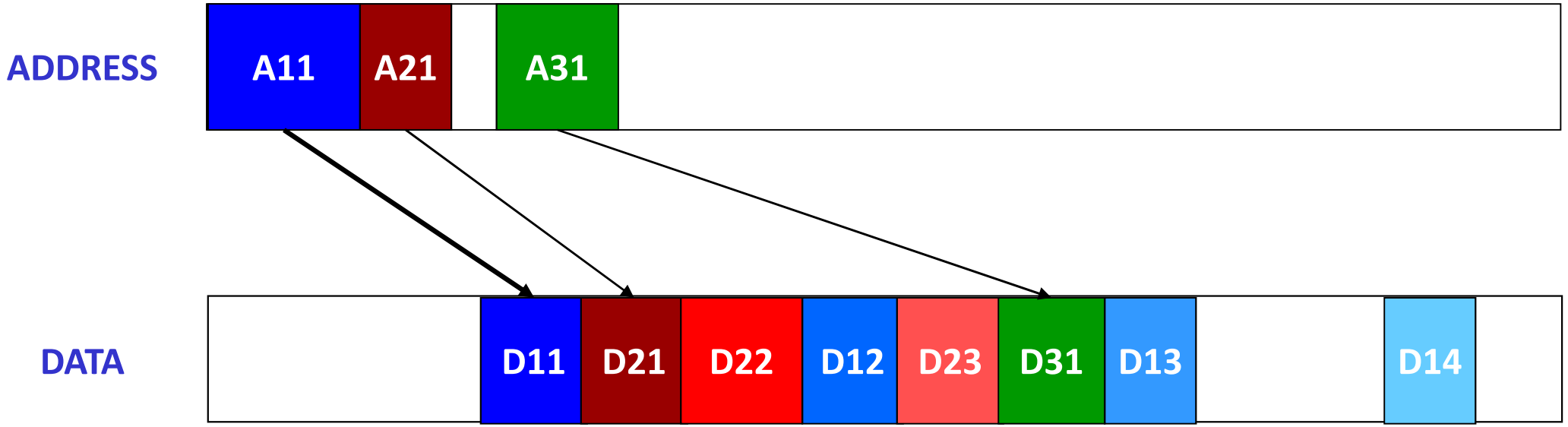
- Transaction id
  - ARID, AWID, RID, WID, & BID
  - Up to four bits
- In-order requirement
  - Transfers with the same id finishes in the same order that they are initiated
- Real implementation
  - Transaction id = <master ID, channel ID>
  - Channel ID = original AXI transaction id
  - Master ID is needed to identify the initiating master among all the masters

Transaction ID Implementation

- Real implementation
  - Transaction ID = <master ID, channel ID>
  - Channel ID = original AXI transaction id
  - Master ID is needed to identify the initiating master among all the masters

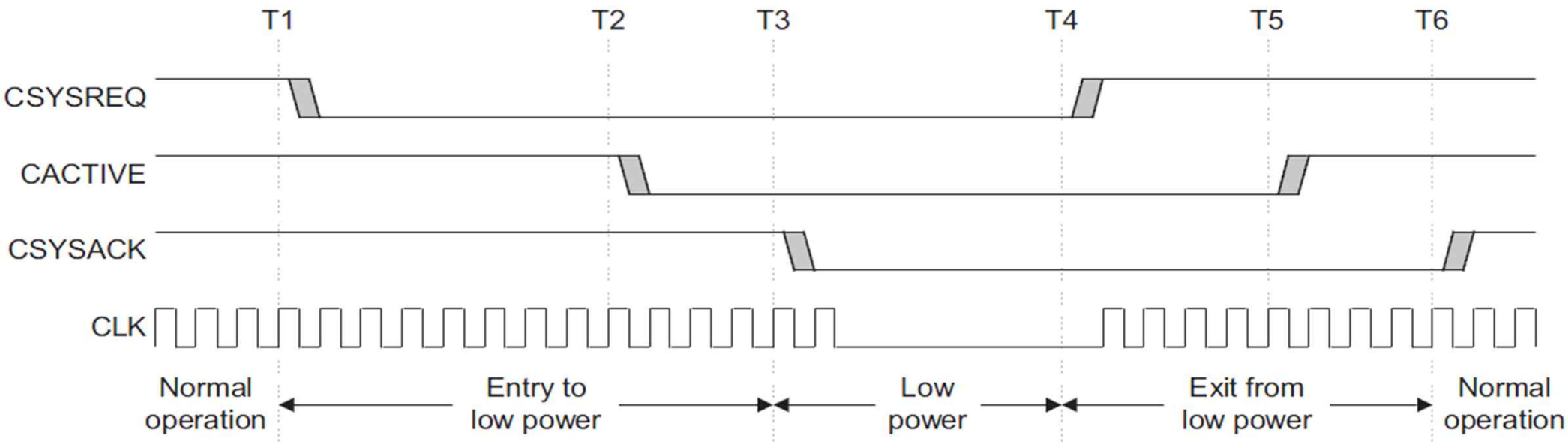


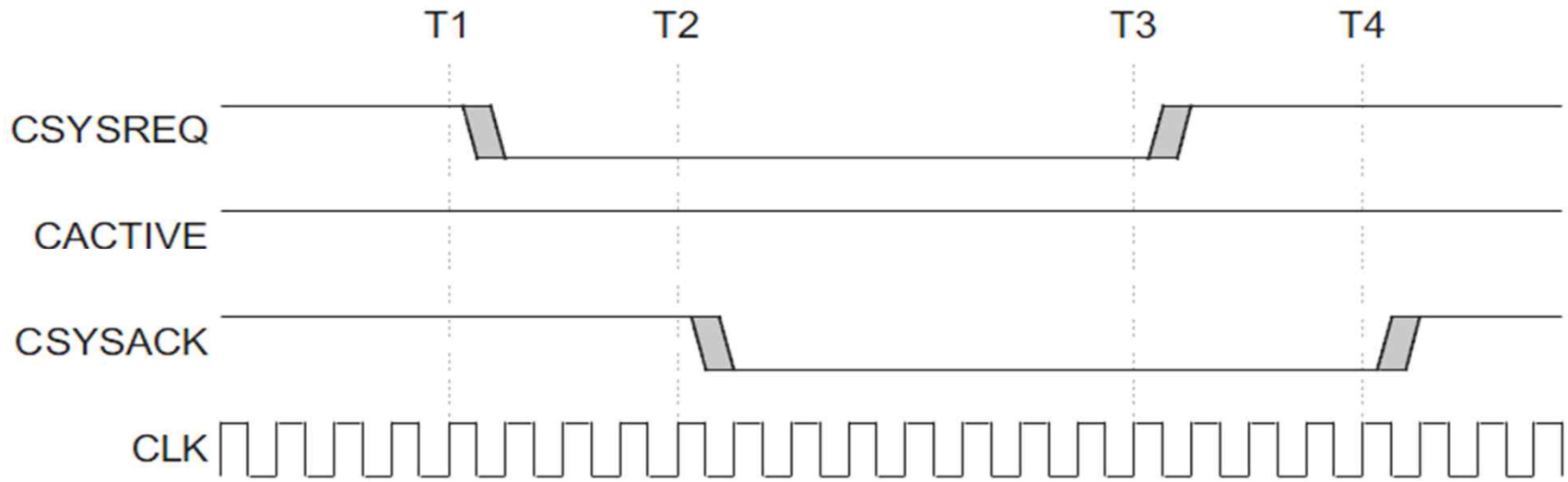
Write Interleaving



- Interleaving rule
  - Data with different ID can be interleaved.
  - The order within a single burst is maintained
  - The order of first data needs to be the same with that of request
  - Write Interleave Cability → The maximum number of transactions that master can interleave

Low Power Interface (C channel)

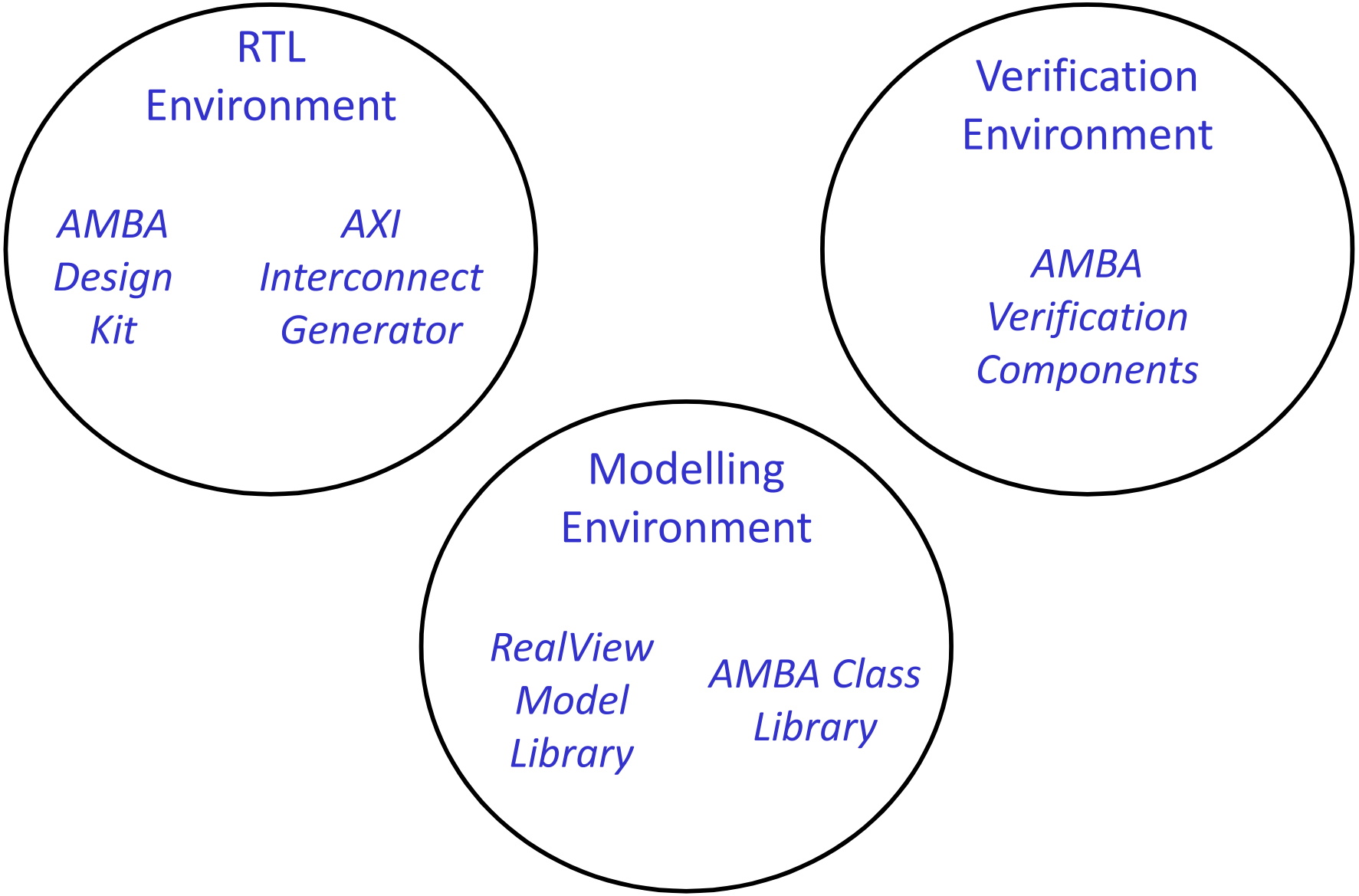




AMBA Class Library

- Defined method for modelling AXI components
  - Data structure used to fully describe a transaction
  - Standard method calls for generating and receiving transactions
- Modelling abstraction levels
  - Programmer view level
  - Programmer's view + timing
  - Cycle callable

AXI Support


RTL
Environment
AMBA
Design
Kit
AXI
Interconnect
Generator
Verification
Environment
AMBA
Verification
Components
Modelling
Environment
RealView
Model
Library
AMBA Class
Library

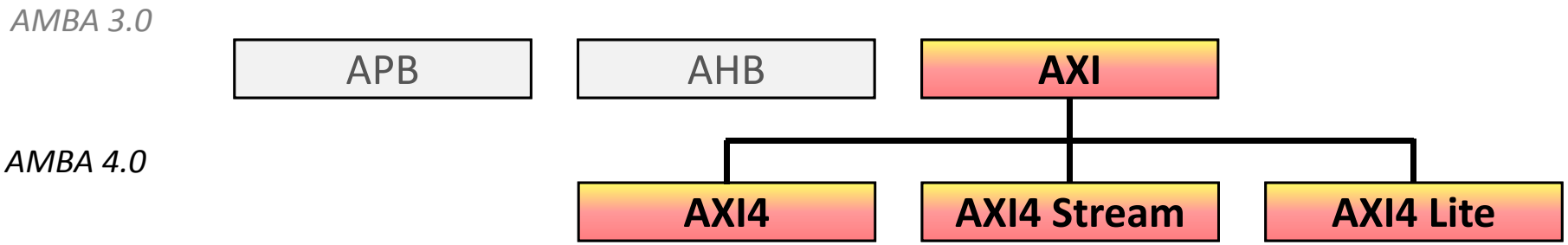
Summary

- AXI is the next generation AMBA bus
  - Channel architecture
  - Registers Slices
  - Burst addressing
  - Multiple outstanding bursts
  - Out of order completion
- Interconnect Options
  - Shared bus, multi-layer and mixed
- More than just a bus protocol
  - RTL, Modelling and Verification environments

AMBA 4.0 AXI Bus

In Brief: AXI4 (1/3)

- Three flavors: AXI4, AXI4-Lite, AXI4-Stream
  - All three share same handshake rules and signal naming



	AXI4	AXI4-Lite	**AXI4-Stream**
Dedicated for	high-performance and memory mapped systems	register-style interfaces (area efficient implementation)	non-address based IP (PCIe, Filters, etc.)
Burst	up to 256	1	Unlimited
Data width	32 to 1,024 bits	32 or 64 bits	any number of bytes
Applications	Embedded, memory	Small footprint control logic	DSP, video, communication

In Brief: AXI4 (2/3)

- Functionality has been added and several known issues in AXI3
  - The maximum burst length has been increased from 16 to 256 transfers for certain types of bursts (INCR, non-exclusive).
  - A Quality-of-Service signaling has been added, where the finer details of the interpretation are implementation defined
- AXI4 defines address regions for slaves, which allows implementations of memory perspectives on the bus level
- Some ordering requirements and transfer dependencies have been refined, as have the meanings of the cache policy signals AxCACHE
  - Abstract memory types as defined by ARMv6/v7 architectures and multicore architectures are much better represented by these changes

In Brief: AXI4 (3/3)

- Legacy (AHB) locked transfers are no longer supported, which does not fit within the idea of a switched interconnect
  - AXI4 does not support locked transactions but, an AXI3 implementation must support locked transactions
- A rather significant change seems to be the banning of write interleaving, which could help improve the system throughput
  - Write interleaving is hardly used by regular masters but can be used by fabrics that gather streams from different sources. With the new AXI4-Stream protocol, write interleaving is still available for fabrics

In Brief: AXI4-Stream

- The new AXI4-Stream protocol was designed for streaming data to destinations that are not memory mapped internally
  - Display controllers, transmitters, but also routing fabrics are among the target applications for this new protocol
- Building upon the proven simple AXI channel handshake AXI4-Stream is essentially an AXI write data channel with additional control signals and a slightly modified protocol
  - The burst (packet) length is not restricted and the number of bytes of the data signal TDATA can be an arbitrary integer including zero

In Brief: AXI4-Lite

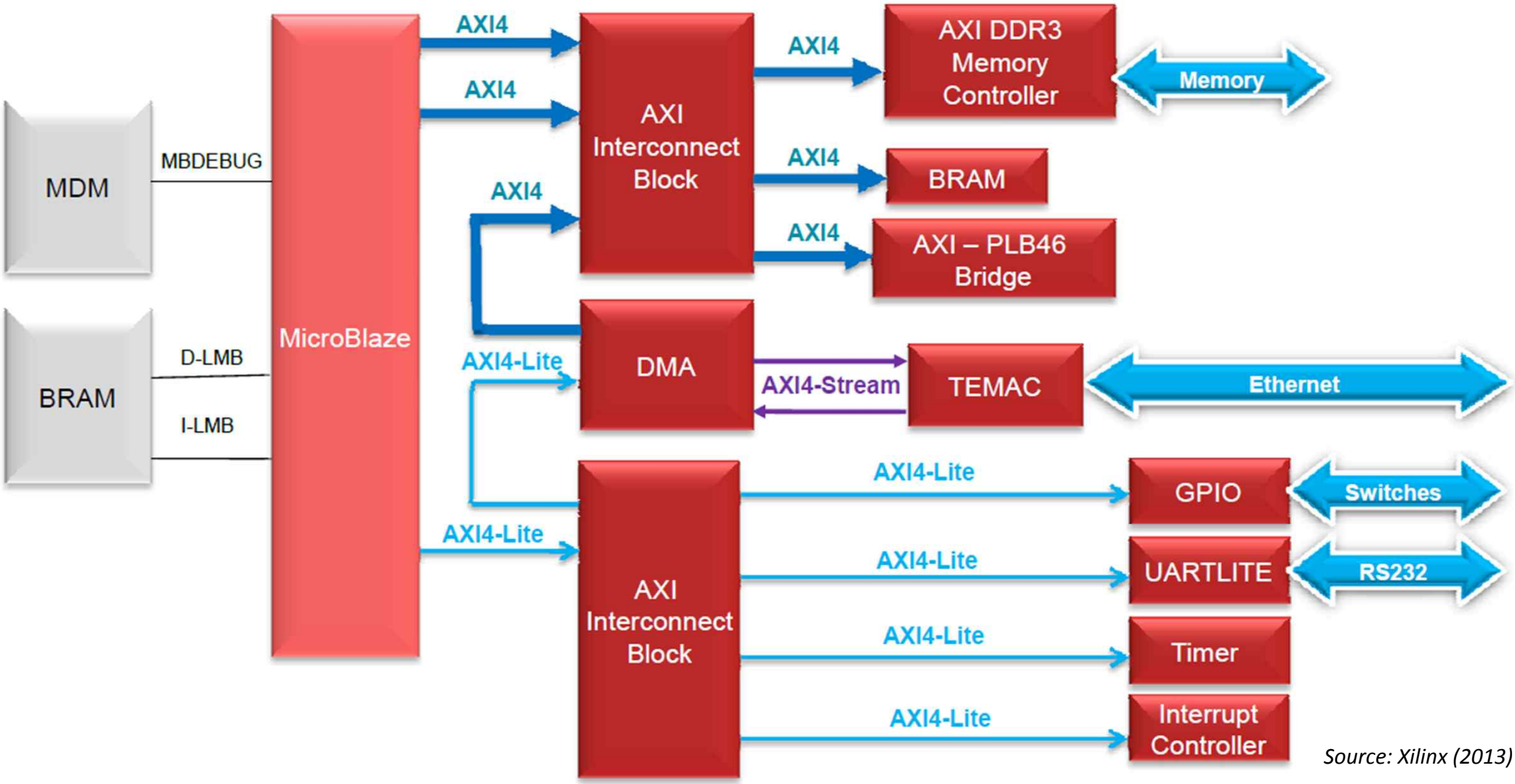
- The focus of AXI has been on high-performance data transfer, but what about the low-end hardware registers, configuration, etc like APB?
  - The issue with APB is the bridge
- AXI4-Lite addresses this last issue by defining certain restrictions that would allow a slave to be connected directly to an AXI fabric
- In AXI4-Lite, you might say that AXI gets "dumbed" down to a few basic transaction types
  - The burst length is fixed to one data transfer, transfers are non-cacheable and non-bufferable, exclusive access is not allowed and access width must always be the same as data bus width
  - This is supposed to make the interface design simple enough to be implemented quickly in custom IP

In Brief: ACE

- The ACE protocol extends the AXI4 protocol and provides support for hardware-coherent caches
  - A five state cache model to define the state of any cache line in the coherent system
    - The cache line state determines what actions are required during access to that cache line
  - Additional signaling on the existing AXI4 channels that enables new transactions and information to be conveyed to locations that require hardware coherency support
  - Additional channels that enable communication with a cached master when another master is accessing an address location that might be shared
- The ACE protocol also provides:
  - Barrier transactions that guarantee transaction ordering within a system
  - Distributed Virtual Memory (DVM) functionality to manage virtual memory

What's New in AMBA4

- Increased productivity
  - Supports memory mapped, control and streaming applications



Source: Xilinx (2013)

What's New in AMBA4:

- New Stream Interface
- AHB absent
- New AXI-Lite spec
- New AXI Features
- AXI Feature removal
- AXI clarifications

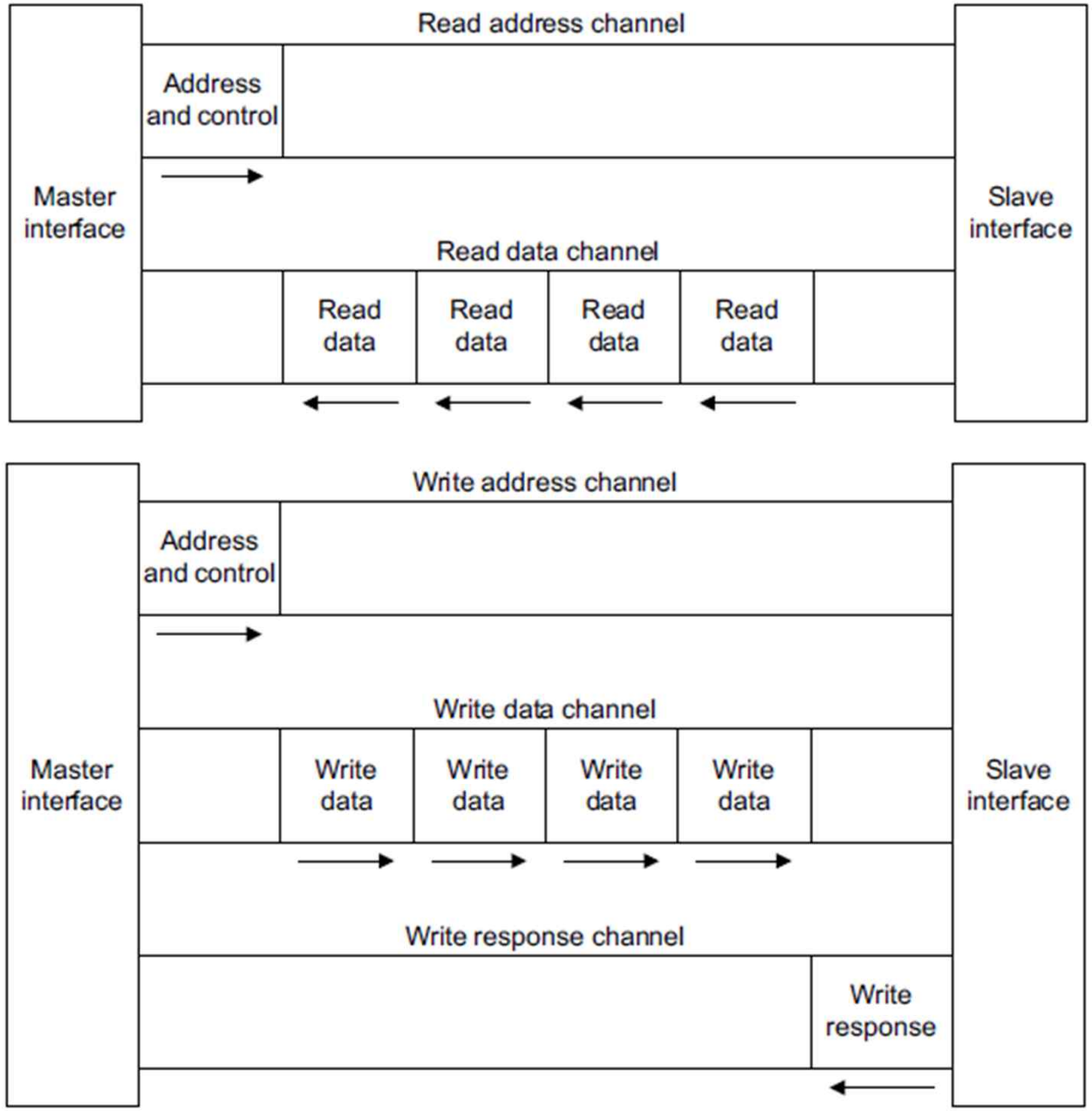
AXI Changes

- Clarifications
  - ordering, posting, memory types / cache impacts
- Removal of locked transactions
- Removal of write data interleave
- New features
  - longer bursts
  - QoS field
  - User fields

Longer Bursts

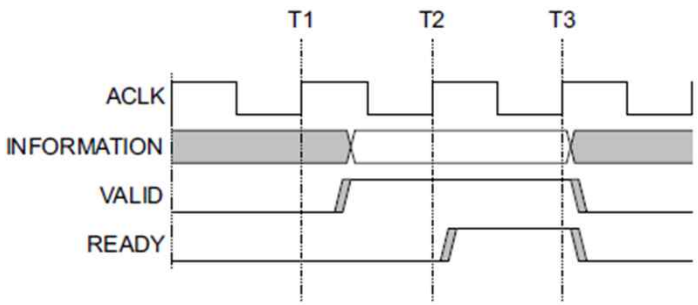
- Can issue up to 256 beat bursts
  - performance, atomicity, simplicity
- Not guaranteed atomic beyond 16 beats
  - even in device space / non-modifiables
  - system (integrator) performance & QoS
- Atomicity may be undesirable for core as well

Basic AXI4 Signaling: 5 Channels, Point-to-Point

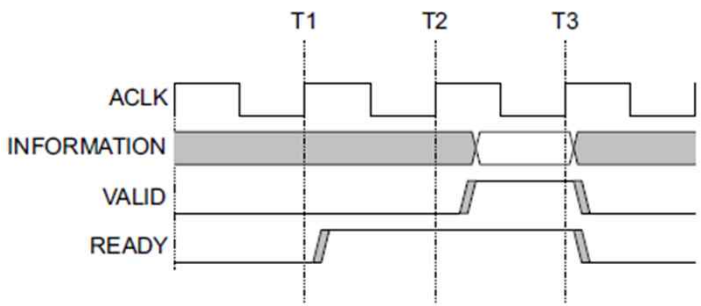


Basic AXI4 Handshaking

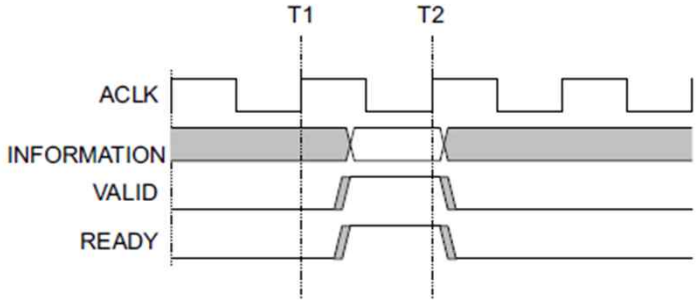
- Master asserts and holds VALID when data is available
- Slave asserts READY if able to accept data
- DATA and other signals transferred when VALID and READY = 1
- Master sends next DATA/other signals or deasserts VALID
- Slave deasserts READY if no longer able to accept data



VALID before READY handshake



READY before VALID handshake

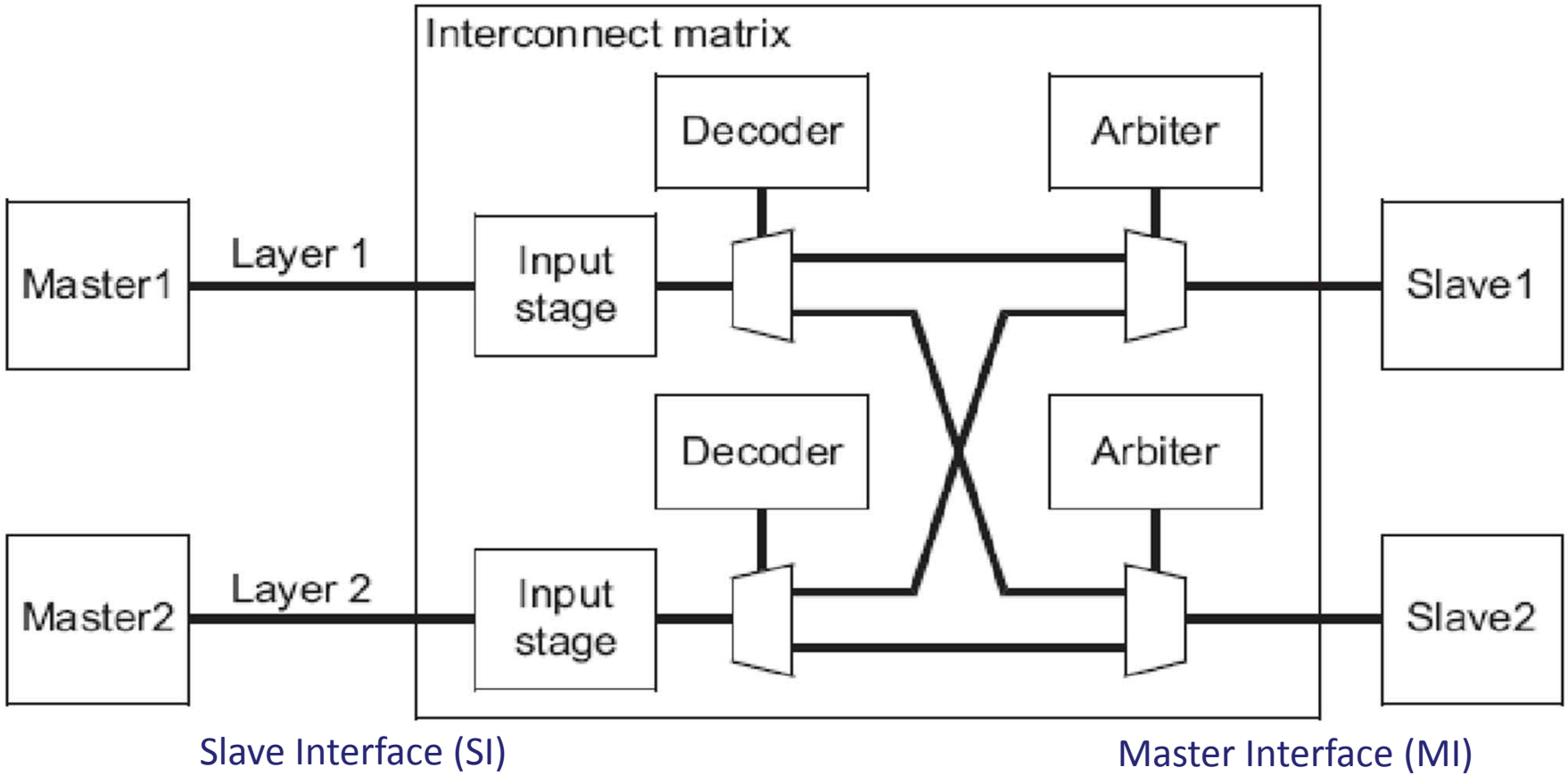


VALID with READY handshake

Appendix:
PL301 Crossbar Bus

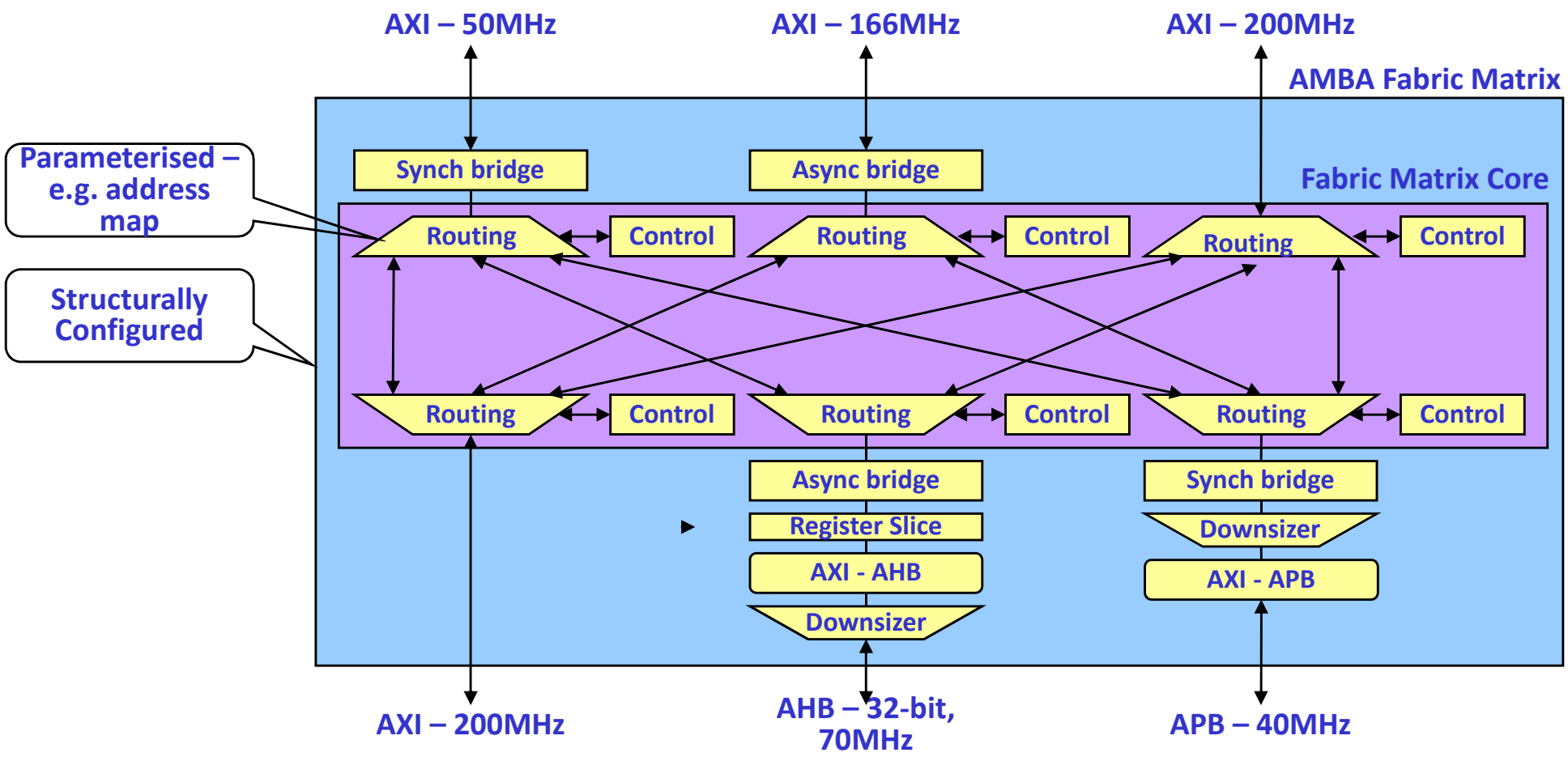
AXI Crossbar Bus: ARM PrimeCell PL301

- PL301: PrimeCell High Performance Matrix



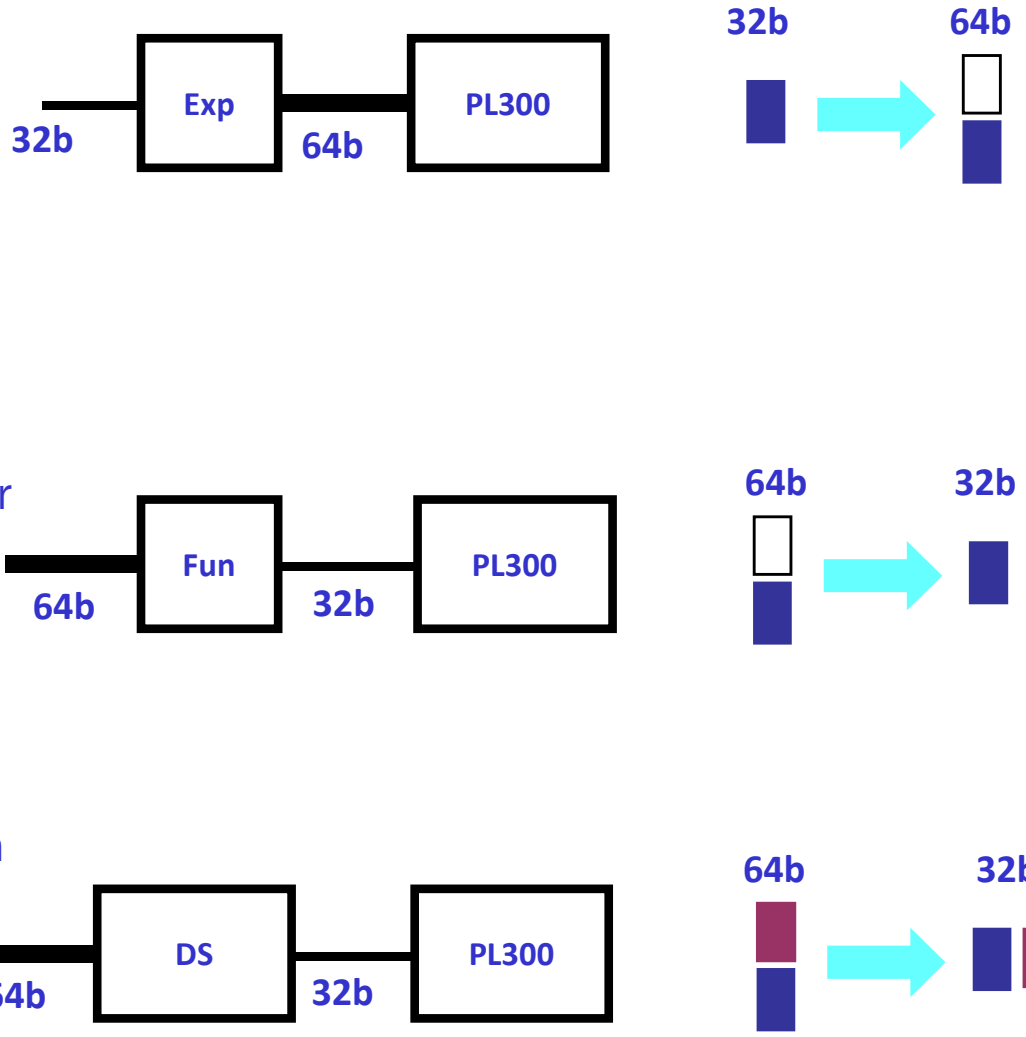
PL301 Internal Structure

- Adjust interface differences
  - Data width, frequency, and protocol



Adjusting Data Width

- ExpanderAxi
  - Converts a narrow master for use on a wide bus
  - Data replication on write data and muxing on read path.
- FunnelAxi
  - Converts a narrow slave for use on a wide bus
  - Mux on write data path and replication on read data path
  - Assumes that transfers are suitably sized for slave (i.e. no wider than the narrow bus)
- DownSizerAXI
  - Converts a wide bus to a narrow bus
  - Transaction <= narrow bus pass through
  - Transactions > narrow bus are broken down in to smaller transactions.

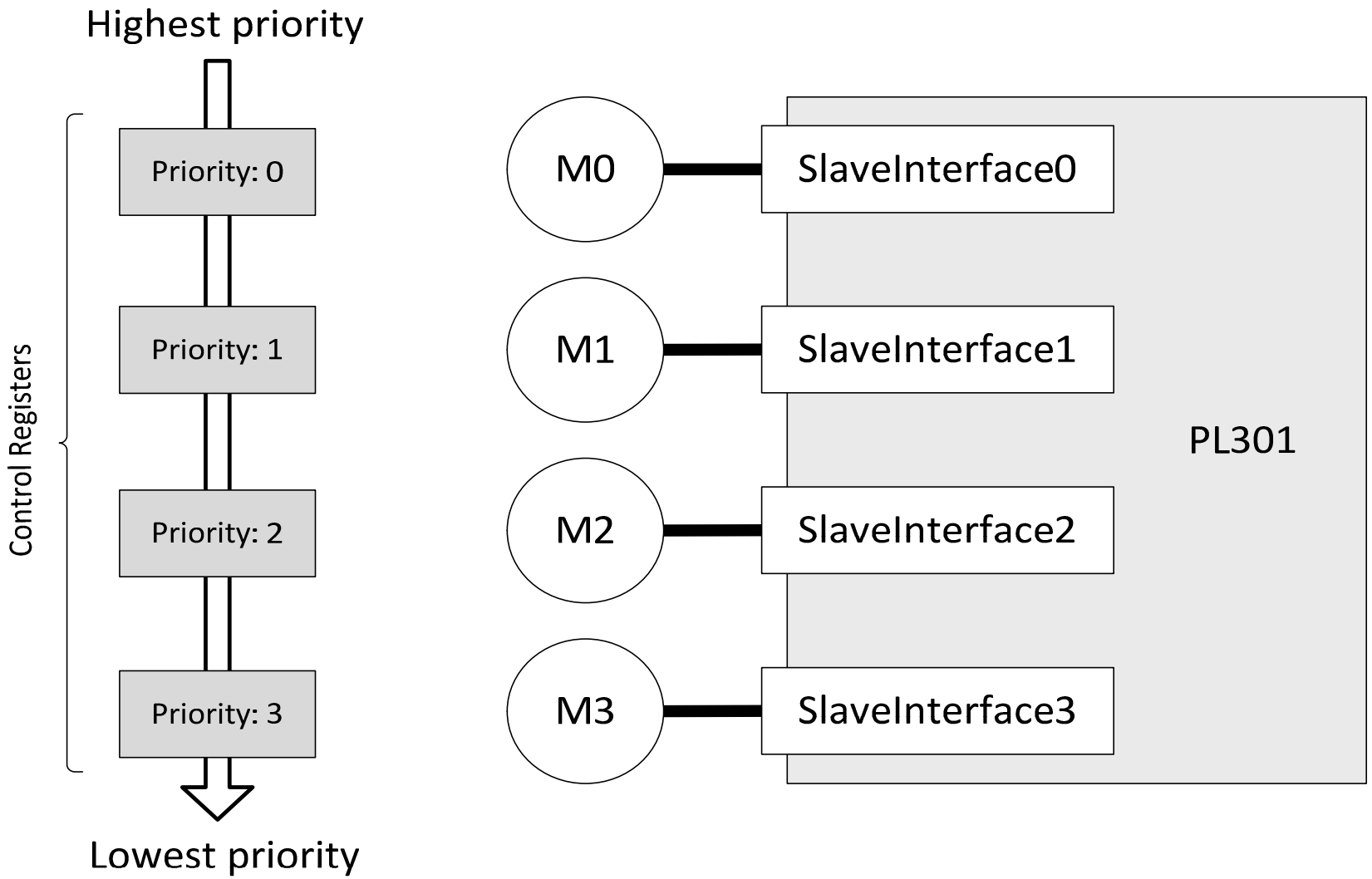


PL301 Features

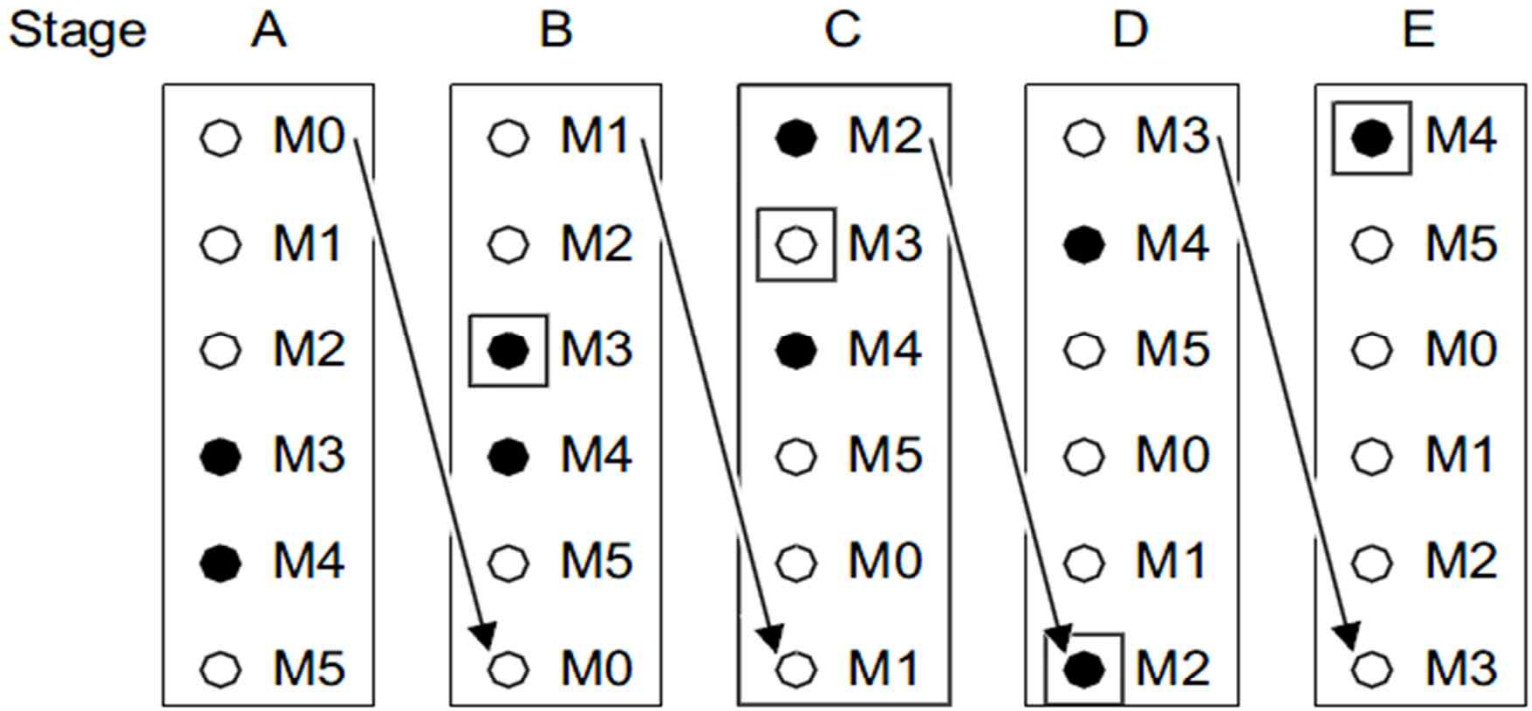
- Configurable number of SIs and MIs
- Sparse connection options to reduce gate count and improve security
- Configurable AXI address/data widths
- Decoded address register that you can configure for each SI
- Flexible register stages to aid timing closure
- An arbitration mechanism that you can configure for each MI, implementing:
  - a fixed *Round-Robin (RR) scheme*
  - a programmable RR scheme
  - a programmable scheme that provides prioritized groups of *Least Recently Granted (LRG) arbitration*
- A programmable *Quality of Service (QoS) scheme*
- Support for multiple clock domains: synchronous and asynchronous.
- Configurable cyclic dependency schemes to enable a master to have outstanding transactions to more than one slave

[Source: PL301 TS]

Arbitration Scheme: Fixed Priority



Arbitration Scheme: Round Robin



○ Indicates an inactive master

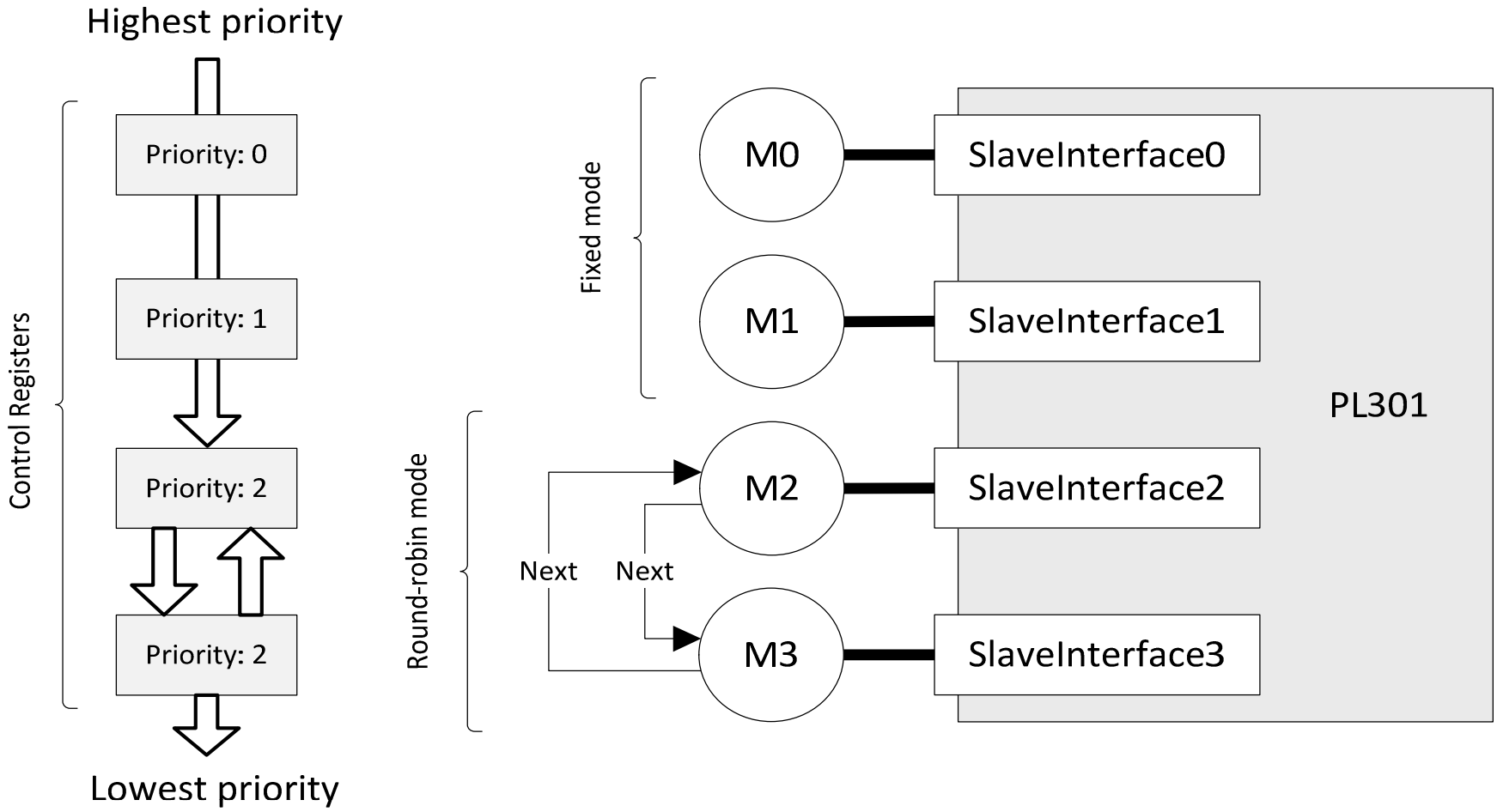
● Indicates an active master

□ Indicates the master that won arbitration in the previous cycle

[Source: PL301 TS]

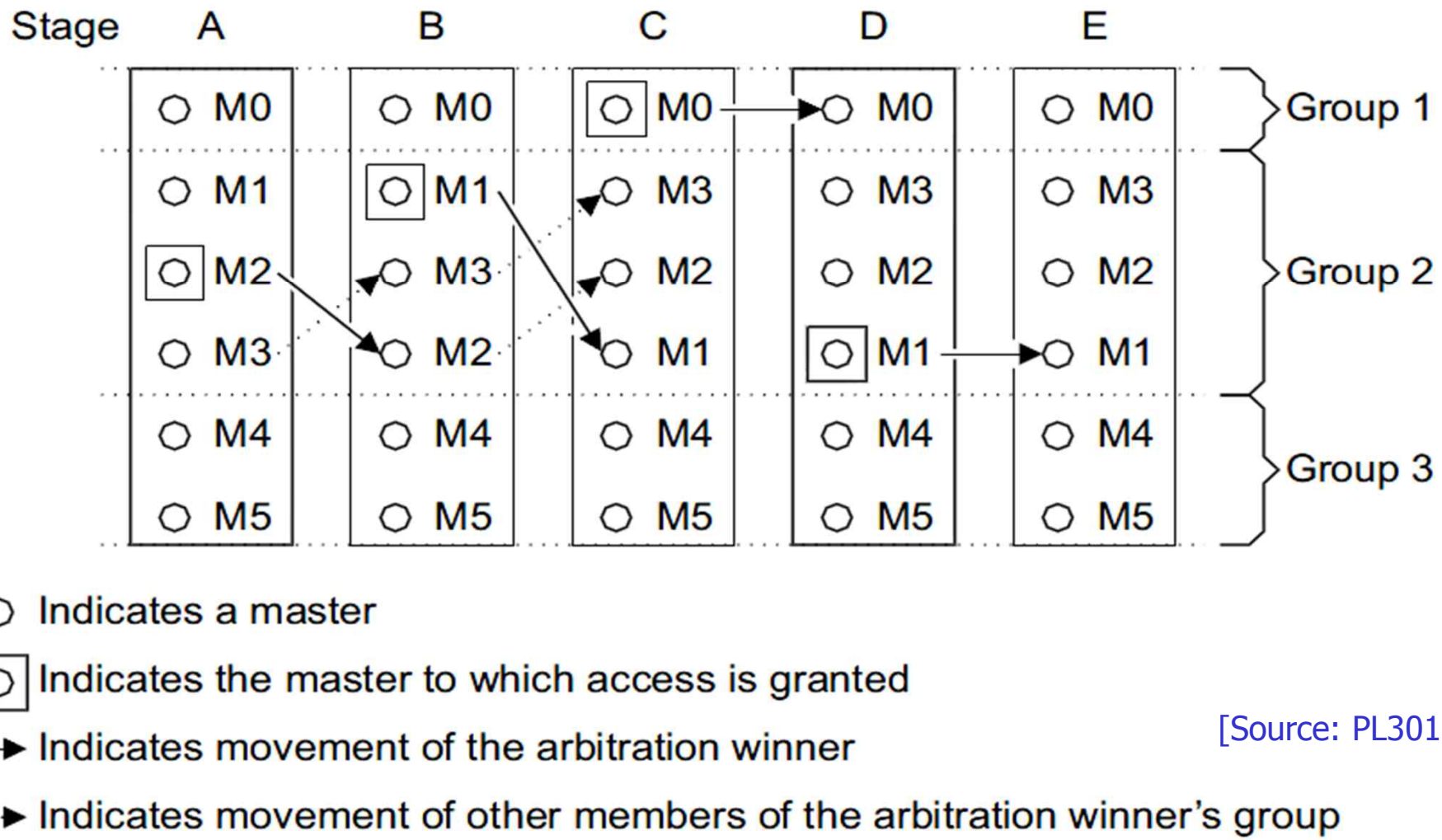
Arbitration Scheme: Hybrid

- Combination of round robin and fixed priority



Arbitration Scheme

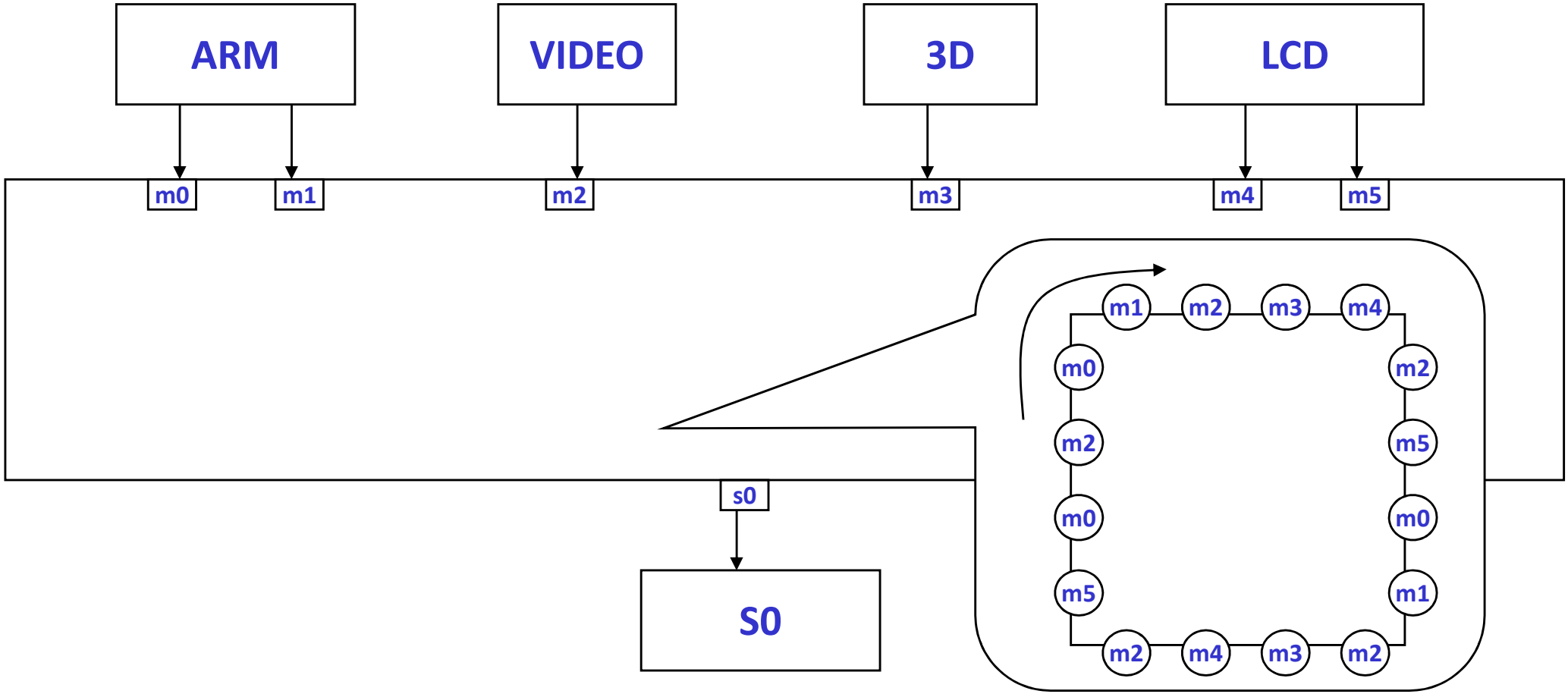
- LRG (least recently granted) scheme



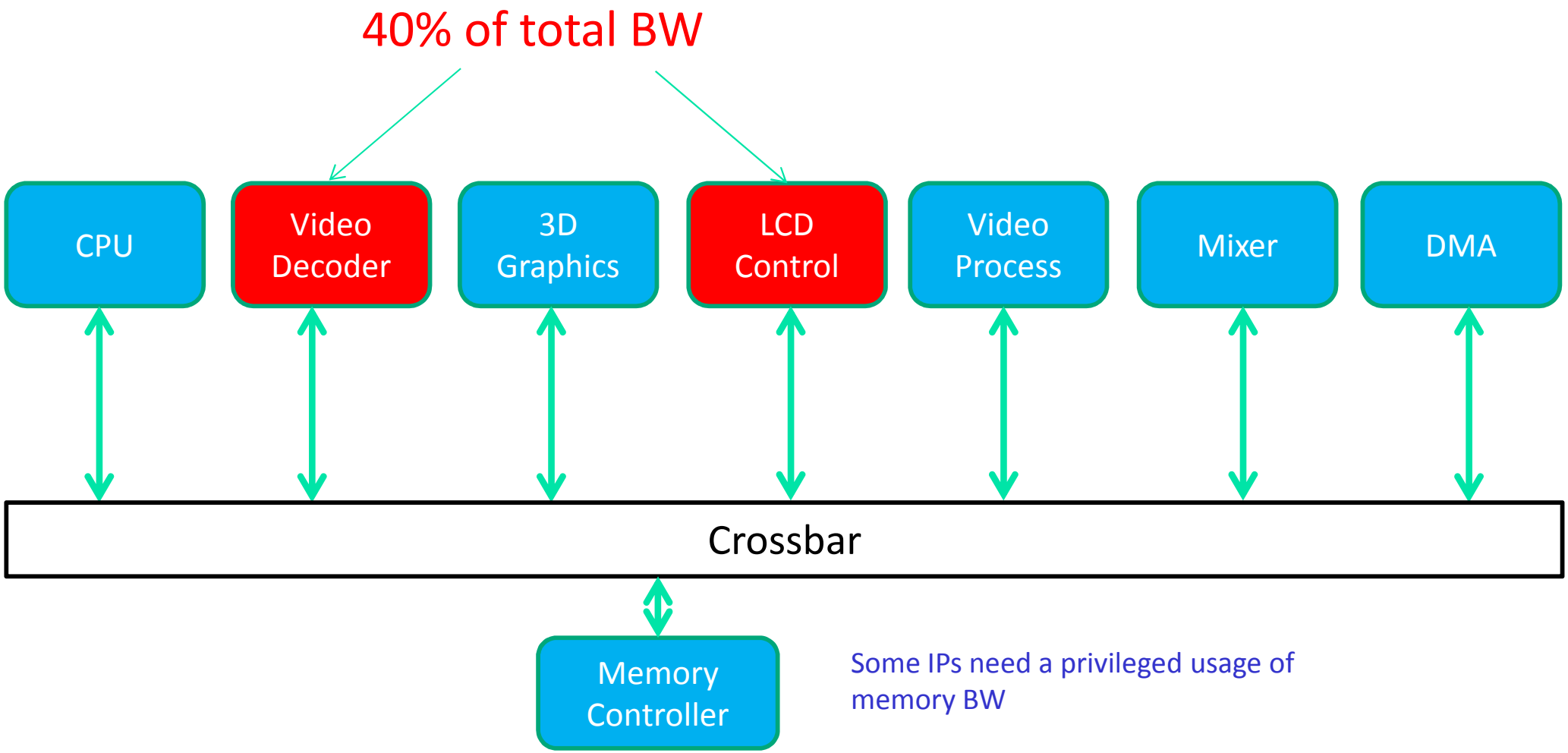
[Source: PL301 TS]

Arbitration Scheme: Fixed Round Robin

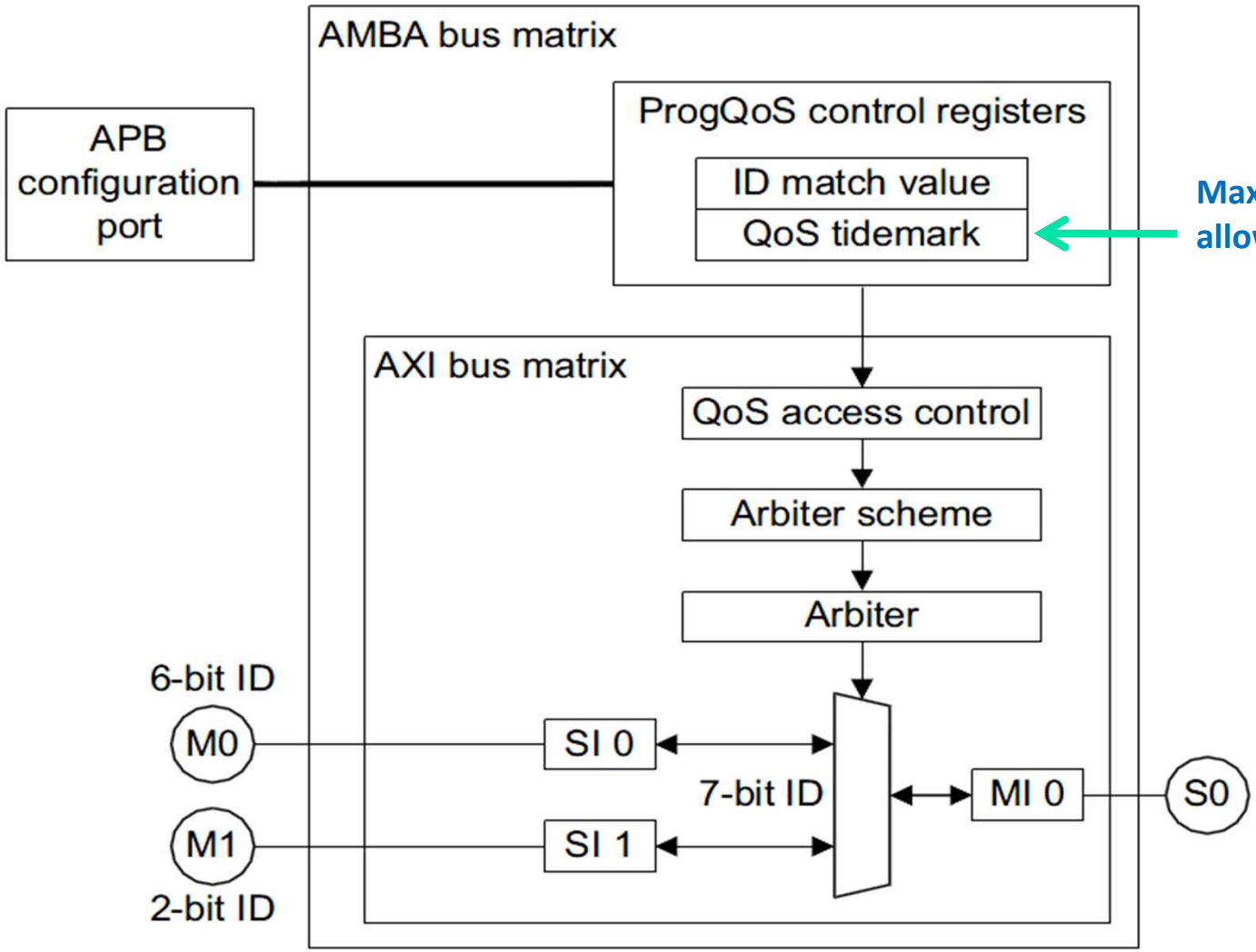
- A weighted round robin in a fixed order



An Example of Crossbar Bus Design



Programmable QoS



Maximum # of requests allowed for best effort traffic

Assume Tidemark = 4, and ID match = M0. If there is 4 outstanding requests for M1, then only requests from M0 are accepted by S0 until one of M1's requests is served

[Source: PL301 TS]

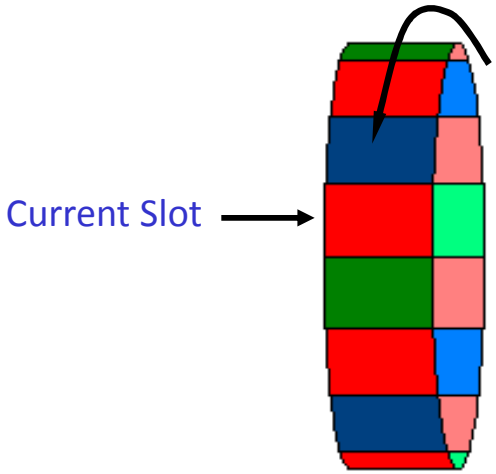
Bus Arbitration: A Generic Arbiter

- Assumptions
  - Each request has its own performance requirement (e.g., bandwidth budget and/or latency)
    - E.g., low latency access from CPU
    - E.g., bandwidth guarantee for LCD / Camera controllers
  - In order to avoid starvation, a global time out is applied
- Priority order: time-out > bandwidth > best effort
  - Time-out (TO) request: TO = 0, and BW budget > 0
  - Bandwidth (BW) request: TO > 0, and BW budget > 0
  - Best effort (BE) request: BW budget = 0
- Priority promotion/demotion
  - Demotion: if BW budget is exhausted, demotion to BE
  - Promotion: if BW budget becomes positive, promotion to BW or TO request
- Time-out counters
  - One type of counter for QoS access w/ time out
  - The other for old request
    - When a normal request (w/ unspecified TO) arrives at the bus, a timer is assigned and starts to be decremented each cycle.

Bus Arbitration: A Generic Arbiter

- If there is any TO request, it is served
  - QoS request w/ time out
  - Old request
- If there is any BW request
  - Give the bus to the BW requests based on BW budget
    - Based on fixed time slot allocation or statistical slot allocation
- To the other BE requests, apply the same priority order that BW requests

Higher Priority

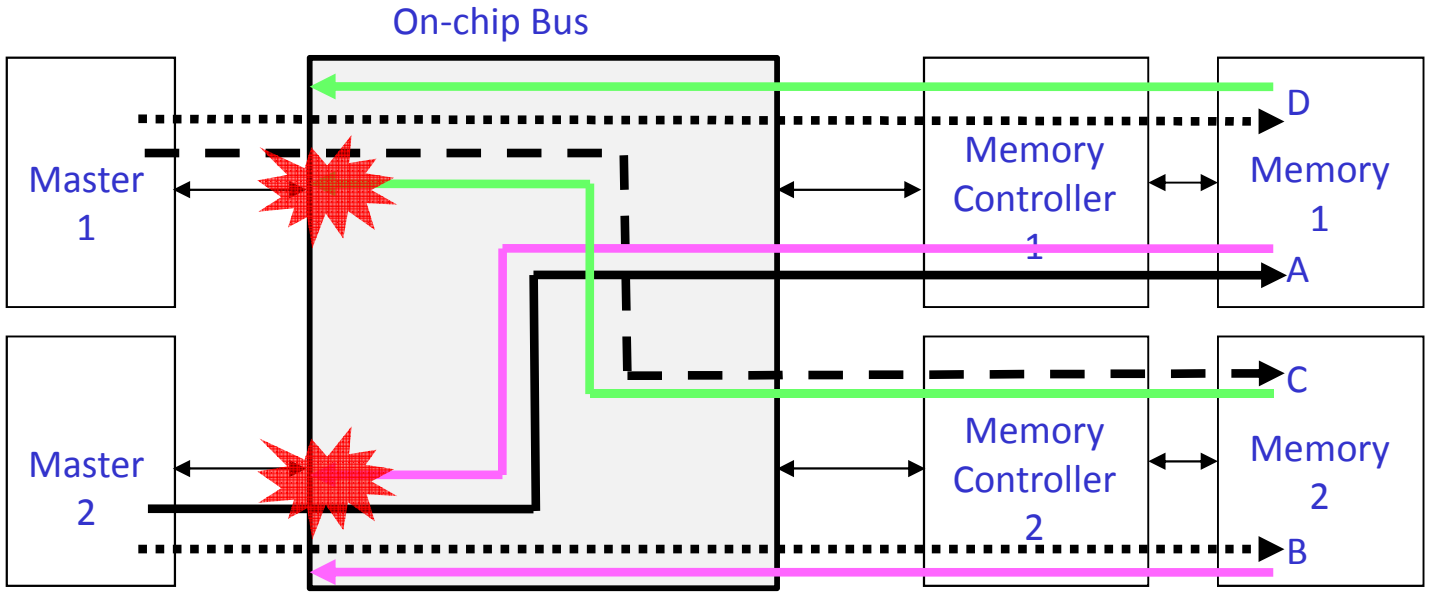


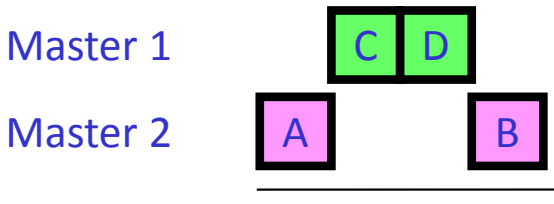
Case of fixed time slot allocation

At a time slot (instant or period)
if the master allocated to the slot has a request,
  then it is served
else
  the other pending requests are served
  (e.g., round robin)

Note: Arbitration scheme is similar to the one in memory access scheduling

A Deadlock Problem in Accessing Multiple Slaves



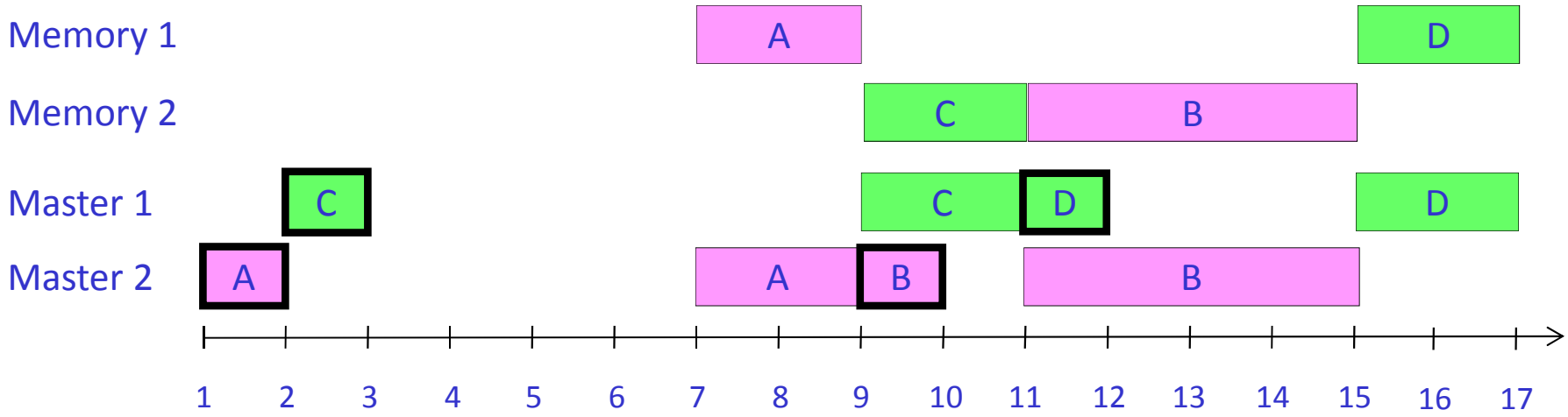


Optimization in Memory Controller

Memory controllers can serve **independent** requests **out-of-order** to increase memory utilization or to lower memory access latency

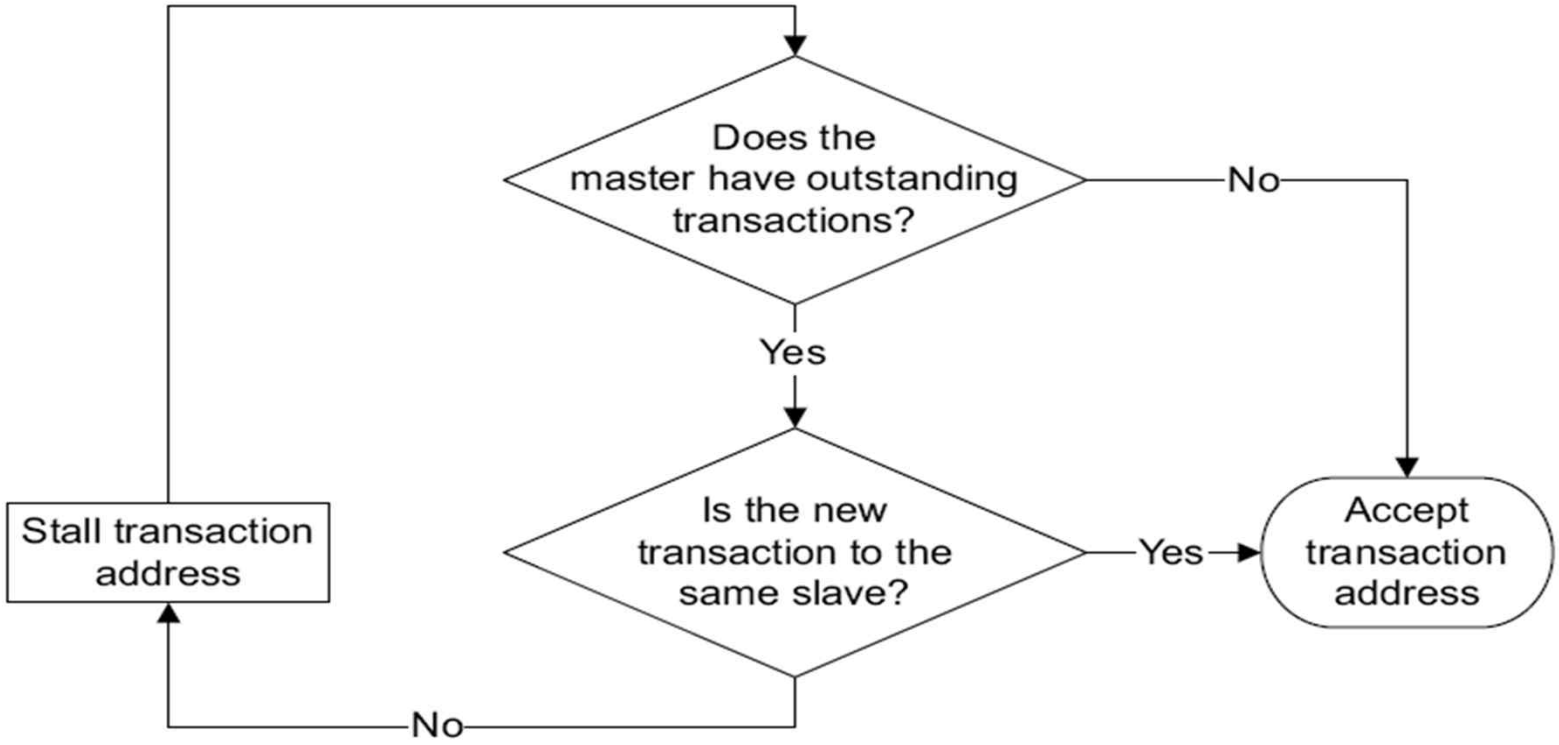
Cyclic Dependency Schemes

- Outstanding requests are permitted only for a single slave per transaction id
- The deadlock problem is resolved while limiting parallel (memory) accesses



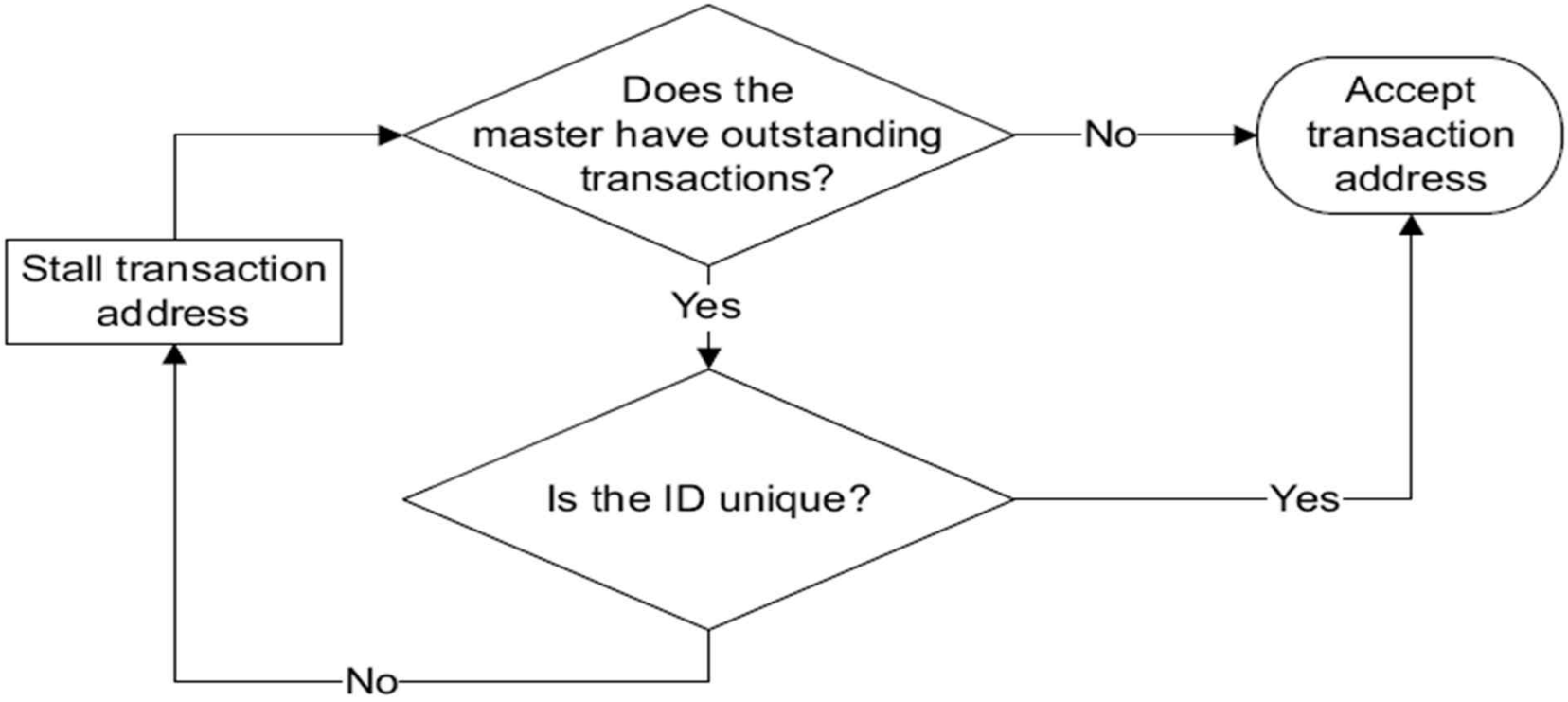
Single Slave Scheme

- Allow multiple outstanding transactions only to the same slave



Unique ID Scheme

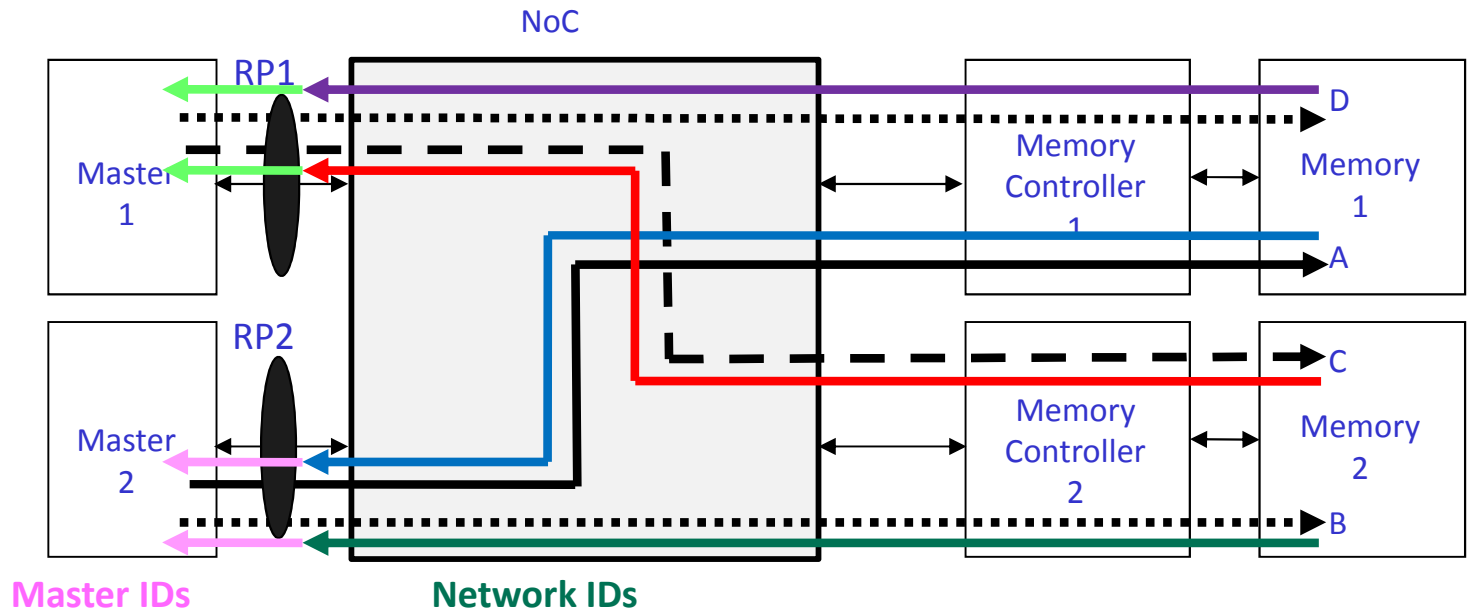
- Accept only out-of-order requests, i.e., requests with different transaction ID's



Single Slave per ID

- Combination of both single slave and unique ID schemes
- Allow multiple outstanding requests to a single slave per transaction ID

Request Parallelizer (RP) for Transaction ID Renaming

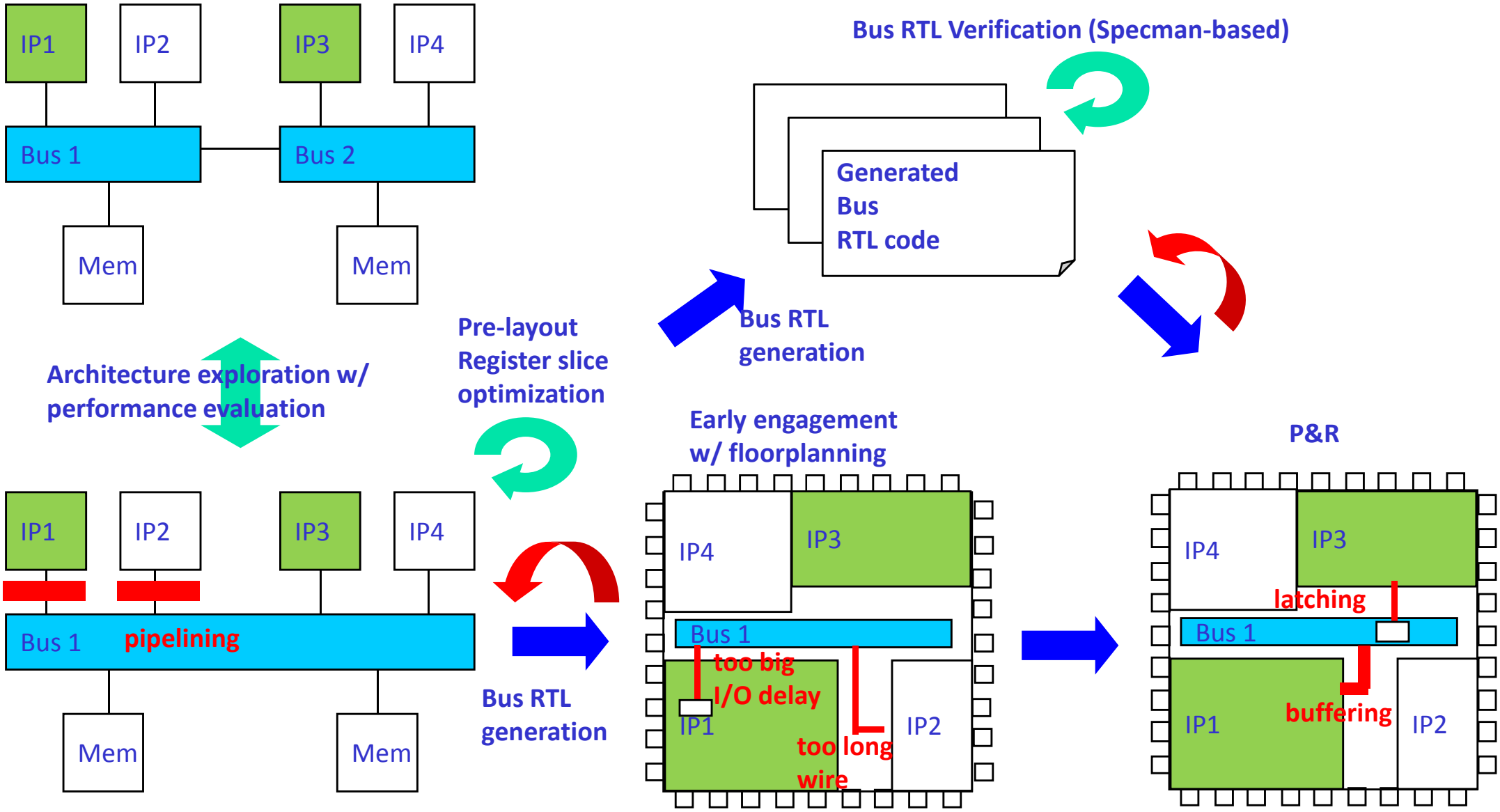


Rename Transaction IDs at NoC Boundary to Enable Out-of-Order Transfers in NoC

*Requirement: Reserve Buffer Space in RP for Transaction ID Renaming

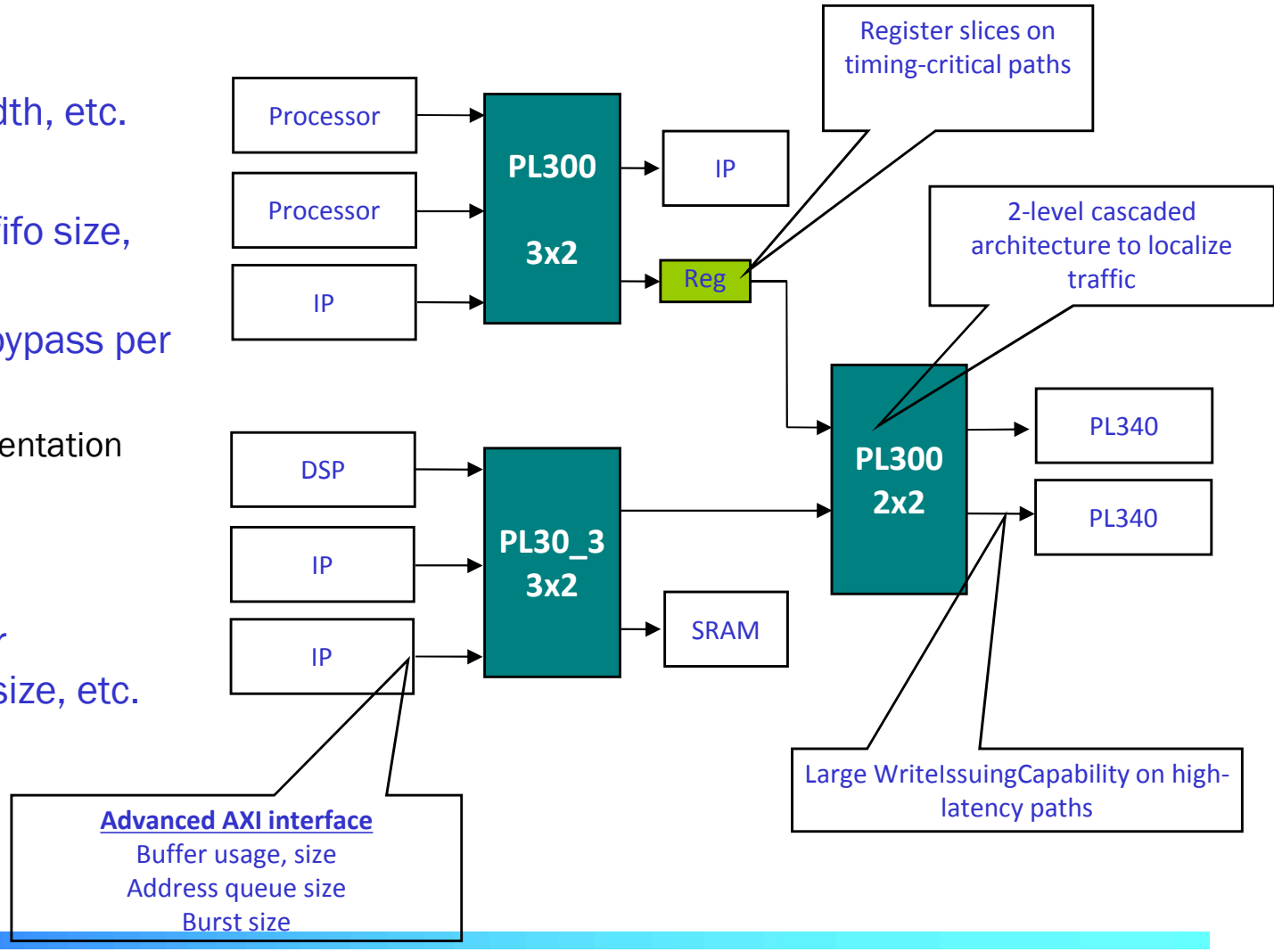
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17

On-Chip Network (OCN) Design Flow



AXI Bus Architecture Design Space

- Interconnection level
  - PL300 decomposition: cascading level
- Bus component level
  - Common parameters: bit width, etc.
  - PL300: WriteIssueCap, etc.
  - PL340: arbiter queue, data fifo size, etc.
  - Register slice: full/forward/bypass per channel
    - Closely related with implementation
- IP interface level
  - # of AXI ports, internal buffer usage/size, address queue size, etc.

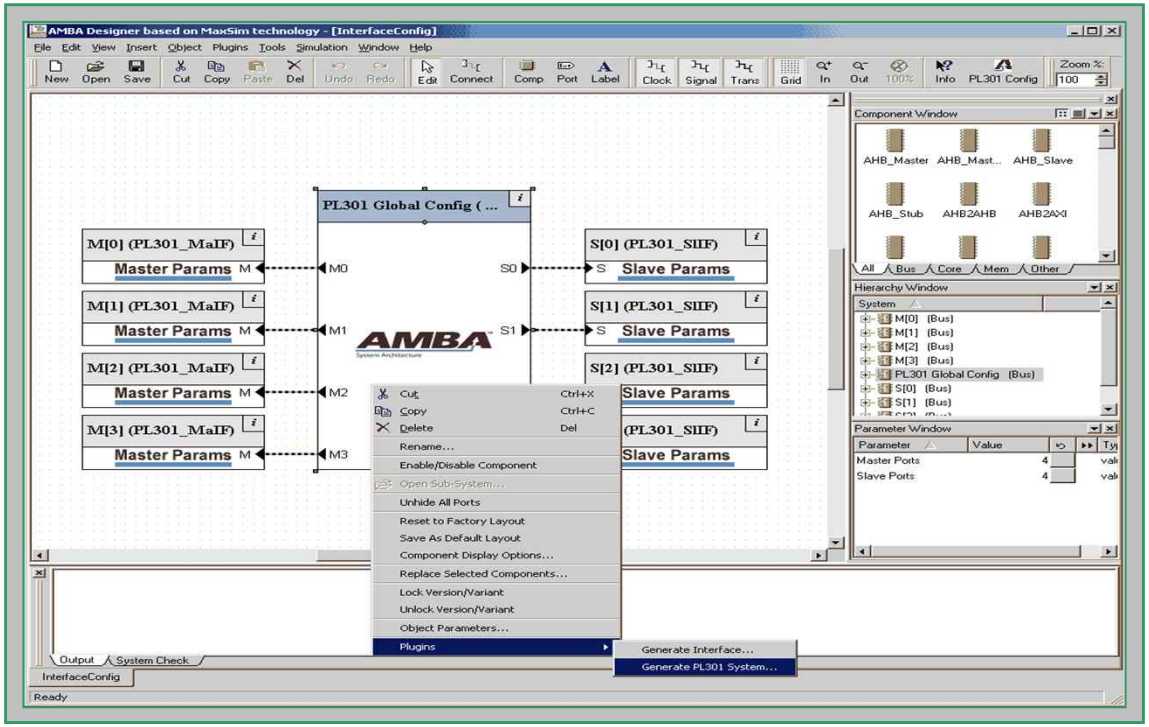


Performance Evaluation of Bus Architecture

- Five methods depending on how to model the traffic of bus masters
  - ViP (virtual platform)-based methods
    - Random bus traffic model
      - ViP version of Sonics' random pattern generator
    - Sequence-based bus traffic model
      - ViP version of Specman-based model in DTV team
    - Timed model
      - Bus masters are modeled in cycle accuracy as ViP models
  - HW Integrator
    - Bus masters are emulated on FPGA
  - RTL simulation
    - Golden RTL is built and simulated

Bus RTL Generation: AMBA Designer

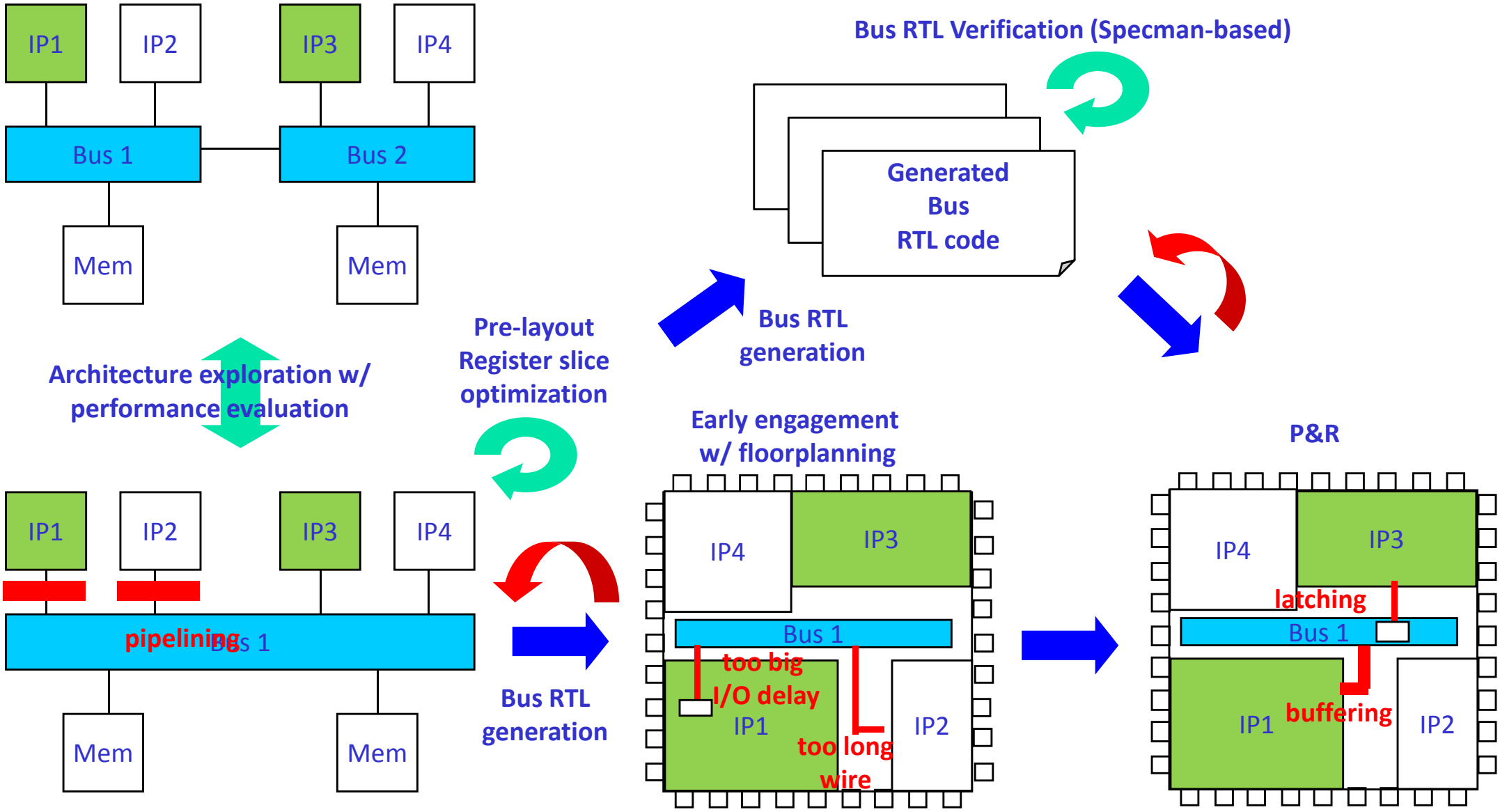
- Generated IP's
  - Interconnect, DMA, Static memory controller etc.
- Parameters
  - data width (32b ~ 128b), protocol (AXI, AHB, APB), etc.
  - Timing (register slice, address decode), security (trustzone), etc.
  - QoS (programmable QoS), arbitration schemes, etc.



Bus RTL Verification

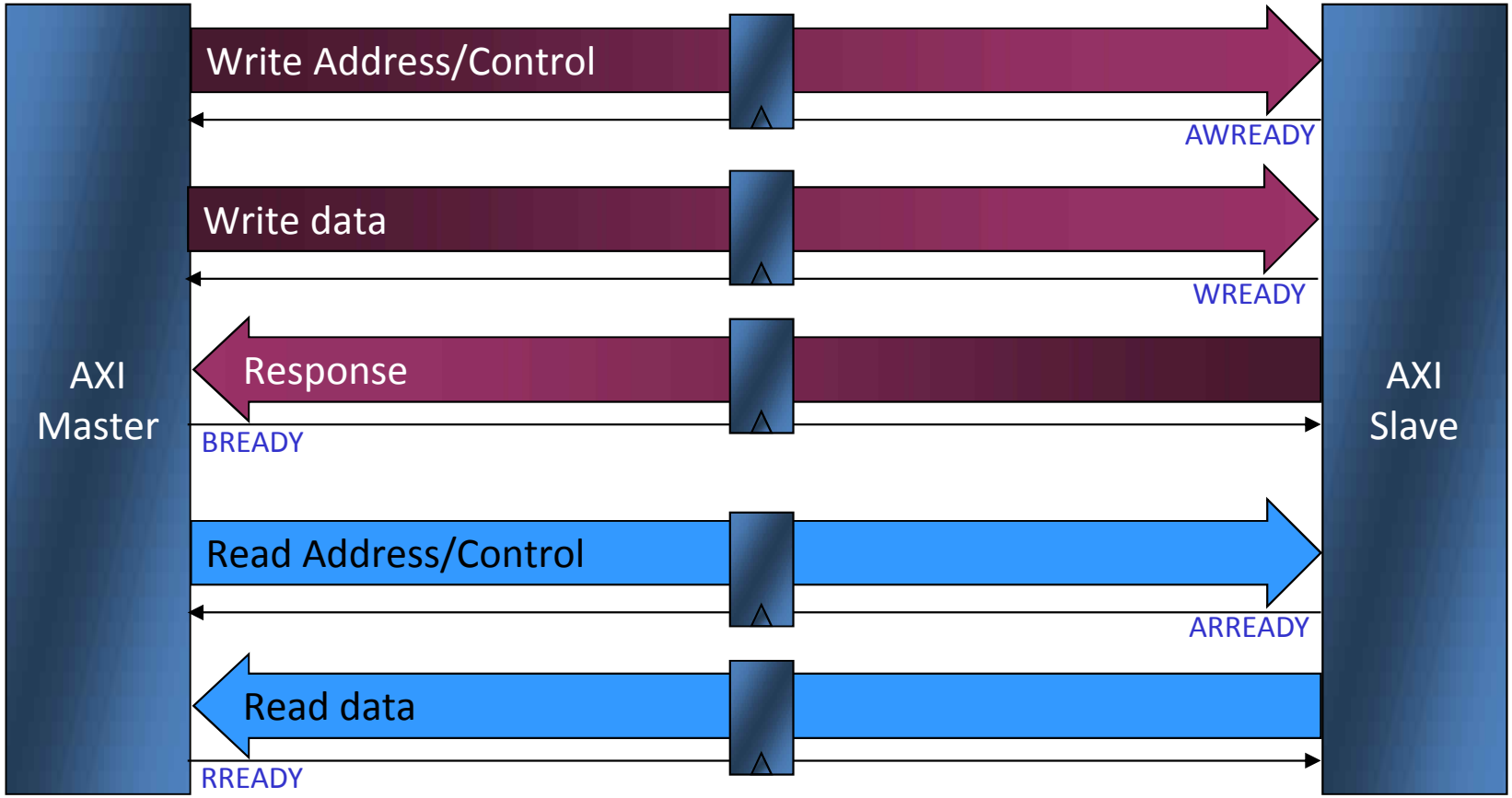
- Bus component verification
  - Verification of each components (PL30x, 340, bridges)
  - Method & criterion: Specman eVC & protocol coverage
- Bus architecture verification
  - Verification of the entire bus architecture
  - Method & criterion: Specman eVC + system scenarios & functional correctness + protocol coverage
- Called Platform Verifier
  - Under development by System LSI Design Technology team

On-Chip Network (OCN) Design Flow



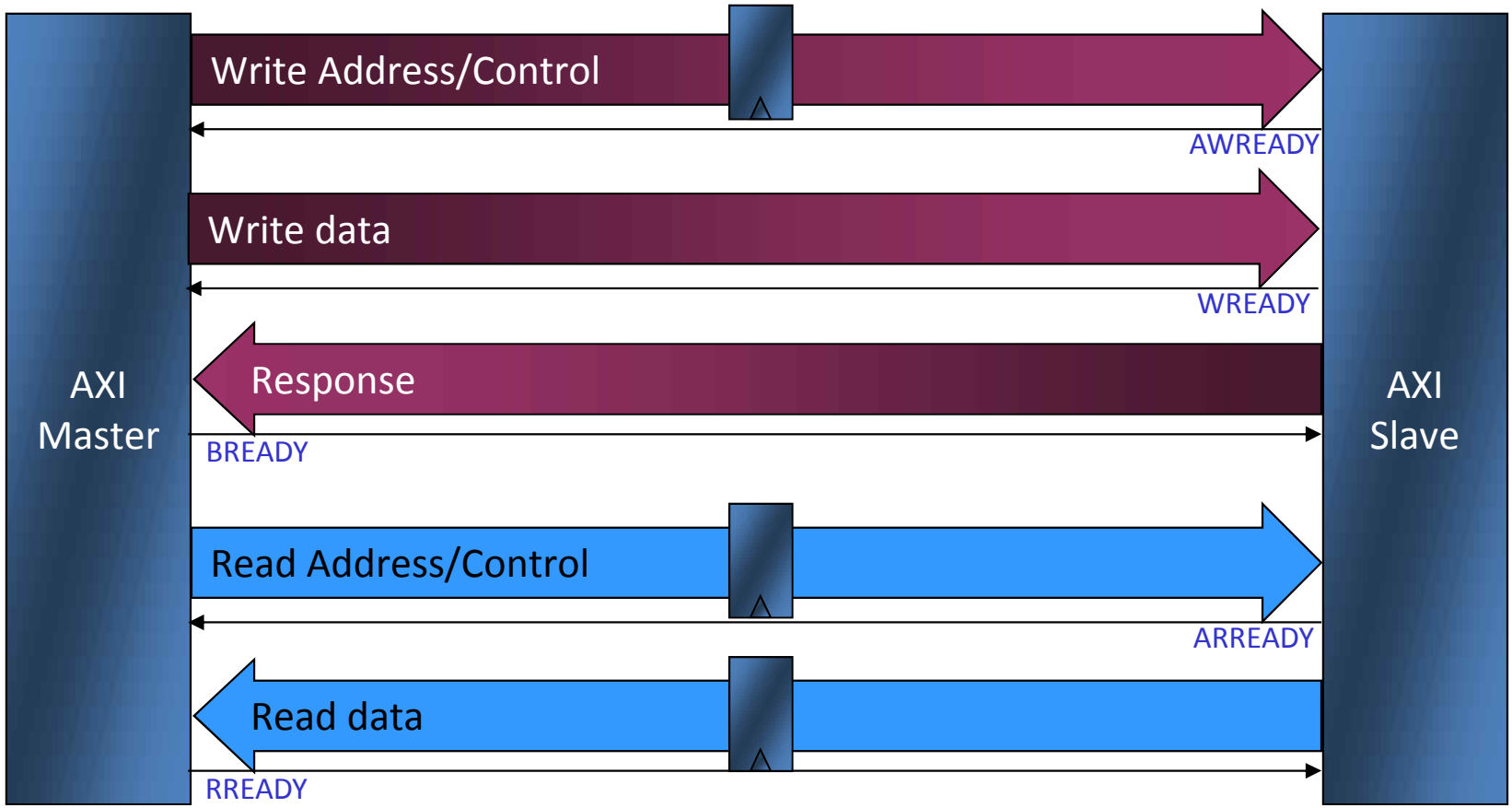
Register Slice for Timing Isolation

- Register slice can be used at any channel independently



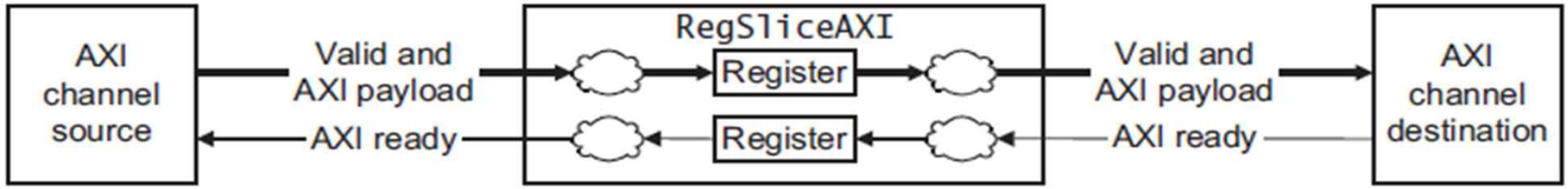
Register Slice for Timing Isolation

- Register slice incurs one cycle latency per insertion

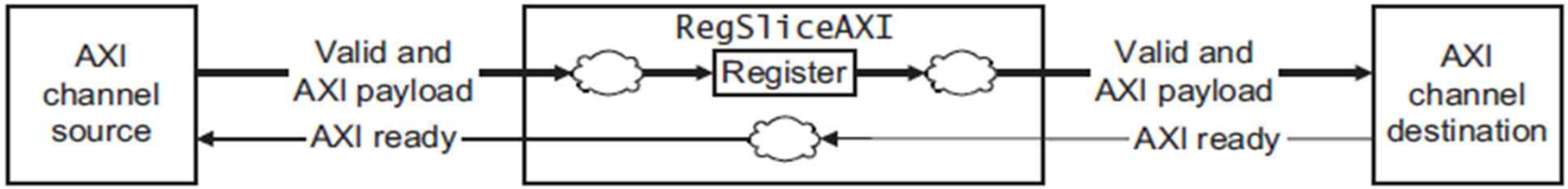


Two Modes: Full and Forward

- Area benefit: forward mode gives smaller area
- The same latency penalty



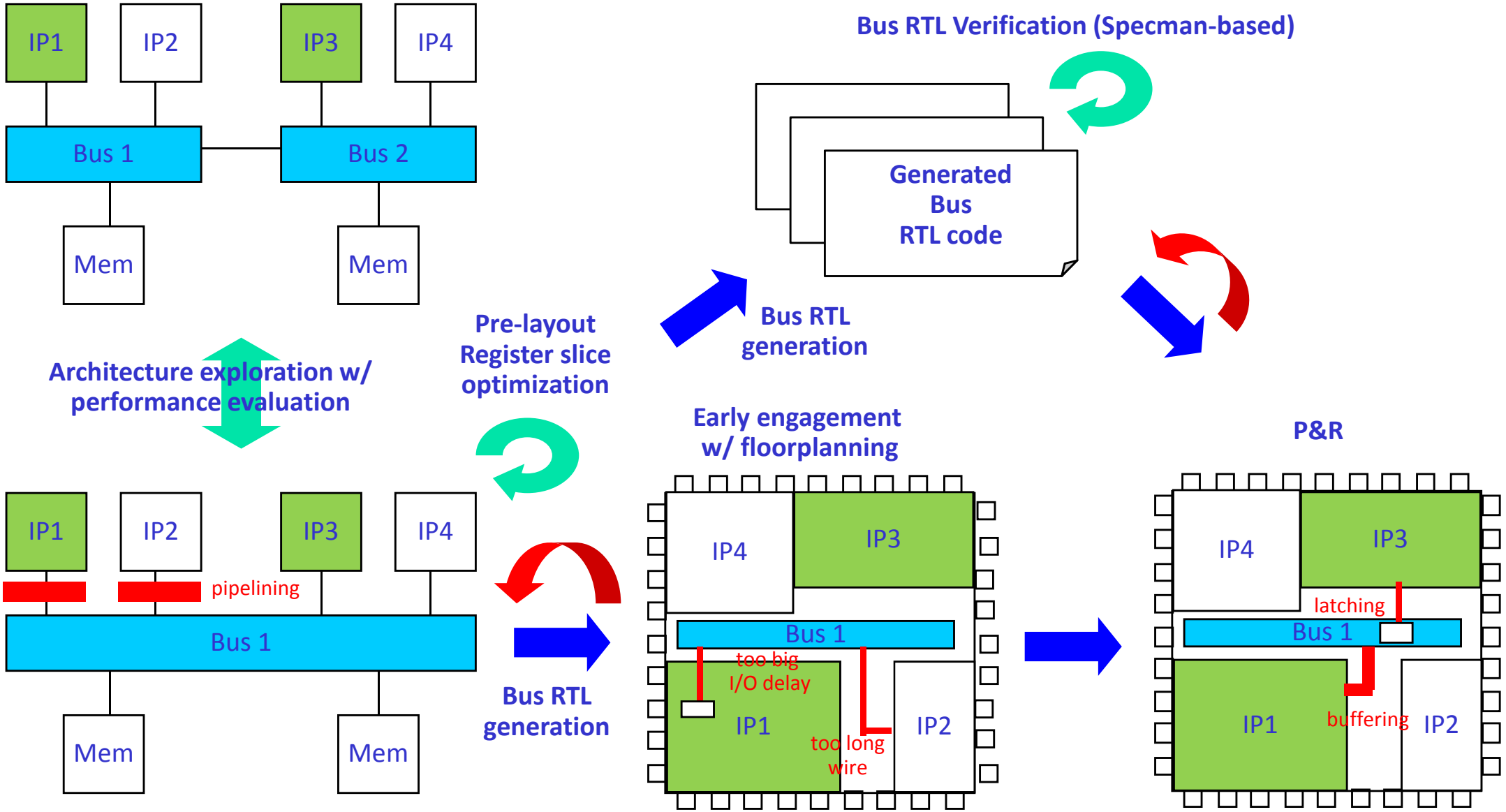
AXI register slice (full mode)



AXI register slice (forward mode)

[Source: PL301 TS]

On-Chip Network (OCN) Design Flow



Announcement

- Homework #5 (Due: Nov. 13)